<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Installation</td>
<td>3</td>
</tr>
<tr>
<td>2 Features</td>
<td>5</td>
</tr>
<tr>
<td>3 Basic Usage</td>
<td>7</td>
</tr>
<tr>
<td>4 Algorithm overview</td>
<td>9</td>
</tr>
<tr>
<td>5 Roadmap</td>
<td>13</td>
</tr>
<tr>
<td>6 Accumulated Local Effects</td>
<td>15</td>
</tr>
<tr>
<td>7 Anchors</td>
<td>21</td>
</tr>
<tr>
<td>8 Contrastive Explanation Method</td>
<td>29</td>
</tr>
<tr>
<td>9 Counterfactual Instances</td>
<td>35</td>
</tr>
<tr>
<td>10 Counterfactuals Guided by Prototypes</td>
<td>41</td>
</tr>
<tr>
<td>11 Integrated Gradients</td>
<td>49</td>
</tr>
<tr>
<td>12 Kernel SHAP</td>
<td>53</td>
</tr>
<tr>
<td>13 Measuring the linearity of machine learning models</td>
<td>65</td>
</tr>
<tr>
<td>14 Trust Scores</td>
<td>69</td>
</tr>
<tr>
<td>15 Tree SHAP</td>
<td>73</td>
</tr>
<tr>
<td>16 Accumulated Local Effects for predicting house prices</td>
<td>87</td>
</tr>
<tr>
<td>17 Accumulated Local Effects for classifying flowers</td>
<td>97</td>
</tr>
<tr>
<td>18 Anchor explanations for income prediction</td>
<td>107</td>
</tr>
<tr>
<td>19 Anchor explanations on the Iris dataset</td>
<td>113</td>
</tr>
<tr>
<td>20 Anchor explanations for movie sentiment</td>
<td>115</td>
</tr>
<tr>
<td>21 Anchor explanations for ImageNet</td>
<td>121</td>
</tr>
<tr>
<td>22 Anchor explanations for fashion MNIST</td>
<td>125</td>
</tr>
</tbody>
</table>
23 Contrastive Explanations Method (CEM) applied to MNIST 131
24 Contrastive Explanations Method (CEM) applied to Iris dataset 139
25 Counterfactual instances on MNIST 145
26 Counterfactuals guided by prototypes on MNIST 153
27 Counterfactuals guided by prototypes on Boston housing dataset 167
28 Counterfactual explanations with one-hot encoded categorical variables 171
29 Counterfactual explanations with ordinally encoded categorical variables 181
30 Kernel SHAP explanation for SVM models 189
31 Kernel SHAP explanation for multinomial logistic regression models 203
32 Handling categorical variables with KernelSHAP 217
33 KernelSHAP: combining preprocessor and predictor 249
34 Distributed KernelSHAP 261
35 Linearity measure applied to Iris 271
36 Linearity measure applied to fashion MNIST 283
37 Trust Scores applied to Iris 293
38 Trust Scores applied to MNIST 299
39 Explaining Tree Models with Interventional Feature Perturbation Tree SHAP 309
40 Explaining Tree Models with Path-Dependent Feature Perturbation Tree SHAP 329
41 Integrated gradients for a ResNet model trained on Imagenet dataset 349
42 Integrated gradients for MNIST 355
43 Integrated gradients for text classification on the IMDB dataset 361
44 alibi 367
45 Indices and tables 457
Python Module Index 459
Index 461
Alibi is an open source Python library aimed at machine learning model inspection and interpretation. The focus of the library is to provide high-quality implementations of black-box, white-box, local and global explanation methods for classification and regression models.
Alibi works with Python 3.6+ and can be installed from PyPI:

```sh
pip install alibi
```

Alternatively, the development version can be installed:

```sh
pip install git+https://github.com/SeldonIO/alibi.git
```
Alibi is a Python package designed to help explain the predictions of machine learning models and gauge the confidence of predictions. The focus of the library is to support the widest range of models using black-box methods where possible.

To get a list of the latest available model explanation algorithms, you can type:

```python
import alibi
alibi.explainers.__all__
```

For gauging model confidence:

```python
alibi.confidence.__all__
```

For detailed information on the methods:

- **Overview of available methods**
  - Accumulated Local Effects
  - Anchor explanations
  - Contrastive Explanation Method (CEM)
  - Counterfactual Instances
  - Counterfactuals Guided by Prototypes
  - Kernel SHAP
  - Integrated gradients
  - Linearity Measure
  - Trust Scores
The alibi explanation API takes inspiration from scikit-learn, consisting of distinct initialize, fit and explain steps. We will use the Anchor method on tabular data to illustrate the API.

First, we import the explainer:

```python
from alibi.explainers import AnchorTabular
```

Next, we initialize it by passing it a prediction function and any other necessary arguments:

```python
explainer = AnchorTabular(predict_fn, feature_names)
```

Some methods require an additional .fit step which requires access to the training set the model was trained on:

```python
explainer.fit(X_train)
```

```python
AnchorTabular(meta={
    'name': 'AnchorTabular',
    'type': ['blackbox'],
    'explanations': ['local'],
    'params': {'seed': None, 'disc_perc': (25, 50, 75)}
})
```

Finally, we can call the explainer on a test instance which will return an Explanation object containing the explanation and any additional metadata returned by the computation:

```python
explanation = explainer.explain(x)
```

The returned Explanation object has meta and data attributes which are dictionaries containing any explanation metadata (e.g. parameters, type of explanation) and the explanation itself respectively:

```python
explanation.meta
```

```python
['name': 'AnchorTabular',
 'type': ['blackbox'],
 'explanations': ['local'],
 'params': {'seed': None,
 'disc_perc': (25, 50, 75),
 'threshold': 0.95,
 'delta': ...truncated output...]
```

```python
explanation.data
```
The top level keys of both meta and data dictionaries are also exposed as attributes for ease of use of the explanation:

```python
explanation.anchor
```

```json
['petal width (cm) > 1.80', 'sepal width (cm) <= 2.80']
```

Some algorithms, such as Kernel SHAP, can run batches of explanations in parallel, if the number of cores is specified in the algorithm constructor:

```python
distributed_ks = KernelShap(predict_fn, distributed_opts={'n_cpus': 10})
```

Note that this requires the user to run `pip install alibi[ray]` to install dependencies of the distributed back-end.

The exact details will vary slightly from method to method, so we encourage the reader to become familiar with the types of algorithms supported in Alibi.
This page provides a high-level overview of the algorithms and their features currently implemented in Alibi.

### 4.1 Model Explanations

These algorithms provide instance-specific (sometimes also called local) explanations of ML model predictions. Given a single instance and a model prediction they aim to answer the question “Why did my model make this prediction?” Most of the following algorithms work with black-box models meaning that the only requirement is to have access to a prediction function (which could be an API endpoint for a model in production).

The following table summarizes the capabilities of the current algorithms:

<table>
<thead>
<tr>
<th>Method</th>
<th>Models</th>
<th>Exp. types</th>
<th>Classification</th>
<th>Regression</th>
<th>Tabular</th>
<th>Text</th>
<th>Image</th>
<th>Cat. data</th>
<th>Train</th>
<th>Dist.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALE</td>
<td>BB</td>
<td>global</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anchors</td>
<td>BB</td>
<td>local</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>For</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEM</td>
<td>BB*</td>
<td>local</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counterfactuals</td>
<td>BB*</td>
<td>local</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prototype Counterfactuals</td>
<td>BB*</td>
<td>local</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrated Gradients</td>
<td>TF/Keras</td>
<td>local</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel SHAP</td>
<td>BB</td>
<td>local</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tree SHAP</td>
<td>WB</td>
<td>local</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key:
- **BB** - black-box (only require a prediction function)
- **BB** - black-box but assume model is differentiable
- **WB** - requires white-box model access. There may be limitations on models supported
- **TF/Keras** - TensorFlow models via the Keras API
- **Local** - instance specific explanation, why was this prediction made?
• **Global** - explains the model with respect to a set of instances
• **Cat. data** - support for categorical features
• **Train** - whether a training set is required to fit the explainer
• **Dist.** - whether a batch of explanations can be executed in parallel

**Accumulated Local Effects (ALE):** calculates first-order feature effects on the model with respect to a dataset. Intended for use on tabular datasets, currently supports numerical features. *Documentation, regression example, classification example.*

**Anchor Explanations:** produce an “anchor” - a small subset of features and their ranges that will almost always result in the same model prediction. *Documentation, tabular example, text classification, image classification.*

**Contrastive Explanation Method (CEM):** produce a pertinent positive (PP) and a pertinent negative (PN) instance. The PP instance finds the features that should be minimally and sufficiently present to predict the same class as the original prediction (a PP acts as the “most compact” representation of the instance to keep the same prediction). The PN instance identifies the features that should be minimally and necessarily absent to maintain the original prediction (a PN acts as the closest instance that would result in a different prediction). *Documentation, tabular example, image classification.*

**Counterfactual Explanations:** generate counterfactual examples using a simple loss function. *Documentation, image classification.*

**Counterfactual Explanations Guided by Prototypes:** generate counterfactuals guided by nearest class prototypes other than the class predicted on the original instance. This method can speed up the search, especially for black box models, and create interpretable counterfactuals. *Documentation, tabular example, tabular example with categorical features, image classification.*

**Integrated gradients:** attribute an importance score to each element of the input or an internal layer of the the model with respect to a given baseline. The attributions are calculated as the path integral of the model gradients along a straight line from the baseline to the input. *Documentation, MNIST example, Imagenet example, IMDB example.*

**Kernel Shapley Additive Explanations (Kernel SHAP):** attribute the change of a model output with respect to a given baseline (e.g., average over a reference set) to each of the input features. This is achieved for each feature in turn, by averaging the difference in the model output observed when the feature whose contribution is to be estimated is part of a group of “present” input features and the value observed when the feature is excluded from said group. The features that are not “present” (i.e., are missing) are replaced with values from a background dataset. This algorithm can be used to explain regression models and it is optimised to distribute batches of explanations. *Documentation, continuous data, more continuous data, categorical data, distributed_batch_explanations.*

**Tree Shapley Additive Explanations (Tree SHAP):** attribute the change of a model output with respect to a baseline (e.g., average over a reference set or inferred from node data) to each of the input features. Similar to Kernel SHAP, the shap value of each feature is computed by averaging the difference of the model output observed when the feature is part of a group of “present” features and when the feature is excluded from said group, over all possible subsets of “present” features. Different estimation procedures for the effect of selecting different subsets of “present” features on the model output give rise to the interventional feature perturbation and the path-dependent feature perturbation variants of Tree SHAP. This algorithm can be used to explain regression models. *Documentation, interventional feature perturbation Tree SHAP, path-dependent feature perturbation Tree SHAP.*
4.2 Model Confidence

These algorithms provide instance-specific scores measuring the model confidence for making a particular prediction.

<table>
<thead>
<tr>
<th>Method</th>
<th>Models</th>
<th>Classification</th>
<th>Regression</th>
<th>Tabular</th>
<th>Text</th>
<th>Images</th>
<th>Categorical Features</th>
<th>Train set required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trust Scores</td>
<td>BB</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Linearity Measure</td>
<td>BB</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>Optional</td>
</tr>
</tbody>
</table>

**Trust scores**: produce a “trust score” of a classifier’s prediction. The trust score is the ratio between the distance to the nearest class different from the predicted class and the distance to the predicted class, higher scores correspond to more trustworthy predictions. Documentation, tabular example, image classification

**Linearity measure**: produces a score quantifying how linear the model is around a test instance. The linearity score measures the model linearity around a test instance by feeding the model linear superpositions of inputs and comparing the outputs with the linear combination of outputs from predictions on single inputs. Documentation Tabular example, image classification

---

1 depending on model
2 may require dimensionality reduction
Alibi aims to be the go-to library for ML model interpretability. There are multiple challenges for developing a high quality, production-ready library that achieves this. In addition to having high quality reference implementations of the most promising algorithms, we need extensive documentation and case studies comparing the different interpretability methods and their respective pros and cons. A clean and a usable API is also a priority.

5.1 Short term

- Complete refactoring to enable multiple backends (TensorFlow, PyTorch) and distributed computing
- AnchorText improvements using generative models

5.2 Medium term

- PyTorch support for white-box gradient based explanations
- Improve black-box counterfactual explanations using gradient-free methods

5.3 Long term

- Ongoing optimizations of existing algorithms (speed, parallelisation, explanation quality)
- Explanations for sequential and structured data

[source]
ACCUMULATED LOCAL EFFECTS

6.1 Overview

Accumulated Local Effects (ALE) is a method for computing feature effects based on the paper Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models by Apley and Zhu. The algorithm provides model-agnostic (black box) global explanations for classification and regression models on tabular data.

ALE addresses some key shortcomings of Partial Dependence Plots (PDP), a popular method for estimating first order feature effects. We discuss these limitations and motivate ALE after presenting the method usage.

6.2 Usage

Initialize the explainer by passing a black-box prediction function and optionally a list of feature names and target (class) names for interpretation:

```python
from alibi.explainers import ALE
ale = ALE(predict_fn, feature_names=feature_names, target_names=target_names)
```

Following the initialization, we can immediately produce an explanation given a dataset of instances $X$:

```python
exp = ale.explain(X)
```

The `explain` method has a default argument, `min_bin_points=4`, which determines the number of bins the range of each feature is subdivided into so that the ALE estimate for each bin is made with at least `min_bin_points`. Smaller values can result in less accurate local estimates while larger values can also result in less accurate estimates by averaging across large parts of the feature range.

Alternatively, we can run the explanation only on a subset of features:

```python
exp = ale.explain(X, features=[0, 1])
```

This is useful if the number of total features is large and only small number is of interest. Also, it can be particularly useful to filter out categorical variable columns as there is no consistent ALE formulation and hence any results for categorical variables would be misleading.

The result `exp` is an `Explanation` object which contains the following data-related attributes:

- `ale_values` - a list of arrays of ALE values (one for each feature). Each array can have multiple columns (if the number of targets is >1 as in classification)
- `constant_value` - the mean prediction over $X$ (zeroth order effects)
\begin{itemize}
\item \texttt{ale0} - a list of “centering” values (one for each feature) used by the algorithm to center the \texttt{ale_values} around the expected effect for the feature (i.e. the sum of \texttt{ale_values} and \texttt{ale0} will be the uncentered ALE)
\item \texttt{feature_values} - a list of arrays (one for each feature) of feature values at which the ALE values were computed
\item \texttt{feature_names} - a list of feature names
\item \texttt{target_names} - a list of target names
\item \texttt{feature_deciles} - a list of arrays (one for each feature) of the feature deciles
\end{itemize}

Plotting \texttt{ale_values} against \texttt{feature_values} recovers the ALE curves. For convenience we include a plotting function \texttt{plot_ale} which automatically produces ALE plots using \texttt{matplotlib}:

```python
from alibi.explainers import plot_ale
plot_ale(exp)
```

The following is an example ALE plot of a logistic regression model on the Iris dataset (see worked \textit{example}):

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{ale_example.png}
\end{figure}

\section*{6.3 Examples}

\textit{ALE regression example (Boston house prices)}

\textit{ALE classification example (Iris dataset)}
6.4 Motivation and definition

The following exposition largely follows Apley and Zhu (2016) and Molnar (2019).

Given a predictive model $f(x)$ where $x = (x_1, \ldots, x_d)$ is a vector of $d$ features, we are interested in computing the feature effects of each feature $x_i$ on the model $f(x)$. A feature effect of feature $x_i$ is some function $g(x_i)$ designed to disentangle the contribution of $x_i$ to the response $f(x)$. To simplify notation, in the following we consider the $d = 2$ case and define the feature effect functions for the first feature $x_1$.

6.4.1 Partial Dependence

Partial Dependence Plots (PDP) is a very common method for computing feature effects. It is defined as

$$
\text{PD}(x_1) = E[f(x_1, X_2)] = \int p(x_2)f(x_1, x_2)dx_2,
$$

where $p(x_2)$ is the marginal distribution of $X_2$. To estimate the expectation, we can take the training set $X$ and average the predictions of instances where the first feature for all instances is replaced by $x_1$:

$$
\hat{\text{PD}}(x_1) = \frac{1}{n} \sum_{j=1}^{n} f(x_1, x_{2,j}).
$$

The PD function attempts to calculate the effect of $x_1$ by averaging the effects of the other feature $x_2$ over it’s marginal distribution. This is problematic because by doing so we are averaging predictions of many out of distribution instances. For example, if $x_1$ and $x_2$ are a person’s height and weight and $f$ predicts some other attribute of the person, then the PD function at a fixed height $x_1$ would average predictions of persons with height $x_1$ and all possible weights $x_2$ observed in the training set. Clearly, since height and weight are strongly correlated this would lead to many unrealistic data points. Since the predictor $f$ has not been trained on such impossible data points, the predictions are no longer meaningful. We can say that an implicit assumption motivating the PD approach is that the features are uncorrelated, however this is rarely the case and severely limits the usage of PDP.

An attempt to fix the issue with the PD function is to average over the conditional distribution instead of the marginal which leads to the following feature effect function:

$$
M(x_1) = E[f(X_1, X_2)|X_1 = x_1] = \int p(x_2|x_1)f(x_1, x_2)dx_2,
$$

where $p(x_2|x_1)$ is the conditional distribution of $X_2$. To estimate this function from the training set $X$ we can compute

$$
\hat{M}(x_1) = \frac{1}{n(x_1)} \sum_{j \in N(x_1)} f(x_1, x_{2,j}),
$$

where $N(x_1)$ is a subset of indices $j$ for which $x_{1,j}$ falls into some small neighbourhood of $x_1$ and $n(x_1)$ is the number of such instances.

While this refinement addresses the issue of the PD function averaging over impossible data points, the use of the $M(x_1)$ function as feature effects remains limited when the features are correlated. To go back to the example with people’s height and weight, if we fix the height to be some particular value $x_1$ and calculate the effects according to $M(x_1)$, because of the correlation of height and weight the function value mixes effects of both features and estimates the combined effect. This is undesirable as we cannot attribute the value of $M(x_1)$ purely to height. Furthermore, suppose height doesn’t actually have any effect on the prediction, only weight does. Because of the correlation between height and weight, $M(x_1)$ would still show an effect which can be highly misleading. Concretely, for a model like $f(x_1, x_2) = x_2$ it is possible that $M(x_1) \neq 0$ if $x_1, x_2$ are correlated.

The following plot summarizes the two approaches for estimating the effect of $x_1$ at a particular value when $x_2$ is strongly correlated with $x_1$:
6.4.2 ALE

ALE solves the problem of mixing effects from different features. As with the function $M(x_1)$, ALE uses the conditional distribution to average over other features, but instead of averaging the predictions directly, it averages differences in predictions to block the effect of correlated features. The ALE function is defined as follows:

$$\text{ALE}(x_1) = \int_{\min(x_1)}^{x_1} \mathbb{E} \left[ \frac{\partial f(X_1, X_2)}{\partial X_1} \bigg| X_1 = z_1 \right] dz_1 - c_1$$

(6.1)

$$= \int_{\min(x_1)}^{x_1} \int p(x_2 | z_1) \frac{\partial f(z_1, x_2)}{\partial z_1} dx_2 dz_1 - c_1,$$

(6.2)

where the constant $c_1$ is chosen such that the resulting ALE values are independent of the point $\min(x_1)$ and have zero mean over the distribution $p(x_1)$.

The term $\frac{\partial f(x_1, x_2)}{\partial x_1}$ is called the local effect of $x_1$ on $f$. Averaging the local effect over the conditional distribution $p(x_2 | x_1)$ allows us to isolate the effect of $x_1$ from the effects of other correlated features avoiding the issue of $M$ plots which directly average the predictor $f$. Finally, note that the local effects are integrated over the range of $x_1$, this corresponds to the accumulated in ALE. This is done as a means of visualizing the global effect of the feature by “piecing together” the calculated local effects.

In practice, we calculate the local effects by finite differences so the predictor $f$ need not be differentiable. Thus, to estimate the ALE from data, we compute the following:

$$\widehat{\text{ALE}}(x_1) = \sum_{k=1}^{k(x_1)} \frac{1}{n(k)} \sum_{i: x_1(i) \in N(k)} \left[ f(z_k, x_{1}^{(i)}) - f(z_{k-1}, x_{1}^{(i)}) \right] - c_1.$$  

Here $z_0, z_1, \ldots$ is a sufficiently fine grid of the feature $x_1$ (typically quantiles so that each resulting interval contains a similar number of points), $N(k)$ denotes the interval $[z_{k-1}, z_k]$, $n(k)$ denotes the number of points falling into interval $N(k)$ and $k(x_1)$ denotes the index of the interval into which $x_1$ falls into, i.e. $x_1 \in [z_{k(x_1)-1}, z_{k(x_1)})$. Finally, the notation $f(z_k, x_{1}^{(i)})$ means that for instance $i$ we replace $x_1$ with the value of the right interval end-point $z_k$ (likewise for the left interval end-point using $z_{k-1}$), leaving the rest of the features unchanged, and evaluate the difference of predictions at these points.

The following plot illustrates the ALE estimation process. We have subdivided the feature range of $x_1$ into 5 bins with roughly the same number of points indexed by $N(k)$. Focusing on bin $N(4)$, for each point falling into this bin, we
replace their $x_1$ feature value by the left and right end-points of the interval, $z_3$ and $z_4$. Then we evaluate the difference of the predictions of these points and calculate the average by dividing by the number of points in this interval $n(4)$. We do this for every interval and sum up (accumulate) the results. Finally, to calculate the constant $c_1$, we subtract the expectation over $p(x_1)$ of the calculated uncentered ALE so that the resulting ALE values have mean zero over the distribution $p(x_1)$.

![ALE estimation diagram](image)

We show the results of ALE calculation for a model $f(x_1, x_2) = 3x_1 + 2x_2^2$. The resulting plots correctly recover the linear effect of $x_1$ and the quadratic effect of $x_2$ on $f$. Note that the ALE is estimated for each interval edge and linearly interpolated in between, for real applications it is important to have a sufficiently fine grid but also one that has enough points into each interval for accurate estimates. The x-axis also shows feature deciles of the feature to help judge in which parts of the feature space the ALE plot is interpolating more and the estimate might be less trustworthy.

The value of ALE$(x_i)$ is the main effect of feature $x_i$ as compared to the average prediction for the data. For example, the value of ALE$(x_1)$ = 0.75 at $x_1 = 0.7$, if we sample data from the joint distribution $p(x_1, x_2)$ (i.e. realistic data points) and $x_1 = 0.7$, then we would expect the first order effect of feature $x_1$ to be 0.75 higher than the average first order effect of this feature. Seeing that the ALE$(x_1)$ plot crosses zero at $x_1 \approx 0.45$, realistic data points with $x_1 \approx 0.45$ will have effect on $f$ similar to the average first order effect of $x_1$. For realistic data points with smaller $x_1$, the effect will become negative with respect to the average effect.

![ALE plots](image)

Because the model $f(x_1, x_2) = 3x_1 + 2x_2^2$ is explicit and differentiable, we can calculate the ALE functions analytically which gives us even more insight. The partial derivatives are given by $(3, 4x_2)$. Assuming that the conditional distributions $p(x_2|x_1)$ and $p(x_1|x_2)$ are uniform, the expectations over the conditional distributions are equal to the...
partial derivatives. Next, we integrate over the range of the features to obtain the uncentered ALE functions:

\[
ALE_u(x_1) = \int_{\min(x_1)}^{x_1} 3dz_1 = 3x_1 - 3 \min(x_1) \tag{6.3}
\]

\[
ALE_u(x_2) = \int_{\min(x_2)}^{x_2} 4z_2dz_2 = 2x_2^2 - 2 \min(x_2)^2. \tag{6.4}
\]

Finally, to obtain the ALE functions, we center by setting \( c_i = \mathbb{E}(ALE_u(x_i)) \) where the expectation is over the marginal distribution \( p(x_i) \):

\[
ALE(x_1) = 3x_1 - 3 \min(x_1) - \mathbb{E}(3x_1 - 3 \min(x_1)) = 3x_1 - 3\mathbb{E}(x_1) \tag{6.5}
\]

\[
ALE(x_2) = 2x_2^2 - 2 \min(x_2)^2 - \mathbb{E}(2x_2^2 - 2 \min(x_2)^2) = 2x_2^2 - 2\mathbb{E}(x_2^2). \tag{6.6}
\]

This calculation verifies that the ALE curves are the desired feature effects (linear for \( x_1 \) and quadratic for \( x_2 \)) relative to the mean feature effects across the dataset. In fact if \( f \) is additive in the individual features like our toy model, then the ALE main effects recover the correct additive components (Apley and Zhu (2016)). Furthermore, for additive models we have the decomposition \( f(x) = \mathbb{E}(f(x)) + \sum_{i=1}^{d} ALE(x_i) \), here the first term which is the average prediction across the dataset \( X \) can be thought of as zeroth order effects.
7.1 Overview

The anchor algorithm is based on the Anchors: High-Precision Model-Agnostic Explanations paper by Ribeiro et al. and builds on the open source code from the paper’s first author.

The algorithm provides model-agnostic (black box) and human interpretable explanations suitable for classification models applied to images, text and tabular data. The idea behind anchors is to explain the behaviour of complex models with high-precision rules called anchors. These anchors are locally sufficient conditions to ensure a certain prediction with a high degree of confidence.

Anchors address a key shortcoming of local explanation methods like LIME which proxy the local behaviour of the model in a linear way. It is however unclear to what extent the explanation holds up in the region around the instance to be explained, since both the model and data can exhibit non-linear behaviour in the neighborhood of the instance. This approach can easily lead to overconfidence in the explanation and misleading conclusions on unseen but similar instances. The anchor algorithm tackles this issue by incorporating coverage, the region where the explanation applies, into the optimization problem. A simple example from sentiment classification illustrates this (Figure 1). Dependent on the sentence, the occurrence of the word not is interpreted as positive or negative for the sentiment by LIME. It is clear that the explanation using not is very local. Anchors however aim to maximize the coverage, and require not to occur together with good or bad to ensure respectively negative or positive sentiment.

![Figure 1: Sentiment predictions, LSTM](image)

Ribeiro et al., Anchors: High-Precision Model-Agnostic Explanations, 2018
As highlighted by the above example, an anchor explanation consists of *if-then rules*, called the anchors, which sufficiently guarantee the explanation locally and try to maximize the area for which the explanation holds. This means that as long as the anchor holds, the prediction should remain the same regardless of the values of the features not present in the anchor. Going back to the sentiment example: as long as *not good* is present, the sentiment is negative, regardless of the other words in the movie review.

### 7.1.1 Text

For text classification, an interpretable anchor consists of the words that need to be present to ensure a prediction, regardless of the other words in the input. The words that are not present in a candidate anchor can be sampled in 2 ways:

- Replace word token by UNK token.
- Replace word token by sampled token from a corpus with the same POS tag and probability proportional to the similarity in the embedding space. By sampling similar words, we keep more context than simply using the UNK token.

### 7.1.2 Tabular Data

Anchors are also suitable for tabular data with both categorical and continuous features. The continuous features are discretized into quantiles (e.g., deciles), so they become more interpretable. The features in a candidate anchor are kept constant (same category or bin for discretized features) while we sample the other features from a training set. As a result, anchors for tabular data need access to training data. Let’s illustrate this with an example. Say we want to predict whether a person makes less or more than £50,000 per year based on the person’s characteristics including age (continuous variable) and marital status (categorical variable). The following would then be a potential anchor: Hugo makes more than £50,000 because he is married and his age is between 35 and 45 years.

### 7.1.3 Images

Similar to LIME, images are first segmented into superpixels, maintaining local image structure. The interpretable representation then consists of the presence or absence of each superpixel in the anchor. It is crucial to generate meaningful superpixels in order to arrive at interpretable explanations. The algorithm supports a number of standard image segmentation algorithms (*felzenszwalb, slic* and *quickshift*) and allows the user to provide a custom segmentation function.

The superpixels not present in a candidate anchor can be masked in 2 ways:

- Take the average value of that superpixel.
- Use the pixel values of a superimposed picture over the masked superpixels.
7.1.4 Efficiently Computing Anchors

The anchor needs to return the same prediction as the original instance with a minimal confidence of e.g. 95%. If multiple candidate anchors satisfy this constraint, we go with the anchor that has the largest coverage. Because the number of potential anchors is exponential in the feature space, we need a faster approximate solution.

The anchors are constructed bottom-up in combination with beam search. We start with an empty rule or anchor, and incrementally add an if-then rule in each iteration until the minimal confidence constraint is satisfied. If multiple valid anchors are found, the one with the largest coverage is returned.

In order to select the best candidate anchors for the beam width efficiently during each iteration, we formulate the problem as a pure exploration multi-armed bandit problem. This limits the number of model prediction calls which can be a computational bottleneck.

For more details, we refer the reader to the original paper.

7.2 Usage

While each data type has specific requirements to initialize the explainer and return explanations, the underlying algorithm to construct the anchors is the same.

In order to efficiently generate anchors, the following hyperparameters need to be set to sensible values when calling the explain method:

- **threshold**: the previously discussed minimal confidence level. threshold defines the minimum fraction of samples for a candidate anchor that need to lead to the same prediction as the original instance. A higher value gives more confidence in the anchor, but also leads to more computation time. The default value is 0.95.

- **tau**: determines when we assume convergence for the multi-armed bandit. A bigger value for tau means faster convergence but also looser anchor conditions. By default equal to 0.15.

- **beam_size**: the size of the beam width. A bigger beam width can lead to a better overall anchor at the expense of more computation time.

- **batch_size**: the batch size used for sampling. A bigger batch size gives more confidence in the anchor, again at the expense of computation time since it involves more model prediction calls. The default value is 100.

- **coverage_samples**: number of samples used to compute the coverage of the anchor. By default set to 10000.

7.2.1 Text

Initialization

Since the explainer works on black box models, only access to a predict function is needed. The model below is a simple logistic regression trained on movie reviews with negative or positive sentiment and pre-processed with a CountVectorizer:

```python
predict_fn = lambda x: clf.predict(vectorizer.transform(x))
```

If we choose to sample similar words from a corpus, we first need to load a spaCy model:
```python
import spacy
from alibi.utils.download import spacy_model

model = 'en_core_web_md'
spacy_model(model=model)
nlp = spacy.load(model)

We can now initialize our explainer:

explainer = AnchorText(nlp, predict_fn)
```

### Explanation

Let’s define the instance we want to explain and verify that the sentiment prediction on the original instance is positive:

```python
text = 'This is a good book .'
class_names = ['negative', 'positive']
pred = class_names[predict_fn([text])[0]]
```

Now we can explain the instance:

```python
explanation = explainer.explain(text, threshold=0.95, use_similarity_proba=False, use_unk=True, sample_proba=0.5)
```

We set the confidence threshold at 95%. use_unk equals True means that we replace words outside of the candidate anchor with UNK tokens with a sample probability equal to sample_proba. Instead of using UNK tokens, we can sample from the top_n similar words to the ground truth word in the corpus by setting use_unk to False.

```python
explanation = explainer.explain(text, threshold=0.95, use_unk=False, sample_proba=0.5, top_n=100)
```

It is also possible to sample words from the corpus proportional to the word similarity with the ground truth word by setting use_similarity_proba to True and use_unk to False. We can put more weight on similar words by decreasing the temperature argument. The following explanation perturbs original tokens with probability equal to sample_proba. The perturbed tokens are then sampled from the top_n most similar tokens in the corpus with sample probability proportional to the word similarity with the original token.

```python
explanation = explainer.explain(text, threshold=0.95, use_similarity_proba=True, use_unk=False, sample_proba=0.5, top_n=20, temperature=0.2)
```

The explain method returns an Explanation object with the following attributes:

- **anchor**: a list of words in the anchor.
- **precision**: the fraction of times the sampled instances where the anchor holds yields the same prediction as the original instance. The precision will always be ≥ threshold for a valid anchor.
- **coverage**: the fraction of sampled instances the anchor applies to.

The raw attribute is a dictionary which also contains example instances where the anchor holds and the prediction is the same as on the original instance, as well as examples where the anchor holds but the prediction changed to give the user a sense of where the anchor fails. raw also stores information on the anchor, precision and coverage of partial anchors. This allows the user to track the improvement in for instance the precision as more features (words in the case of text) are added to the anchor.
7.2.2 Tabular Data

Initialization and fit

To initialize the explainer, we provide a predict function, a list with the feature names to make the anchors easy to understand as well as an optional mapping from the encoded categorical features to a description of the category. An example for categorical_names would be category_map = {0: list('married', 'divorced'), 3: list('high school diploma', 'master’s degree')}. Each key in category_map refers to the column index in the input for the relevant categorical variable, while the values are lists with the options for each categorical variable. To make it easy, we provide a utility function gen_category_map to generate this map automatically from a Pandas dataframe:

```python
from alibi.utils.data import gen_category_map
category_map = gen_category_map(df)
```

Then initialize the explainer:

```python
predict_fn = lambda x: clf.predict(preprocessor.transform(x))
explainer = AnchorTabular(predict_fn, feature_names, categorical_names=category_map)
```

Tabular data requires a fit step to map the ordinal features into quantiles and therefore needs access to a representative set of the training data. disc_perc is a list with percentiles used for binning:

```python
explainer.fit(X_train, disc_perc=[25, 50, 75])
```

Explanation

Let’s check the prediction of the model on the original instance and explain:

```python
class_names = ['<=50K', '>50K']
pred = class_names[explainer.predict_fn(X)[0]]
explanation = explainer.explain(X, threshold=0.95)
```

The returned Explanation object contains the same attributes as the text explainer, so you could explain a prediction as follows:

```
Prediction: <=50K
Anchor: Marital Status = Never-Married AND Relationship = Own-child
Precision: 1.00
Coverage: 0.13
```

7.2.3 Images

Initialization

Besides the predict function, we also need to specify either a built in or custom superpixel segmentation function. The built in methods are felzenszwalb, slic and quickshift. It is important to create sensible superpixels in order to speed up convergence and generate interpretable explanations. Tuning the hyperparameters of the segmentation method is recommended.

```python
explainer = AnchorImage(predict_fn, image_shape, segmentation_fn='slic',
segmentation_kwargs={'n_segments': 15, 'compactness': 20, 'sigma': .5},
images_background=None)
```
Example of superpixels generated for the Persian cat picture using the *slic* method:

![Persian cat superpixels](image1.png)

The following function would be an example of a custom segmentation function dividing the image into rectangles.

```python
def superpixel(image, size=(4, 7)):
    segments = np.zeros([image.shape[0], image.shape[1]])
    row_idx, col_idx = np.where(segments == 0)
    for i, j in zip(row_idx, col_idx):
        segments[i, j] = int((image.shape[1]/size[1]) * (i//size[0]) + j//size[1])
    return segments
```

The *images_background* parameter allows the user to provide images used to superimpose on the masked superpixels, not present in the candidate anchor, instead of taking the average value of the masked superpixel. The superimposed images need to have the same shape as the explained instance.

**Explanation**

We can then explain the instance in the usual way:

```python
explanation = explainer.explain(image, p_sample=.5)
```

*p_sample* determines the fraction of superpixels that are either changed to the average superpixel value or that are superimposed.

The *Explanation* object again contains information about the anchor’s *precision*, *coverage* and examples where the anchor does or does not hold. On top of that, it also contains a masked image with only the anchor superpixels visible under the *anchor* attribute (see image below) as well as the image’s superpixels under *segments*.
7.3 Examples

7.3.1 Image

Anchor explanations for ImageNet
Anchor explanations for fashion MNIST

7.3.2 Tabular Data

Anchor explanations on the Iris dataset
Anchor explanations for income prediction

7.3.3 Text

Anchor explanations for movie sentiment
[source]
CHAPTER
EIGHT

CONTRASTIVE EXPLANATION METHOD

8.1 Overview

The Contrastive Explanation Method (CEM) is based on the paper Explanations based on the Missing: Towards Contrastive Explanations with Pertinent Negatives and extends the code open sourced by the authors. CEM generates instance based local black box explanations for classification models in terms of Pertinent Positives (PP) and Pertinent Negatives (PN). For a PP, the method finds the features that should be minimally and sufficiently present (e.g. important pixels in an image) to predict the same class as on the original instance. PN’s on the other hand identify what features should be minimally and necessarily absent from the instance to be explained in order to maintain the original prediction class. The aim of PN’s is not to provide a full set of characteristics that should be absent in the explained instance, but to provide a minimal set that differentiates it from the closest different class. Intuitively, the Pertinent Positives could be compared to Anchors while Pertinent Negatives are similar to Counterfactuals. As the authors of the paper state, CEM can generate clear explanations of the form: “An input $x$ is classified in class $y$ because features $f_1, \ldots, f_k$ are present and because features $f_m, \ldots, f_p$ are absent.” The current implementation is most suitable for images and tabular data without categorical features.

In order to create interpretable PP’s and PN’s, feature-wise perturbation needs to be done in a meaningful way. To keep the perturbations sparse and close to the original instance, the objective function contains an elastic net ($\beta L_1 + L_2$) regularizer. Optionally, an auto-encoder can be trained to reconstruct instances of the training set. We can then introduce the $L_2$ reconstruction error of the perturbed instance as an additional loss term in our objective function. As a result, the perturbed instance lies close to the training data manifold.

The ability to add or remove features to arrive at respectively PN’s or PP’s implies that there are feature values that contain no information with regards to the model’s predictions. Consider for instance the MNIST image below where the pixels are scaled between 0 and 1. The pixels with values close to 1 define the number in the image while the background pixels have value 0. We assume that perturbations towards the background value 0 are equivalent to removing features, while perturbations towards 1 imply adding features.
It is intuitive to understand that adding features to get a PN means changing 0’s into 1’s until a different number is formed, in this case changing a 4 into a 9.

To find the PP, we do the opposite and change 1’s from the original instance into 0’s, the background value, and only keep a vague outline of the original 4.
It is however often not trivial to find these non-informative feature values and domain knowledge becomes very important.

For more details, we refer the reader to the original paper.

## 8.2 Usage

### 8.2.1 Initialization

The optimizer is defined in TensorFlow (TF) internally. We first load our MNIST classifier and the (optional) auto-encoder. The example below uses Keras or TF models. This allows optimization of the objective function to run entirely with automatic differentiation because the TF graph has access to the underlying model architecture. For models built in different frameworks (e.g. scikit-learn), the gradients of part of the loss function with respect to the input features need to be evaluated numerically. We’ll handle this case later.

```python
# define models
cnn = load_model('mnist_cnn.h5')
ae = load_model('mnist_ae.h5')
```

We can now initialize the CEM explainer:

```python
# initialize CEM explainer
shape = (1,) + x_train.shape[1:]
mode = 'PN'
cem = CEM(cnn, mode, shape, kappa=0., beta=0.1,
          feature_range=(x_train.min(), x_train.max()),
          gamma=100, ae_model=ae, max_iterations=1000,
          c_init=1., c_steps=10, learning_rate_init=1e-2,
          clip=(-1000.,1000.), no_info_val=-1.)
```

Besides passing the the predictive and auto-encoder models, we set a number of hyperparameters... 

... general:

- **mode**: ‘PN’ or ‘PP’.
- **shape**: shape of the instance to be explained, starting with batch dimension. Currently only single explanations are supported, so the batch dimension should be equal to 1.
- **feature_range**: global or feature-wise min and max values for the perturbed instance.

... related to the optimizer:

- **max_iterations**: number of loss optimization steps for each value of \(c\); the multiplier of the first loss term.
- **learning_rate_init**: initial learning rate, follows polynomial decay.
- **clip**: min and max gradient values.

... related to the non-informative value:

- **no_info_val**: as explained in the previous section, it is important to define which feature values are considered background and not crucial for the class predictions. For MNIST images scaled between 0 and 1 or -0.5 and 0.5 as in the notebooks, pixel perturbations in the direction of the (low) background pixel value can be seen as removing features, moving towards the non-informative value. As a result, the \texttt{no_info_val} parameter is set at a low value like -1. \texttt{no_info_val} can be defined globally or feature-wise. For most applications, domain knowledge becomes very important here. If a representative sample of the training set is available, we can always (naively) infer a \texttt{no_info_val} by taking the feature-wise median or mean:
cem.fit(x_train, no_info_type='median')

... related to the objective function:

- **c_init** and **c_steps**: the multiplier $c$ of the first loss term is updated for $c_{steps}$ iterations, starting at $c_{init}$. The first loss term encourages the perturbed instance to be predicted as a different class for a PN and the same class for a PP. If we find a candidate PN or PP for the current value of $c$, we reduce the value of $c$ for the next optimization cycle to put more emphasis on the regularization terms and improve the solution. If we cannot find a solution, $c$ is increased to put more weight on the prediction class restrictions of the PN and PP before focusing on the regularization.

- **kappa**: the first term in the loss function is defined by a difference between the predicted probabilities for the perturbed instance of the original class and the max of the other classes. $\kappa \geq 0$ defines a cap for this difference, limiting its impact on the overall loss to be optimized. Similar to the original paper, we set $\kappa$ to 0. in the examples.

- **beta**: $\beta$ is the $L_1$ loss term multiplier. A higher value for $\beta$ means more weight on the sparsity restrictions of the perturbations. Similar to the paper, we set $\beta$ to 0.1 for the MNIST and Iris datasets.

- **gamma**: multiplier for the optional $L_2$ reconstruction error. A higher value for $\gamma$ means more emphasis on the reconstruction error penalty defined by the auto-encoder. Similar to the paper, we set $\gamma$ to 100 when we have an auto-encoder available.

While the paper’s default values for the loss term coefficients worked well for the simple examples provided in the notebooks, it is recommended to test their robustness for your own applications.

---

**Warning**

Once a CEM instance is initialized, the parameters of it are frozen even if creating a new instance. This is due to TensorFlow behaviour which holds on to some global state. In order to change parameters of the explainer in the same session (e.g. for explaining different models), you will need to reset the TensorFlow graph manually:

```python
import tensorflow as tf
tf.keras.backend.clear_session()
```

You may need to reload your model after this. Then you can create a new CEM instance with new parameters.

### 8.2.2 Explanation

We can finally explain the instance:

```python
explanation = cem.explain(X)
```

The `explain` method returns an Explanation object with the following attributes:

- **X**: original instance
- **X_pred**: predicted class of original instance
- **PN or PP**: Pertinent Negative or Pertinent Positive
- **PN_pred or PP_pred**: predicted class of PN or PP
- **grads_graph**: gradient values computed from the TF graph with respect to the input features at the PN or PP
- **grads_num**: numerical gradient values with respect to the input features at the PN or PP

---

Chapter 8. Contrastive Explanation Method
8.2.3 Numerical Gradients

So far, the whole optimization problem could be defined within the internal TF graph, making autodiff possible. It is however possible that we do not have access to the model architecture and weights, and are only provided with a `predict` function returning probabilities for each class. We initialize the CEM in the same way as before:

```python
# define model
lr = load_model('iris_lr.h5')
predict_fn = lambda x: lr.predict(x)

# initialize CEM explainer
shape = (1,) + x_train.shape[1:]
mode = 'PP'
cem = CEM(predict_fn, mode, shape, kappa=0., beta=.1,
          feature_range=(x_train.min(), x_train.max()),
          eps=(1e-2, 1e-2), update_num_grad=100)
```

In this case, we need to evaluate the gradients of the loss function with respect to the input features numerically:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial x}$$

where $L$ is the loss function, $p$ the predict function and $x$ the input features to optimize. There are now 2 additional hyperparameters to consider:

- **eps**: a tuple to define the perturbation size used to compute the numerical gradients. `eps[0]` and `eps[1]` are used respectively for $\frac{\partial L}{\partial p}$ and $\frac{\partial p}{\partial x}$. `eps[0]` and `eps[1]` can be a combination of float values or numpy arrays. For `eps[0]`, the array dimension should be $(1 \times \text{nb of prediction categories})$ and for `eps[1]` it should be $(1 \times \text{nb of features})$. For the Iris dataset, `eps` could look as follows:

```python
eps0 = np.array([[1e-2, 1e-2, 1e-2]])  # 3 prediction categories, equivalent to 1e-2
eps1 = np.array([[1e-2, 1e-2, 1e-2, 1e-2]])  # 4 features, also equivalent to 1e-2
eps = (eps0, eps1)
```

- **update_num_grad**: for complex models with a high number of parameters and a high dimensional feature space (e.g. Inception on ImageNet), evaluating numerical gradients can be expensive as they involve prediction calls for each perturbed instance. The `update_num_grad` parameter allows you to set a batch size on which to evaluate the numerical gradients, reducing the number of prediction calls required.

8.3 Examples

*Contrastive Explanations Method (CEM) applied to MNIST*

*Contrastive Explanations Method (CEM) applied to Iris dataset*

[source]
COUNTERFACTUAL INSTANCES

9.1 Overview

A counterfactual explanation of an outcome or a situation $Y$ takes the form “If $X$ had not occurred, $Y$ would not have occurred” (Interpretable Machine Learning). In the context of a machine learning classifier $X$ would be an instance of interest and $Y$ would be the label predicted by the model. The task of finding a counterfactual explanation is then to find some $X'$ that is in some way related to the original instance $X$ but leading to a different prediction $Y'$. Reasoning in counterfactual terms is very natural for humans, e.g. asking what should have been done differently to achieve a different result. As a consequence counterfactual instances for machine learning predictions is a promising method for human-interpretable explanations.

The counterfactual method described here is the most basic way of defining the problem of finding such $X'$. Our algorithm loosely follows Wachter et al. (2017): Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR. For an extension to the basic method which provides ways of finding higher quality counterfactual instances $X'$ in a quicker time, please refer to Counterfactuals Guided by Prototypes.

We can reason that the most basic requirements for a counterfactual $X'$ are as follows:

- The predicted class of $X'$ is different from the predicted class of $X$
- The difference between $X$ and $X'$ should be human-interpretable.

While the first condition is straight-forward, the second condition does not immediately lend itself to a condition as we need to first define “interpretability” in a mathematical sense. For this method we restrict ourselves to a particular definition by asserting that $X'$ should be as close as possible to $X$ without violating the first condition. The main issue with this definition of “interpretability” is that the difference between $X'$ and $X$ required to change the model prediction might be so small as to be un-interpretable to the human eye in which case we need a more sophisticated approach.

That being said, we can now cast the search for $X'$ as a simple optimization problem with the following loss:

$$L = L_{\text{pred}} + \lambda L_{\text{dist}},$$

where the first loss term $L_{\text{pred}}$ guides the search towards points $X'$ which would change the model prediction and the second term $\lambda L_{\text{dist}}$ ensures that $X'$ is close to $X$. This form of loss has a single hyperparameter $\lambda$ weighing the contributions of the two competing terms.

The specific loss in our implementation is as follows:

$$L(X'|X) = (f_t(X') - p_t)^2 + \lambda L_1(X', X).$$

Here $t$ is the desired target class for $X'$ which can either be specified in advance or left up to the optimization algorithm to find, $p_t$ is the target probability of this class (typically $p_t = 1$), $f_t$ is the model prediction on class $t$ and $L_1$ is the distance between the proposed counterfactual instance $X'$ and the instance to be explained $X$. The use of the $L_1$
distance should ensure that the $X'$ is a sparse counterfactual - minimizing the number of features to be changed in order to change the prediction.

The optimal value of the hyperparameter $\lambda$ will vary from dataset to dataset and even within a dataset for each instance to be explained and the desired target class. As such it is difficult to set and we learn it as part of the optimization algorithm, i.e. we want to optimize

$$\min_{X'} \max_{\lambda} L(X'|X)$$

subject to

$$|f_t(X') - p_t| \leq \epsilon \text{ (counterfactual constraint)},$$

where $\epsilon$ is a tolerance parameter. In practice this is done in two steps, on the first pass we sweep a broad range of $\lambda$, e.g. $\lambda \in (10^{-1}, \ldots, 10^{-10})$ to find lower and upper bounds $\lambda_{lb}, \lambda_{ub}$ where counterfactuals exist. Then we use bisection to find the maximum $\lambda \in [\lambda_{lb}, \lambda_{ub}]$ such that the counterfactual constraint still holds. The result is a set of counterfactual instances $X'$ with varying distance from the test instance $X$.

### 9.2 Usage

#### 9.2.1 Initialization

The counterfactual (CF) explainer method works on fully black-box models, meaning they can work with arbitrary functions that take arrays and return arrays. However, if the user has access to a full TensorFlow (TF) or Keras model, this can be passed in as well to take advantage of the automatic differentiation in TF to speed up the search. This section describes the initialization for a TF/Keras model, for fully black-box models refer to numerical gradients.

First we load the TF/Keras model:

```python
model = load_model('my_model.h5')
```

Then we can initialize the counterfactual object:

```python
shape = (1,) + x_train.shape[1:]
cf = CounterFactual(model, shape, distance_fn='l1', target_proba=1.0,
                    target_class='other', max_iter=1000, early_stop=50, lam_init=1e-1,
                    max_lam_steps=10, tol=0.05, learning_rate_init=0.1,
                    feature_range=(-1e10, 1e10), eps=0.01, init='identity',
                    decay=True, write_dir=None, debug=False)
```

Besides passing the model, we set a number of hyperparameters …

… general:

- `shape`: shape of the instance to be explained, starting with batch dimension. Currently only single explanations are supported, so the batch dimension should be equal to 1.
- `feature_range`: global or feature-wise min and max values for the perturbed instance.
- `write_dir`: write directory for Tensorboard logging of the loss terms. It can be helpful when tuning the hyperparameters for your use case. It makes it easy to verify that e.g. not 1 loss term dominates the optimization, that the number of iterations is OK etc. You can access Tensorboard by running `tensorboard --logdir {write_dir}` in the terminal.
- `debug`: flag to enable/disable writing to Tensorboard.

… related to the optimizer:
• **max_iterations**: number of loss optimization steps for each value of $\lambda$; the multiplier of the distance loss term.

• **learning_rate_init**: initial learning rate, follows linear decay.

• **decay**: flag to disable learning rate decay if desired

• **early_stop**: early stopping criterion for the search. If no counterfactuals are found for this many steps or if this many counterfactuals are found in a row we change $\lambda$ accordingly and continue the search.

• **init**: how to initialize the search, currently only "identity" is supported meaning the search starts from the original instance.

... related to the **objective function**:

• **distance_fn**: distance function between the test instance $X$ and the proposed counterfactual $X'$, currently only "l1" is supported.

• **target_proba**: desired target probability for the returned counterfactual instance. Defaults to 1.0, but it could be useful to reduce it to allow a looser definition of a counterfactual instance.

• **tol**: the tolerance within the target_proba, this works in tandem with target_proba to specify a range of acceptable predicted probability values for the counterfactual.

• **target_class**: desired target class for the returned counterfactual instance. Can be either an integer denoting the specific class membership or the string other which will find a counterfactual instance whose predicted class is anything other than the class of the test instance.

• **lam_init**: initial value of the hyperparameter $\lambda$. This is set to a high value $\lambda = 1e^{-1}$ and annealed during the search to find good bounds for $\lambda$ and for most applications should be fine to leave as default.

• **max_lam_steps**: the number of steps (outer loops) to search for with a different value of $\lambda$.

While the default values for the loss term coefficients worked well for the simple examples provided in the notebooks, it is recommended to test their robustness for your own applications.

---

**Warning**

Once a `CounterFactual` instance is initialized, the parameters of it are frozen even if creating a new instance. This is due to TensorFlow behaviour which holds on to some global state. In order to change parameters of the explainer in the same session (e.g. for explaining different models), you will need to reset the TensorFlow graph manually:

```python
import tensorflow as tf
tf.keras.backend.clear_session()
```

You may need to reload your model after this. Then you can create a new `CounterFactual` instance with new parameters.

---

### 9.2.2 Fit

The method is purely unsupervised so no fit method is necessary.
9.2.3 Explanation

We can now explain the instance $X$:

$$\text{explanation} = \text{cf}.\text{explain}(X)$$

The explain method returns an Explanation object with the following attributes:

- $\text{cf}$: dictionary containing the counterfactual instance found with the smallest distance to the test instance, it has the following keys:
  - $X$: the counterfactual instance
  - $\text{distance}$: distance to the original instance
  - $\lambda$: value of $\lambda$ corresponding to the counterfactual
  - $\text{index}$: the step in the search procedure when the counterfactual was found
  - $\text{class}$: predicted class of the counterfactual
  - $\text{proba}$: predicted class probabilities of the counterfactual
  - $\text{loss}$: counterfactual loss
- $\text{orig\_class}$: predicted class of original instance
- $\text{orig\_proba}$: predicted class probabilities of the original instance
- $\text{all}$: dictionary of all instances encountered during the search that satisfy the counterfactual constraint but have higher distance to the original instance than the returned counterfactual. This is organized by levels of $\lambda$, i.e. explanation['all'][0] will be a list of dictionaries corresponding to instances satisfying the counterfactual condition found in the first iteration over $\lambda$ during bisection.

9.2.4 Numerical Gradients

So far, the whole optimization problem could be defined within the TF graph, making automatic differentiation possible. It is however possible that we do not have access to the model architecture and weights, and are only provided with a predict function returning probabilities for each class. The counterfactual can then be initialized in the same way as before, but using a prediction function:

```python
# define model
model = load_model('mnist_cnn.h5')
predict_fn = lambda x: cnn.predict(x)

# initialize explainer
shape = (1,) + x_train.shape[1:]
ct = CounterFactual(predict_fn, shape, distance_fn='l1', target_proba=1.0,
                   target_class='other', max_iter=1000, early_stop=50, lam_init=1e-1,
                   max_lam_steps=10, tol=0.05, learning_rate_init=0.1,
                   feature_range=(-1e10, 1e10), eps=0.01, init
```

In this case, we need to evaluate the gradients of the loss function with respect to the input features $X$ numerically:

$$\frac{\partial L_{\text{pred}}}{\partial X} = \frac{\partial L_{\text{pred}}}{\partial p} \frac{\partial p}{\partial X}$$

where $L_{\text{pred}}$ is the predict function loss term, $p$ the predict function and $x$ the input features to optimize. There is now an additional hyperparameter to consider:
• **eps**: a float or an array of floats to define the perturbation size used to compute the numerical gradients of $\frac{\delta p}{\delta X}$. If a single float, the same perturbation size is used for all features, if the array dimension is $(1 \times \text{nb of features})$, then a separate perturbation value can be used for each feature. For the Iris dataset, **eps** could look as follows:

```python
eps = np.array([1e-2, 1e-2, 1e-2])  # 4 features, also equivalent to eps=1e-2
```

### 9.3 Examples

*Counterfactual instances on MNIST*

[source]
CHAPTER
TEN

COUNTERFACTUALS GUIDED BY PROTOTYPES

10.1 Overview

This method is based on the Interpretable Counterfactual Explanations Guided by Prototypes paper which proposes a fast, model agnostic method to find interpretable counterfactual explanations for classifier predictions by using class prototypes.

Humans often think about how they can alter the outcome of a situation. What do I need to change for the bank to approve my loan? is a common example. This form of counterfactual reasoning comes natural to us and explains how to arrive at a desired outcome in an interpretable manner. Moreover, examples of counterfactual instances resulting in a different outcome can give powerful insights of what is important to the the underlying decision process. This makes it a compelling method to explain predictions of machine learning models. In the context of predictive models, a counterfactual instance describes the necessary change in input features of a test instance that alter the prediction to a predefined output (e.g. a prediction class). The counterfactual is found by iteratively perturbing the input features of the test instance during an optimization process until the desired output is achieved.

A high quality counterfactual instance $x_{cf}$ should have the following desirable properties:

- The model prediction on $x_{cf}$ needs to be close to the predefined output.
- The perturbation $\delta$ changing the original instance $x_0$ into $x_{cf} = x_0 + \delta$ should be sparse.
- The counterfactual $x_{cf}$ should be interpretable. This implies that $x_{cf}$ needs to lie close to both the overall and counterfactual class specific data distribution.
- The counterfactual $x_{cf}$ needs to be found fast enough so it can be used in a real life setting.

We can obtain those properties by incorporating additional loss terms in the objective function that is optimized using gradient descent. A basic loss function for a counterfactual can look like this:

$$Loss = cL_{pred} + \beta L_1 + L_2$$

The first loss term, $cL_{pred}$, encourages the perturbed instance to predict another class than the original instance. The $\beta L_1 + L_2$ terms act as an elastic net regularizer and introduce sparsity by penalizing the size of the difference between the counterfactual and the perturbed instance. While we can obtain sparse counterfactuals using this objective function, these are often not very interpretable because the training data distribution is not taken into account, and the perturbations are not necessarily meaningful.

The Contrastive Explanation Method (CEM) uses an autoencoder which is trained to reconstruct instances of the training set. We can then add the $L_2$ reconstruction error of the perturbed instance as a loss term to keep the counterfactual close to the training data distribution. The loss function becomes:

$$Loss = cL_{pred} + \beta L_1 + L_2 + L_{AE}$$

The $L_{AE}$ does however not necessarily lead to interpretable solutions or speed up the counterfactual search. The lack of interpretability occurs because the overall data distribution is followed, but not the class specific one. That’s where the prototype loss term $L_{proto}$ comes in. To define the prototype for each prediction class, we can use the encoder
part of the previously mentioned autoencoder. We also need the training data or at least a representative sample. We use the model to make predictions on this data set. For each predicted class, we encode the instances belonging to that class. The class prototype is simply the average of the k closest encodings in that class to the encoding of the instance that we want to explain. When we want to generate a counterfactual, we first find the nearest prototype other than the one for the predicted class on the original instance. The $L_{proto}$ loss term tries to minimize the $L_2$ distance between the counterfactual and the nearest prototype. As a result, the perturbations are guided to the closest prototype, speeding up the counterfactual search and making the perturbations more meaningful as they move towards a typical in-distribution instance. If we do not have a trained encoder available, we can build class representations using k-d trees for each class. The prototype is then the k nearest instance from a k-d tree other than the tree which represents the predicted class on the original instance. The loss function now looks as follows:

$$Loss = cL_{pred} + \beta L_1 + L_2 + L_{AE} + L_{proto}$$

The method allows us to select specific prototype classes to guide the counterfactual to. For example, in MNIST the closest prototype to a 9 could be a 4. However, we can specify that we want to move towards the 7 prototype and avoid 4.

In order to help interpretability, we can also add a trust score constraint on the proposed counterfactual. The trust score is defined as the ratio of the distance between the encoded counterfactual and the prototype of the class predicted on the original instance, and the distance between the encoded counterfactual and the prototype of the class predicted for the counterfactual instance. Intuitively, a high trust score implies that the counterfactual is far from the originally predicted class compared to the counterfactual class. For more info on trust scores, please check out the documentation.

Because of the $L_{proto}$ term, we can actually remove the prediction loss term and still obtain an interpretable counterfactual. This is especially relevant for fully black box models. When we provide the counterfactual search method with a Keras or TensorFlow model, it is incorporated in the TensorFlow graph and evaluated using automatic differentiation. However, if we only have access to the model’s prediction function, the gradient updates are numerical and typically require a large number of prediction calls because of $L_{pred}$. These prediction calls can slow the search down significantly and become a bottleneck. We can represent the gradient of the loss term as follows:

$$\frac{\partial L_{pred}}{\partial x} = \frac{\partial L_{pred}}{\partial p} \frac{\partial p}{\partial x}$$

where $p$ is the prediction function and $x$ the input features to optimize. For a 28 by 28 MNIST image, the $\delta p / \delta x$ term alone would require a prediction call with batch size $28 \times 28 \times 2 = 1568$. By using the prototypes to guide the search however, we can remove the prediction loss term and only make a single prediction at the end of each gradient update to check whether the predicted class on the proposed counterfactual is different from the original class.

### 10.2 Categorical Variables

It is crucial for many machine learning applications to deal with both continuous numerical and categorical data. Explanation methods which rely on perturbations or sampling of the input features need to make sure those perturbations are meaningful and capture the underlying structure of the data. If not done properly, the perturbed or sampled instances are possibly out of distribution compared to the training data and result in misleading explanations. The perturbation or sampling process becomes tricky for categorical features. For instance random perturbations across possible categories or enforcing a ranking between categories based on frequency of occurrence in the training data do not capture this structure.

Our method first computes the pairwise distances between categories of a categorical variable based on either the model predictions (MVDM) or the context provided by the other variables in the dataset (ABDM). For MVDM, we use the difference between the conditional model prediction probabilities of each category. This method is based on the Modified Value Difference Metric (MVDM) by Cost et al (1993). ABDM stands for Association-Based Distance Metric, a categorical distance measure introduced by Le et al (2005). ABDM infers context from the presence of other variables in the data and computes a dissimilarity measure based on the Kullback-Leibler divergence. Both methods can also be combined as ABDM-MVDM. We can then apply multidimensional scaling to project the pairwise distances into Euclidean space. More details will be provided in a forthcoming paper.
The different use cases are highlighted in the example notebooks linked at the bottom of the page.

## 10.3 Usage

### 10.3.1 Initialization

The counterfactuals guided by prototypes method works on fully black-box models. This means that they can work with arbitrary functions that take arrays and return arrays. However, if the user has access to a full TensorFlow (TF) or Keras model, this can be passed in as well to take advantage of the automatic differentiation in TF to speed up the search. This section describes the initialization for a TF/Keras model. Please see the numerical gradients section for black box models.

We first load our MNIST classifier and the (optional) autoencoder and encoder:

```python
import tensorflow as tf

# load models
load_model = tf.keras.models.load_model

cnn = load_model('mnist_cnn.h5')
ae = load_model('mnist_ae.h5')
enc = load_model('mnist_enc.h5')
```

We can now initialize the counterfactual:

```python
shape = (1,) + x_train.shape[1:]
cf = CounterFactualProto(cnn, shape, kappa=0., beta=.1, gamma=100., theta=100.,
ae_model=ae, enc_model=enc, max_iterations=500,
feature_range=(-.5, .5), c_init=1., c_steps=5,
learning_rate_init=1e-2, clip=(-1000., 1000.), write_dir='./cf')
```

Besides passing the predictive, and (optional) autoencoder and models, we set a number of hyperparameters...

... general:

- **shape**: shape of the instance to be explained, starting with batch dimension. Currently only single explanations are supported, so the batch dimension should be equal to 1.
- **feature_range**: global or feature-wise min and max values for the perturbed instance.
- **write_dir**: write directory for Tensorboard logging of the loss terms. It can be helpful when tuning the hyperparameters for your use case. It makes it easy to verify that e.g. not 1 loss term dominates the optimization, that the number of iterations is OK etc. You can access Tensorboard by running `tensorboard --logdir {write_dir}` in the terminal. The figure below for example shows the loss to be optimized over different \(c\) iterations. It is clear that within each iteration, the number of `max_iterations` steps is too high and we can speed up the search.

![Loss Optimization](image)

... related to the optimizer:

- **max_iterations**: number of loss optimization steps for each value of \(c\); the multiplier of the first loss term.
- **learning_rate_init**: initial learning rate, follows polynomial decay.
• **clip**: min and max gradient values.

... related to the **objective function**:

• **c_init** and **c_steps**: the multiplier $c$ of the first loss term is updated for **c_steps** iterations, starting at **c_init**. The first loss term encourages the perturbed instance to be predicted as a different class than the original instance. If we find a candidate counterfactual for the current value of $c$, we reduce the value of $c$ for the next optimization cycle to put more emphasis on the other loss terms and improve the solution. If we cannot find a solution, $c$ is increased to put more weight on the prediction class restrictions of the counterfactual.

• **kappa**: the first term in the loss function is defined by a difference between the predicted probabilities for the perturbed instance of the original class and the max of the other classes. $\kappa \geq 0$ defines a cap for this difference, limiting its impact on the overall loss to be optimized. Similar to CEM, we set $\kappa$ to 0 in the examples.

• **beta**: $\beta$ is the $L_1$ loss term multiplier. A higher value for $\beta$ means more weight on the sparsity restrictions of the perturbations. $\beta$ equal to 0.1 works well for the example datasets.

• **gamma**: multiplier for the optional $L_2$ reconstruction error. A higher value for $\gamma$ means more emphasis on the reconstruction error penalty defined by the autoencoder. A value of 100 is reasonable for the examples.

• **theta**: multiplier for the $L_{proto}$ loss term. A higher $\theta$ means more emphasis on the gradients guiding the counterfactual towards the nearest class prototype. A value of 100 worked well for the examples.

When the dataset contains categorical variables, we need to additionally pass the following arguments:

• **cat_vars**: if the categorical variables have ordinal encodings, this is a dictionary with as keys the categorical columns and values the number of categories for the categorical variable in the dataset. If one-hot encoding is applied to the data, then the keys of the **cat_vars** dictionary represent the column where each categorical variable starts while the values still return the number of categories.

• **ohe**: a flag (True or False) whether the categories are one-hot encoded.

It is also important to remember that the perturbations are applied in the numerical feature space, after the categorical variables have been transformed into numerical features. This has to be reflected by the dimension and values of **feature_range**. Imagine for example that we have a dataset with 10 columns. Two of the features are categorical and one-hot encoded. They can both take 3 values each. As a result, the number of columns in the dataset is reduced to 6 when we transform those categorical features to numerical features. As a result, the **feature_range** needs to contain the upper and lower ranges for 6 features.

While the default values for the loss term coefficients worked well for the simple examples provided in the notebooks, it is recommended to test their robustness for your own applications.

---

**Warning**

Once a **CFProto** instance is initialized, the parameters of it are frozen even if creating a new instance. This is due to TensorFlow behaviour which holds on to some global state. In order to change parameters of the explainer in the same session (e.g. for explaining different models), you will need to reset the TensorFlow graph manually:

```python
import tensorflow as tf
tf.keras.backend.clear_session()
```

You may need to reload your model after this. Then you can create a new **CFProto** instance with new parameters.
10.3.2 Fit

If we use an encoder to find the class prototypes, we need an additional fit step on the training data:

```python
cf.fit(x_train)
```

We also need the fit step if the data contains categorical features so we can compute the numerical transformations. In practice, most of these optional arguments do not need to be changed.

```python
cf.fit(x_train, d_type='abdm', w=None, disc_perc=[25, 50, 75], standardize_cat_vars=False, smooth=1., center=True, update_feature_range=True)
```

- `d_type`: the distance metric used to compute the pairwise distances between the categories of each categorical variable. As discussed in the introduction, the options are "abdm", "mvdm" or "abdm-mvdm".
- `w`: if the combined metric "abdm-mvdm" is used, `w` is the weight (between 0 and 1) given to abdm.
- `disc_perc`: for abdm, we infer context from the other features. If there are continuous numerical features present, these are binned according to the quartiles in `disc_perc` before computing the similarity metric.
- `standardize_cat_vars`: whether to return the standardized values for the numerical distances of each categorical feature.
- `smooth`: if the difference in the distances between the categorical variables is too large, then a lower value of the `smooth` argument (0, 1) can smoothen out this difference. This would only be relevant if one categorical variable has significantly larger differences between its categories than others. As a result, the counterfactual search process will likely leave that categorical variable unchanged.
- `center`: whether to center the numerical distances of the categorical variables between the min and max feature ranges.
- `update_feature_range`: whether to update the `feature_range` parameter for the categorical variables based on the min and max values it computed in the fit step.

10.3.3 Explanation

We can now explain the instance:

```python
explanation = cf.explain(X, Y=None, target_class=None, k=20, k_type='mean',
threshold=0., verbose=True, print_every=100, log_every=100)
```

- `X`: original instance
- `Y`: one-hot-encoding of class label for `X`, inferred from the prediction on `X` if `None`.
- `target_class`: classes considered for the nearest class prototype. Either a list with class indices or `None`.
- `k`: number of nearest instances used to define the prototype for a class. Defaults to using all instances belonging to the class.
- `k_type`: use either the average encoding of the `k` nearest instances in a class as the class prototype (`k_type='mean'`) or the `k`-nearest encoding in the class (`k_type='point'`). This parameter is only relevant if an encoder is used to define the prototypes.
- `threshold`: threshold level for the ratio between the distance of the counterfactual to the prototype of the predicted class for the original instance over the distance to the prototype of the predicted class for the counterfactual. If the trust score is below the threshold, the proposed counterfactual does not meet the requirements and is rejected.
alibi Documentation, Release 0.5.6

- **verbose**: if True, print progress of counterfactual search every print_every steps.
- **log_every**: if write_dir for Tensorboard is specified, then log losses every log_every steps.

The `explain` method returns an Explanation object with the following attributes:

- **cf**: a dictionary with the overall best counterfactual found. `explanation['cf']` has the following key: value pairs:
  - `X`: the counterfactual instance
  - `class`: predicted class for the counterfactual
  - `proba`: predicted class probabilities for the counterfactual
  - `grads_graph`: gradient values computed from the TF graph with respect to the input features at the counterfactual
  - `grads_num`: numerical gradient values with respect to the input features at the counterfactual

- **orig_class**: predicted class for original instance

- **orig_proba**: predicted class probabilities for original instance

- **all**: a dictionary with the iterations as keys and for each iteration a list with counterfactuals found in that iteration as values. So for instance, during the first iteration, `explanation['all'][0]`, initially we typically find fairly noisy counterfactuals that improve over the course of the iteration. The counterfactuals for the subsequent iterations then need to be better (sparser) than the previous best counterfactual. So over the next few iterations, we probably find less but better solutions.

### 10.3.4 Numerical Gradients

So far, the whole optimization problem could be defined within the TF graph, making automatic differentiation possible. It is however possible that we do not have access to the model architecture and weights, and are only provided with a `predict` function returning probabilities for each class. The counterfactual can then be initialized in the same way:

```python
# define model
cnn = load_model('mnist_cnn.h5')
predict_fn = lambda x: cnn.predict(x)
ae = load_model('mnist_ae.h5')
enc = load_model('mnist_enc.h5')

# initialize explainer
shape = (1,) + x_train.shape[1:]
cf = CounterFactualProto(predict_fn, shape, gamma=100., theta=100.,
                        ae_model=ae, enc_model=enc, max_iterations=500,
                        feature_range=(-.5, .5), c_init=1., c_steps=4,
                        eps=(1e-2, 1e-2), update_num_grad=100)
```

In this case, we need to evaluate the gradients of the loss function with respect to the input features numerically:

\[
\frac{\partial L_{pred}}{\partial x} = \frac{\partial L_{pred}}{\partial p} \frac{\partial p}{\partial x}
\]

where \( L_{pred} \) is the loss term related to the prediction function, \( p \) is the prediction function and \( x \) are the input features to optimize. There are now 2 additional hyperparameters to consider:

- **eps**: a tuple to define the perturbation size used to compute the numerical gradients. `eps[0]` and `eps[1]` are used respectively for \( \delta L_{pred}/\delta p \) and \( \delta p/\delta x \). `eps[0]` and `eps[1]` can be a combination of float values or numpy arrays. For `eps[0]`, the array dimension should be \( (1 \times \text{nb of prediction categories}) \) and for `eps[1]` it should be \( (1 \times \text{nb of features}) \). For the Iris dataset, `eps` could look as follows:
eps0 = np.array([[1e-2, 1e-2, 1e-2]])  # 3 prediction categories, equivalent to 1e-2
eps1 = np.array([[1e-2, 1e-2, 1e-2, 1e-2]])  # 4 features, also equivalent to 1e-2
eps = (eps0, eps1)

- **update_num_grad**: for complex models with a high number of parameters and a high dimensional feature space (e.g. Inception on ImageNet), evaluating numerical gradients can be expensive as they involve prediction calls for each perturbed instance. The `update_num_grad` parameter allows you to set a batch size on which to evaluate the numerical gradients, reducing the number of prediction calls required.

We can also remove the prediction loss term by setting `c_init` to 0 and only run 1 `c_steps`, and still obtain an interpretable counterfactual. This dramatically speeds up the counterfactual search (e.g. by 100x in the MNIST example notebook):

```python
cf = CounterFactualProto(predict_fn, shape, gamma=100., theta=100.,
                        ae_model=ae, enc_model=enc, max_iterations=500,
                        feature_range=(-.5, .5), c_init=0., c_steps=1)
```

### 10.3.5 k-d trees

So far, we assumed that we have a trained encoder available to find the nearest class prototype. This is however not a hard requirement. As mentioned in the **Overview** section, we can use k-d trees to build class representations, find prototypes by querying the trees for each class and return the k nearest class instance as the closest prototype. We can run the counterfactual as follows:

```python
cf = CounterFactualProto(cnn, shape, use_kdtree=True, theta=10., feature_range=(-.5, .5))
cf.fit(x_train, trustscore_kwargs=None)
explanation = cf.explain(X, k=2)
```

- **trustscore_kwargs**: keyword arguments for the trust score object used to define the k-d trees for each class. Please check the trust scores documentation for more info.

### 10.4 Examples

*Counterfactuals guided by prototypes on MNIST*

*Counterfactuals guided by prototypes on Boston housing dataset*

*Counterfactual explanations with one-hot encoded categorical variables*

*Counterfactual explanations with ordinally encoded categorical variables*
11.1 Overview

Integrated gradients is a method originally proposed in Sundararajan et al., “Axiomatic Attribution for Deep Networks” that aims to attribute an importance value to each input feature of a machine learning model based on the gradients of the model output with respect to the input. In particular, integrated gradients defines an attribution value for each feature by considering the integral of the gradients taken along a straight path from a baseline instance $x'$ to the input instance $x$.

11.2 Integrated gradients method

The method is applicable to regression and classification models. In the case of a non-scalar output, such as in classification models or multi-target regression, the gradients are calculated for one given element of the output. For classification models, the gradient usually refers to the output corresponding to the true class or to the class predicted by the model.

Let us consider an input instance $x$, a baseline instance $x'$ and a model $M : X \to Y$ which acts on the feature space $X$ and produces an output $y$ in the output space $Y$. Let us now define the function $F$ as

- $F(x) = M(x)$ if the model output is a scalar;
- $F(x) = M_k(x)$ if the model output is a vector, with the index $k$ denoting the $k$-th element of $M(x)$.

For example, in case of a $K$-class classification, $M_k(x)$ is the probability of class $k$, which could be the true class corresponding to $x$ or the highest probability class predicted by the model. The attributions $A_i(x, x')$ for each feature $x_i$ with respect to the corresponding feature $x'_i$ in the baseline are calculated as

$$A_i(x, x') = (x_i - x'_i) \int_0^1 \frac{\partial F(x' + \alpha(x - x'))}{\partial x_i} d\alpha,$$

where the integral is taken along a straight path from the baseline $x'$ to the instance $x$ parameterized by the parameter $\alpha$.

It is shown that such attributions satisfy the following axioms:

- Sensitivity axiom: if we consider a baseline $x'$ which differs from the input instance $x$ for the value of one feature $x_i$ and yields different predictions, the attribution given to feature $x_i$ must be non-zero.
- Implementation invariance axiom: an attribution method should be such that the attributions do not depend on the particular implementation of the model.
• Completeness axiom: The completeness axiom states that the sum over all features attributions should be equal to the difference between the model output at the instance $x$ and the model output at the baseline $x'$:

$$\sum_i A_i(x, x') = F(x) - F(x').$$

The proofs that integrated gradients satisfies these axioms are relatively straightforward and are discussed in Sections 2 and 3 of the original paper “Axiomatic Attribution for Deep Networks”.

### 11.3 Usage

The alibi implementation of the integrated gradients method is specific to TensorFlow and Keras models.

```python
import tensorflow as tf
from alibi.explainers import IntegratedGradients

model = tf.keras.models.load_model("path_to_your_model")

ig = IntegratedGradients(model,
                          layer=None,
                          method="gausslegendre",
                          n_steps=50,
                          internal_batch_size=100)

explanation = ig.explain(X,
                         baselines=None,
                         target=None)

attributions = explanation.attributions
```

- **model**: Tensorflow or Keras model.
- **layer**: Layer with respect to which the gradients are calculated. If not provided, the gradients are calculated with respect to the input.
- **method**: Method for the integral approximation. Methods available: riemann_left, riemann_right, riemann_middle, riemann_trapezoid, gausslegendre.
- **n_steps**: Number of step in the path integral approximation from the baseline to the input instance.
- **internal_batch_size**: Batch size for the internal batching.

```python
explanation = ig.explain(X,
                         baselines=None,
                         target=None)

attributions = explanation.attributions
```

- **X**: Instances for which integrated gradients attributions are computed.
- **baselines**: Baselines (starting point of the path integral) for each instance. If the passed value is an np.ndarray must have the same shape as X. If not provided, all features values for the baselines are set to 0.
- **target**: Defines which element of the model output is considered to compute the gradients. It can be a list of integers or a numeric value. If a numeric value is passed, the gradients are calculated for the same element of the output for all data points. It must be provided if the model output dimension is higher than 1. For regression models whose output is a scalar, target should not be provided. For classification models target can be either the true classes or the classes predicted by the model.
11.3.1 Layer attributions

It is possible to calculate the integrated gradients attributions for the model input features or for the elements of an intermediate layer of the model. Specifically,

- If the parameter `layer` is set to its default value `None` as in the example above, the attributions are calculated for each input feature.
- If a layer of the model is passed, the attributions are calculated for each element of the layer passed.

Calculating attribution with respect to an internal layer of the model is particularly useful for models that take text as an input and use word-to-vector embeddings. In this case, the integrated gradients are calculated with respect to the embedding layer (see example on the IMDB dataset).

11.3.2 Baselines

Conceptually, baselines represent data points which do not contain information useful for the model task, and they are used as a benchmark by the integrated gradients method. Common choices for the baselines are data points with all features values set to zero (for example the black image in case of image classification) or set to a random value.

However, the choice of the baselines can have a significant impact on the values of the attributions. For example, if we consider a simple binary image classification task where a model is trained to predict whether a picture was taken at night or during the day, considering the black image as a baseline would be misleading: in fact, with such a baseline all the dark pixels of the images would have zero attributions, while they are likely to be important for the task at hand.

An extensive discussion about the impact of the baselines on integrated gradients attributions can be found in P. Sturmfels at al., “Visualizing the Impact of Feature Attribution Baselines”.

11.3.3 Targets

In the context of integrated gradients, the target variable specifies which element of the output should be considered to calculate the attributions. If the output of the model is a scalar, as in the case of single target regression, a target is not necessary, and the gradients are calculated in a straightforward way.

If the output of the model is a vector, the target value specifies the position of the element in the output vector considered for the calculation of the attributions. In case of a classification model, the target can be either the true class or the class predicted by the model for a given input.

11.4 Examples

`MNIST dataset`
`Imagenet dataset`
`IMDB dataset text classification`
[source]
12.1 Overview

The Kernel SHAP (SHapley Additive exPlanations) algorithm is based on the paper A Unified Approach to Interpreting Model Predictions by Lundberg et al. and builds on the open source shap library from the paper’s first author.

The algorithm provides model-agnostic (black box), human interpretable explanations suitable for regression and classification models applied to tabular data. This method is a member of the additive feature attribution methods class; feature attribution refers to the fact that the change of an outcome to be explained (e.g., a class probability in a classification problem) with respect to a baseline (e.g., average prediction probability for that class in the training set) can be attributed in different proportions to the model input features.

A simple illustration of the explanation process is shown in Figure 1. Here we see depicted a model which takes as input features such as Age, BMI or Sex and outputs a continuous value. We know that the average value of that output in a dataset of interest is 0.1. Using the Kernel SHAP algorithm, we attribute the 0.3 difference to the input features. Because the sum of the attribute values equals output \(-\) base rate, this method is additive. We can see for example that the Sex feature contributes negatively to this prediction whereas the remainder of the features have a positive contribution. For explaining this particular data point, the Age feature seems to be the most important. See our examples on how to perform explanations with this algorithm and visualise the results using the shap library visualisations here, here and here.

![Figure 1: Cartoon illustration of black-box explanation models with Kernel SHAP](source)

Image Credit: Scott Lundberg (see source here)
12.2 Usage

In order to compute the shap values, the following hyperparameters can be set when calling the `explain` method:

- `nsamples`: Determines the number of subsets used for the estimation of the shap values. A default of $2M + 2^{11}$ is provided where $M$ is the number of features. One is encouraged to experiment with the number of samples in order to determine a value that balances explanation accuracy and runtime.

- `l1_reg`: can take values 0, `False` to disable, `auto` for automatic regularisation selection, `bic` or `aic` to use $\ell_1$ regularised regression with the Bayes/Akaike information criteria for regularisation parameter selection, `num_features(10)` to specify the number of feature effects to be returned or a float value that is used as the regularisation coefficient for the $\ell_1$ penalised regression. The default option `auto` uses the least angle regression algorithm with the Akaike Information Criterion if a fraction smaller than 0.2 of the total number of subsets is enumerated.

If the dataset to be explained contains categorical variables, then the following options can be specified unless the categorical variables have been grouped (see example below):

- `summarise_result`: if True, the shap values estimated for dimensions of an encoded categorical variable are summed and a single shap value is returned for the categorical variable. This requires that both arguments below are specified:

  - `cat_var_start_idx`: a list containing the column indices where categorical variables start. For example if the feature matrix contains a categorical feature starting at index 0 and one at index 10, then `cat_var_start_idx=[0, 10]`.

  - `cat_vars_enc_dim`: a list containing the dimension of the encoded categorical variables. The number of columns specified in this list is summed for each categorical variable starting with the corresponding index in `cat_var_start_idx`. So if `cat_var_start_idx=[0, 10]` and `cat_vars_enc_dim=[3, 5]`, then the columns with indices 0, 1 and 2 and 10, 11, 12, 13 and 14 will be combined to return one shap value for each categorical variable, as opposed to 3 and 5.

12.2.1 Explaining continuous datasets

Initialisation and fit

The explainer is initialised by specifying:

- a predict function.

- optionally, setting `link='logit'` if the the model to be explained is a classifier that outputs probabilities. This will apply the logit function to convert outputs to margin space.

- optionally, providing a list of `feature_names`

Hence assuming the classifier takes in 4 inputs and returns probabilities of 3 classes, we initialise its explainer as:

```python
from alibi.explainers import KernelShap
predict_fn = lambda x: clf.predict_proba(x)
explainer = KernelShap(predict_fn, link='logit', feature_names=['a', 'b', 'c', 'd'])
```

To fit our classifier, we simply pass our background or ‘reference’ dataset to the explainer:

```python
explainer.fit(X_reference)
```

Note that `X_reference` is expected to have a `samples x features` layout.
Explanation

To explain an instance \( X \), we simply pass it to the `explain` method:

```python
explanation = explainer.explain(X)
```

The returned explanation object has the following fields:

- **explanation.meta**:
  
  ```
  {'name': 'KernelShap',
   'type': ['blackbox'],
   'explanations': ['local', 'global'],
   'params': {
     'groups': None,
     'group_names': None,
     'weights': None,
     'summarise_background': False
   }
  
  This field contains metadata such as the explainer name and type as well as the type of explanations this method can generate. In this case, the `params` attribute shows that none of the `fit` method optional parameters have been set.

- **explanation.data**:
  
  ```
  {'shap_values': [array([ 0.8340445 , 0.12000589, -0.07984099, 0.61758141]),
   array([-0.71522546, 0.31749045, 0.3146705 , -0.13365639]),
   array([-0.12984616, -0.47194649, -0.23036243, -0.52314911])],
   'expected_value': array([0.74456904, 1.05058744, 1.15837362]),
   'link': 'logit',
   'feature_names': [ 'a', 'b', 'c', 'd' ],
   'categorical_names': {},
   'raw': {
     'raw_prediction': array([ 2.23635984, 0.83386654, -0.19693058]),
     'prediction': array([0]),
     'instances': array([ 0.93884707, -0.63216607, -0.4350103 , -0.91969562]),
     'importances': {
       '0': {'ranked_effect': array([0.8340445 , 0.61758141, 0.12000589, 0.]),
           'names': [ 'a', 'd', 'b', 'c' ]},
       '1': {'ranked_effect': array([0.71522546, 0.31749045, 0.3146705 , 0.]),
              'names': [ 'a', 'b', 'c', 'd' ]},
       '2': {'ranked_effect': array([0.52314911, 0.47194649, 0.23036243, 0.]),
              'names': [ 'd', 'b', 'c', 'a' ]},
       'aggregated': {'ranked_effect': array([1.67911611, 1.27438691, 0.90944283, 0.]),
                      'names': [ 'a', 'd', 'b', 'c' ]}
     }
   }
  
  This field contains:
  ```
  ```
  - **shap_values**: a list of length equal to the number of model outputs, where each entry is an array of dimension `samples x features` of shap values. For the example above, only one instance with 4 features has been explained so the shap values for each class are of dimension `1 x 4`.
  ```
  ```
  - **expected_value**: an array of the expected value for each model output across `X_reference`


- **link**: which function has been applied to the model output prior to computing the expected_value and estimation of the shap_values

- **feature_names**: a list with the feature names, if provided. Defaults to a list containing strings of with the format feature_{()} if no names are passed

- **categorical_names**: a mapping of the categorical variables (represented by indices in the shap_values columns) to the description of the category

- **raw**: this field contains:
  - **raw_prediction**: a samples x n_outputs array of predictions for each instance to be explained. Note that this is calculated by applying the link function specified in link to the output of pred_fn
  - **prediction**: a samples array containing the index of the maximum value in the raw_prediction array
  - **instances**: a samples x n_features array of instances which have been explained
  - **importances**: a dictionary where each entry is a dictionary containing the sorted average magnitude of the shap value (ranked_effect) along with a list of feature names corresponding to the re-ordered shap values (names). There are n_outputs + 1 keys, corresponding to n_outputs and to the aggregated output (obtained by summing all the arrays in shap_values)

Please see our examples on how to visualise these outputs using the shap library visualisations [here](#), [here](#) and [here](#).

### 12.2.2 Explaining heterogeneous (continuous and categorical) datasets

When the dataset contains both continuous and categorical variables, **categorical_names**, an optional mapping from the encoded categorical features to a description of the category can be passed in addition to the **feature_names** list. This mapping is currently used for determining what type of summarisation should be applied if X_reference is large and the fit argument summarise_background='auto' or summarise_background=True but in the future it might be used for annotating visualisations. The definition of the map depends on what method is used to handle the categorical variables.

#### By grouping categorical data

By grouping categorical data we estimate a single shap value for each categorical variable.

#### Initialisation and fit

Assume that we have a dataset with features such as Marital Status (first column), Age (2nd column), Income (3rd column) and Education (4th column). The 2nd and 3rd column are continuous variables, whereas the 1st and 4th are categorical ones.

The mapping of categorical variables could be generated from a Pandas dataframe using the utility `gen_category_map`, imported from alibi.utils.data. For this example the output could look like:

```python
category_map = {
    0: ["married", "divorced"],
    3: ["high school diploma", "master's degree"],
}
```

Hence, using the same predict function as before, we initialise the explainer as:
To group our data, we have to provide the `groups` list, which contains lists with indices that are grouped together. In our case this would be:

```python
groups = [[0, 1], [2], [3], [4, 5]]
```

Similarly, the `group_names` are the same as the feature names:

```python
group_names = "Marital Status", "Age", "Income", "Education"
```

Note that, in this case, the keys of the `category_map` are indices into `groups`. To fit our explainer we pass one-hot encoded data to the explainer along with the grouping information.

```python
explainer.fit(
    X_reference,
    group_names=group_names,
    groups=groups,
)
```

**Explanation**

To perform an explanation, we pass one hot encoded instances `X` to the `explain` method:

```python
explanation = explainer.explain(X)
```

The explanation returned will contain the grouping information in its `meta` attribute:

```json
{"name": "KernelShap",
'type': ['blackbox'],
'explanations': ["local", "global"],
'params': {'groups': [[0, 1], [2], [3], [4, 5]],
    'group_names': ["Marital Status", "Age", "Income", "Education"],
    'weights': None,
    'summarise_background': False}
}
```

whereas inspecting the `data` attribute shows that one shap value is estimated for each of the four groups:

```json
{"shap_values": [array([ 0.8340445, 0.12000589, -0.07984099, 0.61758141]),
    array([-0.71522546, 0.31749045, 0.3146705, -0.13365639]),
    array([-0.12984616, -0.47194649, -0.23036243, -0.52314911]),
    'expected_value': array([0.74456904, 1.05058744, 1.15837362]),
    'link': "logit",
    'feature_names': ["Marital Status", "Age", "Income", "Education"],
    'categorical_names': {},
    'raw': {"raw_prediction": array([2.23635984, 0.83386654, -0.19693058]),
    'prediction': array([0])},
```

(continues on next page)
By summing output

An alternative to grouping, with a higher runtime cost, is to estimate one shap value for each dimension of the one-hot encoded data and sum the shap values of the encoded dimensions to obtain only one shap value per categorical variable.

Initialisation and fit

The initialisation step is as before:

```python
explainer = KernelShap(
    predict_fn,
    link='logit',
    feature_names=['Marital Status', 'Age', 'Income', 'Education'],
    categorical_names=category_map,
)
```

However, note that the keys of the `category_map` have to correspond to the locations of the categorical variables after the effects for the encoded dimensions have been summed up (see details below).

The fit step requires one hot encoded data and simply takes the reference dataset:

```python
explainer.fit(X_reference)
```

Explanation

To obtain a single shap value per categorical result, we have to specify the following arguments to the `explain` method:

- **summarise_result**: indicates that some shap values will be summed
- **cat_vars_start_idx**: the column indices where the first encoded dimension is for each categorical variable
- **cat_vars_enc_dim**: the length of the encoding dimensions for each categorical variable
In our case, Marital Status starts at column 0 and occupies 2 columns, Age and Income occupy columns 2 and 3 and Education occupies columns 4 and 5.

**By combining preprocessor and predictor**

Finally, an alternative is to combine the preprocessor and the predictor together in the same object, and fit the explainer on data before preprocessing.

**Initialisation and fit**

To do so, we first redefine our predict function as

```python
predict_fn = lambda x: clf.predict(preprocessor.transform(x))
```

The explainer can be initialised as:

```python
explainer = KernelShap(
    predict_fn,
    link='logit',
    feature_names=['Marital Status', 'Age', 'Income', 'Education'],
    categorical_names=category_map,
)
```

Then, the explainer should be fitted on unprocessed data:

```python
explainer.fit(X_reference_unprocessed)
```

**Explanation**

We can explain unprocessed records simply by calling explain:

```python
explanation = explainer.explain(X_unprocessed)
```

**12.2.3 Running batches of explanations in parallel**

Increases in the size of the background dataset, the number of samples used to estimate the shap values or simply explaining a large number of instances dramatically increase the cost of running Kernel SHAP.

To explain batches of instances in parallel, first run `pip install alibi[ray]` to install required dependencies and then simply initialise `KernelShap` specifying the number of physical cores available as follows:

```python
distrib_kernel_shap = KernelShap(predict_fn, distributed_opts={'n_cpus': 10})
```

To explain, simply call the `explain` as before - no other changes are required.
12.2.4 Miscellaneous

Runtime considerations

For a given instance, the runtime of the algorithm depends on:

- the size of the reference dataset
- the dimensionality of the data
- the number of samples used to estimate the shap values

Adjusting the size of the reference dataset

The algorithm automatically warns the user if a background dataset size of more than 300 samples is passed. If the runtime of an explanation with the original dataset is too large, then the algorithm can automatically subsample the background dataset during the fit step. This can be achieved by specifying the fit step as

```python
explainer.fit(
    X_reference,
    summarise_background=True,
    n_background_samples=150,
)
```
or

```python
explainer.fit(
    X_reference,
    summarise_background='auto'
)
```

The auto option will select 300 examples, whereas using the boolean argument allows the user to directly control the size of the reference set. If categorical variables or grouping options are specified, the algorithm uses subsampling of the data. Otherwise, a kmeans clustering algorithm is used to select the background dataset and the samples are weighted according to the frequency of occurrence of the cluster they are assigned to, which is reflected in the expected_value attribute of the explainer.

As described above, the explanations are performed with respect to the expected (or weighted-average) output over this dataset so the shap values will be affected by the dataset selection. We recommend experimenting with various ways to choose the background dataset before deploying explanations.

The dimensionality of the data and the number of samples used in shap value estimation

The dimensionality of the data has a slight impact on the runtime, since by default the number of samples used for estimation is $2 \times n_{\text{features}} + 2 \times 11$. In our experiments, we found that either grouping the data or fitting the explainer on unprocessed data resulted in runtime savings (but did not run rigorous comparison experiments). If grouping/fitting on unprocessed data alone does not give enough runtime savings, the background dataset could be adjusted. Additionally (or alternatively), the number of samples could be reduced as follows:

```python
explanation = explainer.explain(X, nsamples=500)
```

We recommend experimenting with this setting to understand the variance in the shap values before deploying such configurations.
Imbalanced datasets

In some situations, the reference datasets might be imbalanced so one might wish to perform an explanation of the model behaviour around \( x \) with respect to \( \sum_{i} w_i f(y_i) \) as opposed to \( E_D[f(y)] \). This can be achieved by passing a list or an 1-D numpy array containing a weight for each data point in \( X_{\text{reference}} \) as the \texttt{weights} argument of the \texttt{fit} method.

12.3 Theoretical overview

Consider a model \( f \) that takes as an input \( M \) features. Assume that we want to explain the output of the model \( f \) when applied to an input \( x \). Since the model output scale does not have an origin (it is an affine space), one can only explain the difference of the observed model output with respect to a chosen origin point. This point can be taken to be the function output value for an arbitrary record or the average output over a set of records, \( D \). Assuming the latter case, for the explanation to be accurate, one requires

\[
 f(x) - E_{y \sim D}[f(y)] = \sum_{i=1}^{M} \phi_i
\]

where \( D \) is also known as a background dataset and \( \phi_i \) is the portion of the change attributed to the \( i \)th feature. This portion is sometimes referred to as feature importance, effect or simply shap value.

One can conceptually imagine the estimation process for the shap value of the \( i \)th feature \( x_i \) as consisting of the following steps:

- enumerate all subsets \( S \) of the set \( F = \{1, ..., M\} \setminus \{i\} \)
- for each \( S \subseteq F \setminus \{i\} \), compute the contribution of feature \( i \) as \( C(i|S) = f(S \cup \{i\}) - f(S) \)
- compute the shap value according to

\[
 \phi_i := \frac{1}{M} \sum_{S \subseteq F \setminus \{i\}} \frac{1}{M - |S|} C(i|S).
\]

The semantics of \( f(S) \) in the above is to compute \( f \) by treating \( \bar{S} \) as missing inputs. Thus, we can imagine the process of computing the SHAP explanation as starting with \( S \) that does not contain our feature, adding feature \( i \) and then observing the difference in the function value. For a nonlinear function the value obtained will depend on which features are already in \( S \), so we average the contribution over all possible ways to choose a subset of size \( |S| \) and over all subset sizes. The issue with this method is that:

- the summation contains \( 2^M \) terms, so the algorithm complexity is \( O(M2^M) \)
- since most models cannot accept an arbitrary pattern of missing inputs at inference time, calculating \( f(S) \) would involve model retraining the model an exponential number of times

To overcome this issue, the following approximations are made:

- the missing features are simulated by replacing them with values from the background dataset
- the feature attributions are estimated instead by solving

\[
 \min_{\phi_1, ..., \phi_M} \left\{ \sum_{S \subseteq F} \left[ f(S) - \sum_{j \in S} \phi_j \right]^2 \pi_x(S) \right\}
\]

where

\[
 \pi_x(S) = \frac{M - 1}{(M - |S|)(M - |S|)}
\]
is the Shapley kernel (Figure 2).

![Shapley Kernel Values as a function of missing feature subset size for set of M=15 features](image)

**Figure 2: Shapley kernel**

Note that the optimisation objective implies above an exponential number of terms. In practice, one considers a finite number of samples \( n \), selecting \( n \) subsets \( S_1, ..., S_n \) according to the probability distribution induced by the kernel weights. We can see that the kernel favours either small or large subset sizes, since most of the information about the effect of a particular feature for an outcome change can be obtained by excluding that feature or excluding all the features except for it from the input set.

Therefore, Kernel SHAP returns an approximation of the true Shapley values, whose variability depends on factors such as the size of the structure of the background dataset used to estimate the feature attributions and the number of subsets of missing features sampled. Whenever possible, algorithms specialised for specific model structures (e.g., Tree SHAP, Linear SHAP, integrated gradients) should be used since they are faster and more accurate.

### 12.3.1 Comparison to other methods

Like LIME, this method provides local explanations, in the sense that the attributions are estimated to explain the change from a baseline for a given data point, \( x \). LIME computes the feature attributions by optimising the following objective in order to obtain a locally accurate explanation model (i.e., one that approximates the model to explained well around an instance \( x \)):

\[
\zeta = \arg \min_{g \in \mathcal{G}} L(f, g, \pi_x) + \Omega(g).
\]

Here \( f \) is the model to be explained, \( g \) is the explanation model (assumed linear), \( \pi \) is a local kernel around instance \( x \) (usually cosine or \( \ell_2 \) kernel) and \( \Omega(g) \) penalises explanation model complexity. The choices for \( L, \pi \) and \( \Omega \) in
LIME are heuristic, which can lead to unintuitive behaviour (see Section 5 of Lundberg et al. for a study). Instead, by computing the shap values according to the weighted regression in the previous section, the feature attributions estimated by Kernel SHAP have desirable properties such as local accuracy, consistency and missingness, detailed in Section 3 of Lundberg et al..

Although, in general, local explanations are limited in that it is not clear to what a given explanation applies around and instance $x$ (see anchors algorithm overview here for a discussion), insights into global model behaviour can be drawn by aggregating the results from local explanations (see the work of Lundberg et al. here). In the future, a distributed version of the Kernel SHAP algorithm will be available in order to reduce the runtime requirements necessary for explaining large datasets.

12.4 Examples

12.4.1 Continuous Data

Introductory example: Kernel SHAP on Wine dataset
Comparison with interpretable models

12.4.2 Mixed Data

Handling categorical variables with Kernel SHAP: an income prediction application
Handling categorical variables with Kernel SHAP: fitting explainers on data before pre-processing
Distributed Kernel SHAP: parallelizing explanations on multiple cores
[source]
MEASURING THE LINEARITY OF MACHINE LEARNING MODELS

13.1 Overview

Machine learning models include in general linear and non-linear operations: neural networks may include several layers consisting of linear algebra operations followed by non-linear activation functions, while models based on decision trees are by nature highly non-linear. The linearity measure function and class provide an operational definition for the amount of non-linearity of a map acting on vector spaces. Roughly speaking, the amount of non-linearity of the map is defined based on how much the output of the map applied to a linear superposition of input vectors differs from the linear superposition of the map’s outputs for each individual vector. In the context of supervised learning, this definition is immediately applicable to machine learning models, which are fundamentally maps from a input vector space (the feature space) to an output vector space that may represent probabilities (for classification models) or actual values of quantities of interest (for regression models).

Given an input vector space \( V \), an output vector space \( W \) and a map \( M : V \rightarrow W \), the amount of non-linearity of the map \( M \) in a region \( \beta \) of the input space \( V \) and relative to some coefficients \( \alpha(v) \) is defined as

\[
L_{\beta,\alpha}^{(M)} = \left\| \int_{\beta} \alpha(v)M(v)dv - M\left(\int_{\beta} \alpha(v)v dv\right) \right\|,
\]

where \( v \in V \) and \( \| \cdot \| \) denotes the norm of a vector. If we consider a finite number of vectors \( N \), the amount of non-linearity can be defined as

\[
L_{\beta,\alpha}^{(M)} = \left\| \sum_{i} \alpha_i M(v_i) - M\left(\sum_{i} \alpha_i v_i\right) \right\|,
\]

where, with an abuse of notation, \( \beta \) is no longer a continuous region in the input space but a collection of input vectors \( \{v_i\} \) and \( \alpha \) is no longer a function but a collection of real coefficients \( \{\alpha_i\} \) with \( i \in \{1, ..., N\} \). Note that the second expression may be interpreted as an approximation of the integral quantity defined in the first expression, where the vectors \( \{v_i\} \) are sampled uniformly in the region \( \beta \).

13.2 Application to machine learning models

In supervised learning, a model can be considered as a function \( M \) mapping vectors from the input space (feature vectors) to vectors in the output space. The output space may represents probabilities in the case of a classification model or values of the target quantities in the case of a regression model. The definition of the linearity measure given above can be applied to the case of a regression model (either a single target regression or a multi target regression) in a straightforward way.

In case of a classifier, let us denote by \( z \) the logits vector of the model such that the probabilities of the model \( M \) are given by softmax(\( z \)). Since the activation function of the last layer is usually highly non-linear, it is convenient to apply the definition of linearity given above to the logits vector \( z \). In the “white box” scenario, in which we have
access to the internal architecture of the model, the vector \( z \) is accessible and the amount of non-linearity can be calculated immediately. On the other hand, if the only accessible quantities are the output probabilities (the “black box” scenario), we need to invert the last layer’s activation function in order to retrieve \( z \). In other words, that means defining a new map \( M' = f^{-1} \circ M(v) \) where \( f \) is the activation function at the last layer and considering \( L_{M'}^{(M')}^{(M')} \) as a measure of the non-linearity of the model. The activation function of the last layer is usually a sigmoid function for binary classification tasks or a softmax function for multi-class classification. The inversion of the sigmoid function does not present any particular challenge, and the map \( M' \) can be written as

\[
M' = - \log \left( \frac{1 - M(v)}{M(v)} \right).
\]

On the other hand, the softmax probabilities \( p \) are defined in terms of the vector \( z \) as \( p_j = \frac{e^{z_j}}{\sum_i e^{z_i}} \), where \( z_j \) are the components of \( z \). The inverse of the softmax function is thus defined up to a constant \( C \) which does not depend on \( j \) but might depend on the input vector \( v \). The inverse map \( M' = \text{softmax}^{-1} \circ M(v) \) is then given by:

\[
M' = \log \circ M(v) + C(v),
\]

where \( C(v) \) is an arbitrary constant depending in general on the input vector \( v \).

Since in the black box scenario it is not possible to assess the value of \( C \), henceforth we will ignore it and define the amount of non-linearity of a machine learning model whose output is a probability distribution as

\[
L_{(log \circ M)}^\beta,\alpha = \left\| \sum_i^n \alpha_i \log \circ M(v_i) - \log \circ M \left( \sum_i^n \alpha_i v_i \right) \right\|.
\]

It must be noted that the quantity above may in general be different from the “actual” amount of non-linearity of the model, i.e. the quantity calculated by accessing the activation vectors \( z \) directly.

## 13.3 Implementation

### 13.3.1 Sampling

The module implements two different methods for the sampling of vectors in a neighbourhood of the instance of interest \( v \).

- The first sampling method \texttt{grid} consists of defining the region \( \beta \) as a discrete lattice of a given size around the instance of interest, with the size defined in terms of the L1 distance in the lattice; the vectors are then sampled from the lattice according to a uniform distribution. The density and the size of the lattice are controlled by the resolution parameter \texttt{res} and the size parameter \texttt{epsilon}. This method is highly efficient and scalable from a computational point of view.

- The second sampling method \texttt{knn} consists of sampling from the same probability distribution the instance \( v \) was drawn from; this method is implemented by simply selecting the \( K \) nearest neighbours to \( v \) from a training set, when this is available. The \texttt{knn} method imposes the constraint that the neighbourhood of \( v \) must include only vectors from the training set, and as a consequence it will exclude out-of-distribution instances from the computation of linearity.
13.3.2 Pairwise vs global linearity

The module implements two different methods to associate a value of the linearity measure to \( v \).

- The first method consists of measuring the global linearity in a region around \( v \). This means that we sample \( N \) vectors \( \{v_i\} \) from a region \( \beta \) of the input space around \( v \) and apply

\[
L_{\beta, \alpha}^{(M)} = \left\| \sum_{i=1}^{N} \alpha_i M(v_i) - M\left(\sum_{i=1}^{N} \alpha_i v_i\right) \right\|,
\]

(13.1)

- The second method consists of measuring the pairwise linearity between the instance of interest and other vectors close to it, averaging over all such pairs. In other words, we sample \( N \) vectors \( \{v_i\} \) from \( \beta \) as in the global method, but in this case we calculate the amount of non-linearity \( L_{(v,v_i),\alpha}^{(v)} \) for every pair of vectors \( (v,v_i) \) and average over all the pairs. Given two coefficients \( \{\alpha_0, \alpha_1\} \) such that \( \alpha_0 + \alpha_1 = 1 \), we can define the pairwise linearity measure relative to the instance of interest \( v \) as

\[
L^{(M)} = \frac{1}{N} \sum_{i=0}^{N} \left\| \alpha_0 M(v) + \alpha_1 M(v_i) - M(\alpha_0 v + \alpha_1 v_i) \right\|.
\]

(13.2)

The two methods are slightly different from a conceptual point of view: the global linearity measure combines all \( N \) vectors sampled in \( \beta \) in a single superposition, and can be conceptually regarded as a direct approximation of the integral quantity. Thus, the quantity is strongly linked to the model behavior in the whole region \( \beta \). On the other hand, the pairwise linearity measure is an averaged quantity over pairs of superimposed vectors, with the instance of interest \( v \) included in each pair. For that reason, it is conceptually more tied to the instance \( v \) itself rather than the region \( \beta \) around it.

13.4 Usage

13.4.1 LinearityMeasure class

Given a model class with a predict method that return probabilities distribution in case of a classifier or numeric values in case of a regressor, the linearity measure \( L \) around an instance of interest \( X \) can be calculated using the class LinearityMeasure as follows:

```python
from alibi.confidence.model_linearity import LinearityMeasure

predict_fn = lambda x: model.predict(x)

lm = LinearityMeasure(method='grid',
                      epsilon=0.04,
                      nb_samples=10,
                      res=100,
                      alphas=None,
                      model_type='classifier',
                      agg='pairwise',
                      verbose=False)

lm.fit(X_train)
L = lm.score(predict_fn, X)
```

Where \( x_{\text{train}} \) is the dataset the model was trained on. The feature_range is inferred form \( x_{\text{train}} \) in the fit step.
13.4.2 linearity_measure function

Given a model class with a `predict` method that return probabilities distribution in case of a classifier or numeric values in case of a regressor, the linearity measure $L$ around an instance of interest $X$ can also be calculated using the `linearity_measure` function as follows:

```python
from alibi.confidence.model_linearity import linearity_measure, _infer_feature_range

predict_fn = lambda x: model.predict(x)

feature_range = _infer_feature_range(X_train)
L = linearity_measure(predict_fn,
                     X,
                     feature_range=feature_range,
                     method='grid',
                     X_train=None,
                     epsilon=0.04,
                     nb_samples=10,
                     res=100,
                     alphas=None,
                     agg='global',
                     model_type='classifier')
```

Note that in this case the `feature_range` must be explicitly passed to the function and it is inferred beforehand.

13.5 Examples

*Iris dataset*

*Fashion MNIST dataset*

[source]
CHAPTER
FOURTEEN

TRUST SCORES

14.1 Overview

It is important to know when a machine learning classifier’s predictions can be trusted. Relying on the classifier’s (uncalibrated) prediction probabilities is not optimal and can be improved upon. Enter trust scores. Trust scores measure the agreement between the classifier and a modified nearest neighbor classifier on the predicted instances. The trust score is the ratio between the distance of the instance to the nearest class different from the predicted class and the distance to the predicted class. A score of 1 would mean that the distance to the predicted class is the same as to the nearest other class. Higher scores correspond to more trustworthy predictions. The original paper on which the algorithm is based is called To Trust Or Not To Trust A Classifier. Our implementation borrows heavily from and extends the authors’ open source code.

The method requires labeled training data to build $k$-d trees for each prediction class. When the classifier makes predictions on a test instance, we measure the distance of the instance to each of the trees. The trust score is then calculated by taking the ratio of the smallest distance to any other class than the predicted class and the distance to the predicted class. The distance is measured to the $k$th nearest neighbor in each tree or by using the average distance from the first to the $k$th neighbor.

In order to filter out the impact of outliers in the training data, they can optionally be removed using 2 filtering techniques. The first technique builds a $k$-d tree for each class and removes a fraction $\alpha$ of the training instances with the largest $k$ nearest neighbor (kNN) distance to the other instances in the class. The second fits a kNN-classifier to the training set, and removes a fraction $\alpha$ of the training instances with the highest prediction class disagreement. Be aware that the first method operates on the prediction class level while the second method runs on the whole training set. It is also important to keep in mind that kNN methods might not be suitable when there are significant scale differences between the input features.

Trust scores can for instance be used as a warning flag for machine learning predictions. If the score drops below a certain value and there is disagreement between the model probabilities and the trust score, the prediction can be explained using techniques like anchors or contrastive explanations.

Trust scores work best for low to medium dimensional feature spaces. When working with high dimensional observations like images, dimensionality reduction methods (e.g. auto-encoders or PCA) could be applied as a pre-processing step before computing the scores. This is demonstrated by the following example notebook.
14.2 Usage

14.2.1 Initialization and fit

At initialization, the optional filtering method used to remove outliers during the fit stage needs to be specified as well:

```python
from alibi.confidence import TrustScore
ts = TrustScore(alpha=.05,
                filter_type='distance_knn',
                k_filter=10,
                leaf_size=40,
                metric='euclidean',
                dist_filter_type='point')
```

All the hyperparameters are optional:

- **alpha**: target fraction of instances to filter out.
- **filter_type**: filter method; one of None (no filtering), distance_knn (first technique discussed in Overview) or probability_knn (second technique).
- **k_filter**: number of neighbors used for the distance or probability based filtering method.
- **leaf_size**: affects the speed and memory usage to build the k-d trees. The memory scales with the ratio between the number of samples and the leaf size.
- **metric**: distance metric used for the k-d trees. Euclidean by default.
- **dist_filter_type**: point uses the distance to the $k$-nearest point while mean uses the average distance from the 1st to the $k$th nearest point during filtering.

In this example, we use the distance_knn method to filter out 5% of the instances of each class with the largest distance to its 10th nearest neighbor in that class:

```python
ts.fit(X_train, y_train, classes=3)
```

- **classes**: equals the number of prediction classes.

$X_{train}$ is the training set and $y_{train}$ represents the training labels, either using one-hot encoding (OHE) or simple class labels.

14.2.2 Scores

The trust scores are simply calculated through the `score` method. `score` also returns the class labels of the closest not predicted class as a numpy array:

```python
score, closest_class = ts.score(X_test,
                               y_pred,
                               k=2,
                               dist_type='point')
```

$y_{pred}$ can again be represented using both OHE or via class labels.

- **k**: $k$th nearest neighbor used to compute distance to for each class.
- **dist_type**: similar to the filtering step, we can compute the distance to each class either to the $k$-th nearest point (point) or by using the average distance from the 1st to the $k$th nearest point (mean).
14.3 Examples

*Trust Scores applied to Iris*

*Trust Scores applied to MNIST*

[source]
15.1 Overview

The tree SHAP (SHapley Additive exPlanations) algorithm is based on the paper From local explanations to global understanding with explainable AI for trees by Lundberg et al. and builds on the open source shap library from the paper’s first author.

The algorithm provides human interpretable explanations suitable for regression and classification of models with tree structure applied to tabular data. This method is a member of the additive feature attribution methods class; feature attribution refers to the fact that the change of an outcome to be explained (e.g., a class probability in a classification problem) with respect to a baseline (e.g., average prediction probability for that class in the training set) can be attributed in different proportions to the model input features.

A simple illustration of the explanation process is shown in Figure 1. Here we see depicted a tree-based model which takes as an input features such as Age, BMI or Blood pressure and outputs Mortality risk score, a continuous value. Let’s assume that we aim to explain the difference between and observed outcome and no risk, corresponding to a base value of 0.0. Using the Tree SHAP algorithm, we attribute the 4.0 difference to the input features. Because the sum of the attribute values equals output − base value, this method is additive.

We can see for example that the Sex feature contributes negatively to this prediction whereas the remainder of the features have a positive contribution (i.e., increase the mortality risk). For explaining this particular data point, the Blood Pressure feature seems to have the largest effect, and corresponds to an increase in the mortality risk. See our example on how to perform explanations with this algorithm and visualise the results using the shap library visualisations here and here.

Figure 1: Cartoon illustration of explanation models with Tree SHAP.

Image Credit: Scott Lundberg (see source here)
15.2 Usage

In order to compute the shap values, the following arguments can optionally be set when calling the `explain` method:

- `interactions`: set to `True` to decompose the shap value of every feature for every example into a main effect and interaction effects
- `approximate`: set to `True` to calculate an approximation to shap values (see our example)
- `check_additivity`: if the explainer is initialised with `model_output = raw` and this option is `True` the explainer checks that the sum of the shap values is equal to model output - expected value
- `tree_limit`: if an int is passed, an ensemble formed of only `tree_limit` trees is explained

If the dataset contains categorical variables that have been encoded before being passed to the explainer and a single shap value is desired for each categorical variable, the following options should be specified:

- `summarise_result`: set to `True`
- `cat_var_start_idx`: a sequence of integers containing the column indices where categorical variables start. If the feature matrix contains a categorical feature starting at index 0 and one at index 10, then `cat_var_start_idx=[0, 10]`
- `cat_vars_enc_dim`: a list containing the dimension of the encoded categorical variables. The number of columns specified in this list is summed for each categorical variable starting with the corresponding index in `cat_var_start_idx`. So if `cat_var_start_idx=[0, 10]` and `cat_vars_enc_dim=[3, 5]`, then the columns with indices 0, 1 and 2 and 10, 11, 12, 13 and 14 will be combined to return one shap value for each categorical variable, as opposed to 3 and 5.

15.2.1 Path-dependent feature perturbation algorithm

Initialisation and fit

The explainer is initialised with the following arguments:

- a model, which could be an `sklearn`, `xgboost`, `catboost` or `lightgbm` model. Note that some of the models in these packages or models trained with specific objectives may not be supported. In particular, passing raw strings as categorical levels for `catboost` and `lightgbm` is not supported
- `model_output` should always default to `raw` for this algorithm
- optionally, set `task` to `classification` or `regression` to indicate the type of prediction the model makes. If set to `regression` the `prediction` field of the response is empty
- optionally, a list of feature names via `feature_names`. This is used to provide information about feature importances in the response
- optionally, a dictionary, `category_names`, that maps the columns of the categorical variables to a list of strings representing the names of the categories. This may be used for visualisation in the future.

```python
from alibi.explainers import TreeShap

explainer = TreeShap(
    model,
    feature_names=['size', 'age'],
    categorical_names={0: ['S', 'M', 'L', 'XL', 'XXL']}
)
```

For this algorithm, `fit` is called with no arguments:
Explanation

To explain an instance \(X\), we simply pass it to the `explain` method:

```python
explanation = explainer.explain(X)
```

The returned explanation object has the following fields:

- **explanation.meta**:

  ```json
  {"name": "TreeShap", "type": ["whitebox"], "task": "classification", "explanations": ["local", "global"], "params": {"summarise_background": false, "algorithm": "tree_path_dependent", "kwargs": {}}}
  ```

  This field contains metadata such as the explainer name and type as well as the type of explanations this method can generate. In this case, the `params` attribute shows the Tree SHAP variant that will be used to explain the model in the `algorithm` attribute.

- **explanation.data**:

  ```json
  data={"shap_values": [array([ 5.0661433e-01, 2.7620478e-02], [-4.1725192e+00, 4.4859368e-03], [ 4.1338313e-01, -5.5618007e-02]),
               dtype=float32)
          ,
          "shap_interaction_values": [array([], dtype=float64)],
          "expected_value": array([-0.06472124]),
          "model_output": "raw",
          "categorical_names": {0: ['S', 'M', 'L', 'XL', 'XXL']},
          "feature_names": ['size', 'age'],
          "raw": {
            "raw_prediction": array([-0.73818872, -8.8434663 , -3.24204564]),
            "loss": [],
            "prediction": array([0, 0, 0]),
            "instances": array([[0, 23],
                                  [4, 55],
                                  [2, 43]]),
            "labels": array([], dtype=float64),
            "importances": {
              '0': {
                'ranked_effect': array([1.6975055 , 1.3598266], dtype=float32),
                'names': ['size', 'age'],
              },
              'aggregated': {
                'ranked_effect': array([1.6975055 , 1.3598266], dtype=float32),
                'names': ['size', 'age'],
              }
            }
          }
  ```

  (continues on next page)
This field contains:

- **shap_values**: a list of length equal to the number of model outputs, where each entry is an array of dimension samples x features of shap values. For the example above, 3 instances with 2 features has been explained so the shap values for each class are of dimension 3 x 2
- **shap_interaction_values**: an empty list since this interactions was set to False in the explain call
- **expected_value**: an array containing expected value for each model output
- **model_output**: raw indicates that the model raw output was explained, the only option for the path dependent algorithm
- **feature_names**: a list with the feature names
- **categorical_names**: a mapping of the categorical variables (represented by indices in the shap_values columns) to the description of the category
- **raw**: this field contains:
  - **raw_prediction**: a samples x n_outputs array of predictions for each instance to be explained.
  - **prediction**: an array containing the index of the maximum value in the raw_prediction array
  - **instances**: a samples x n_features array of instances which have been explained
  - **labels**: an array containing the labels for the instances to be explained
  - **importances**: a dictionary where each entry is a dictionary containing the sorted average magnitude of the shap value (ranked_effect) along with a list of feature names corresponding to the re-ordered shap values (names). There are n_outputs + 1 keys, corresponding to n_outputs and the aggregated output (obtained by summing all the arrays in shap_values)

Please see our examples on how to visualise these outputs using the shap library visualisations library visualisations [here](#) and [here](#).

### Shapley interaction values

#### Initialisation and fit

Shapley interaction values can only be calculated using the path-dependent feature perturbation algorithm in this release, so no arguments are passed to the fit method:

```python
explaner = TreeShap(
    model,
    model_output='raw',
)
explaner.fit()
```
Explanation

To obtain the Shapley interaction values, the `explain` method is called with the option `interactions=True`:

```python
explanation = explainer.explain(X, interactions=True)
```

The explanation contains a list with the shap interaction values for each model output in the `shap_interaction_values` field of the `data` property.

### 15.2.2 Interventional feature perturbation algorithm

#### Explaining model output

**Initialisation and fit**

```python
explainer = TreeShap(
    model,
    model_output='raw',
)
explainer.fit(X_reference)
```

Model output can be set to `model_output='probability'` to explain models which return probabilities. Note that this requires the model to be trained with specific objectives. Please the footnote to our path-dependent feature perturbation example for an example of how to set the model training objective in order to explain probability outputs.

**Explanation**

To explain instances in `X`, the explainer is called as follows:

```python
explanation = explainer.explain(X)
```

#### Explaining loss functions

**Initialisation and fit**

To explain loss function, the following configuration and fit steps are necessary:

```python
explainer = TreeShap(
    model,
    model_output='log_loss',
)
explainer.fit(X_reference)
```

Only square loss regression objectives and cross-entropy classification objectives are supported in this release.
Explanation

Note that the labels need to be passed to the `explain` method in order to obtain the explanation:

```python
explanation = explainer.explain(X, y)
```

15.2.3 Miscellaneous

Runtime considerations

Adjusting the size of the reference dataset

The algorithm automatically warns the user if a background dataset size of more than 1000 samples is passed. If the runtime of an explanation with the original dataset is too large, then the algorithm can automatically subsample the background dataset during the `fit` step. This can be achieve by specifying the fit step as

```python
explainer.fit(
    X_reference,
    summarise_background=True,
    n_background_samples=300,
)
```

or

```python
explainer.fit(
    X_reference,
    summarise_background='auto'
)
```

The `auto` option will select 1000 examples, whereas using the boolean argument allows the user to directly control the size of the reference set. If categorical variables are specified, the algorithm uses subsampling of the data. Otherwise, a kmeans clustering algorithm is used to select the background dataset.

As describe above, the explanations are performed with respect to the expected output over this dataset so the shap values will be affected by the dataset selection. We recommend experimenting with various ways to choose the background dataset before deploying explanations.

15.3 Theoretical overview

Recall that, for a model $f$, the Kernel SHAP algorithm [1] explains a certain outcome with respect to a chosen reference (or an expected value) by estimating the shap values of each feature $i$ from $\{1, ..., M\}$, as follows:

- enumerate all subsets $S$ of the set $F \setminus \{i\}$
- for each $S \subseteq F \setminus \{i\}$, compute the contribution of feature $i$ as $C(i|S) = f(S \cup \{i\}) - f(S)$
- compute the shap value according to

$$
\phi_i := \frac{1}{M} \sum_{S \subseteq F \setminus \{i\}} \frac{1}{M - |S|} C(i|S).
$$

Since most models do not accept arbitrary patterns of missing values at inference time, $f(S)$ needs to be approximated. The original formulation of the Kernel Shap algorithm [1] proposes to compute $f(S)$ as the observational conditional
expectation

\[ f(S) := \mathbb{E}[f(x_S, X_{\bar{S}} | X_S = x_S)] \]

where the expectation is taken over a background dataset, \( \mathcal{D} \), after conditioning. Computing this expectation involves drawing sufficiently many samples from \( X_{\bar{S}} \) for every sample from \( X_S \), which is expensive. Instead, (2) is approximated by

\[ f(S) := \mathbb{E}[f(x_S, X_{\bar{S}})] \]

where features in a subset \( S \) are fixed and features in \( \bar{S} \) are sampled from the background dataset. This quantity is referred to as marginal or interventional conditional expectation, to emphasise that setting features in \( S \) to the values \( x_S \) can be viewed as an intervention on the instance to be explained.

As described in [2], if estimating impact of a feature \( i \) on the function value by \( \mathbb{E}[f | X_i = x_i] \), one should bear in mind that observing \( X_i = x_i \) changes the distribution of the features \( X_{j \neq i} \) if these variables are correlated. Hence, if the conditional expectation if used to estimate \( f(S) \), the Shapley values might not be accurate since they also depend on the remaining variables, effect which becomes important if there are strong correlations amongst the independent variables. Furthermore, the authors show that estimating \( f(S) \) using the conditional expectation violates the sensitivity principle, according to which the Shapley value of a redundant variable should be 0. On the other hand, the intervention breaks the dependencies, ensuring that the sensitivity holds. One potential drawback of this method is that setting a subset of values to certain values without regard to the values of the features in the complement (i.e., \( \bar{S} \)) can generate instances that are outside the training data distribution, which will affect the model prediction and hence the contributions.

The following sections detail how these methods work and how, unlike Kernel SHAP, compute the exact shap values in polynomial time. The algorithm estimating contributions using interventional expectations is presented, with the remaining sections being dedicated to presenting an approximate algorithm for evaluating the interventional expectation that does not require a background dataset and Shapley interaction values.

15.3.1 Interventional feature perturbation

The interventional feature perturbation algorithm provides an efficient way to calculate the expectation \( f(S) := \mathbb{E}[f(x_S, X_{\bar{S}})] \) for all possible subsets \( S \), and to combine these values according to equation (1) in order to obtain the Shapley value. Intuitively, one can proceed as follows:

- choose a background sample \( r \in \mathcal{D} \)
- for each feature \( i \), enumerate all subsets \( S \subseteq F \setminus \{i\} \)
- for each such subset, \( S \), compute \( f(S) \) by traversing the tree with a hybrid sample where the features in \( \bar{S} \) are replaced by their corresponding values in \( r \)
- combine results according to equation (1)

If \( R \) samples from the background distribution are used, then the complexity of this algorithm is \( O(RM2^M) \) since we perform \( 2^M \) enumerations for each of the \( M \) features, \( R \) times. The key insight into this algorithm is that multiple hybrid samples will end up traversing identical paths and that this can be avoided if the shap values’ calculation is reformulated as a summation over the paths in the tree (see [4] for a proof):

\[ \phi_i = \sum_P \phi_i^P \]

where the summation is over paths \( P \) in the tree descending from \( i \). The value and sign of the contribution of each path descending through a node depends on whether the split from the node is due to a foreground or a background feature, as explained in the practical example below.
Computing contributions with interventional Tree SHAP: a practical example.

Figure 2: Illustration of the feature contribution and expected value estimation process using interventional perturbation Tree SHAP. The positive and the negative contributions of a node are represented in green and red, respectively.

In the figure above, the paths followed due the instance to be explained $x$ are coloured in red, paths followed due to the background sample in red, and common paths in yellow.

The instance to be explained is perturbed using a reference sample by the values of the features $F_1$, $F_3$ and $F_5$ in $x$ with the corresponding values in $r$. This process gives the name of the algorithm since following the paths indicated by the background sample is akin to intervening on the instance to be explained with features from the background sample. Therefore, one defines the set $F$ in the previous section as $F = \{ j : x_j \neq r_j \}$ for this case. Note that these are the only features for which one can estimate a contribution given this background sample; the same path is followed for features $F_2$ and $F_4$ for both the original and the perturbed sample, so these features do not contribute to explain the difference between the observed outcome ($v_6$) and the outcome that would have been observed if the tree had been traversed according to the reference ($v_{10}$).

Considering the structure of the tree for the given $x$ and $r$ together with equation (1) reveals that the left subtree can be traversed to compute the negative terms in the summation whereas the right subtree will provide positive terms. This is because the nodes in the left subtree can only be reached if $F_1$ takes the value from the background sample, that is, only $F_1$ is missing. Because $F_2$ and $F_4$ do not contribute to explaining $f(x) - f(r)$, the negative contribution of the left subtree will be equal to the negative contribution of node 8. This node sums two negative components: one when the downstream feature $F_5$ is also missing (corresponding to evaluating $f$ at $S = \varnothing$) and one when $F_5$ is present (corresponding to evaluating $f$ at $S = \{F_5\}$). These negative values are weighted according to the combinatorial factor in equation (1). By a similar reasoning, the nodes in the right subtree are reached only if $F_1$ is present and they provide the positive terms for the shap value computation. Note that the combinatorial factor in (1) should be evaluated with $|S| \leftarrow |S| - 1$ for positive contributions since $|S|$ is increased by 1 because of the feature whose contribution is calculated is present in the right subtree.

A similar reasoning is applied to compute the contributions of the downstream nodes. For example, to estimate the contribution of $F_5$, one considers a set $S = \varnothing$ and observes the value of node 10, and weights that with the combinatorial factor from equation (1) where $M - 1 = 1$ and $|S| = 0$ (because there are no features present on
the path) and a positive contribution from node 9 weighted by the same combinatorial factor (because $S = \{F5\}$ so $|S| - 1 = 0$).

To summarise, the efficient algorithm relies on the following key ideas:

- each node in the tree is assigned a positive contribution reflecting membership of the splitting feature in a subset $S$ and a negative contribution to indicate the feature is missing ($i \in \bar{S}$)
- the positive and negative contributions of a node can be computed by summing the positive and negative contributions of the children nodes, in keeping with the fact that the Shapley value can be computed by summing a contribution from each path the feature is on
- to compute the contribution of a feature at a node, one adds a positive contribution from the node reached by splitting on the feature from the instance to be explained and a negative contribution from the node reached by splitting on the feature in the background sample
- features for which the instance to be explained and the reference follow the same path are assigned 0 contribution.

**Explaining loss functions**

One advantage of the interventional approach is that it allows to approximately transform the shap values to account for nonlinear transformation of outputs, such as the loss function. Recall that given $\phi_1, ..., \phi_M$ the local accuracy property guarantees that given $\phi_0 = E[f(x)]$

$$f(x) = \phi_0 + \sum_{i=1}^{M} \phi_i.$$  

Hence, in order to account for the effect of the nonlinear transformation $h$, one has to find the functions $g_0, ..., g_M$ such that

$$h(f(x)) = g(\phi_0) + \sum_{i=1}^{M} g_i(\phi_i).$$

For simplicity, let $y = h(x)$. Then using a first-order Taylor series expansion around $E[y]$ one obtains

$$h(y) \approx h(E[y]) + \frac{\partial h(y)}{\partial y} \bigg|_{y=E[y]} (y - E[y]).$$

Substituting (3) in (5) and comparing coefficients with (4) yields

$$g_0 \approx h(E[y]),$$

$$g_i \approx \phi_i \frac{\partial h(y)}{\partial y} \bigg|_{y=E[y]}.$$

Hence, an approximate correction is given by simply scaling the shap values using the gradient of the nonlinear function. Note that in practice one may take the Taylor series expansion at a reference point $r$ from the background dataset and average over the entire background dataset to compute the scaling factor. This introduces an additional source of noise since $h(E[y]) = E[h(y)]$ only when $h$ is linear.
Computational complexity

For a single foreground and background sample and a single tree, the algorithm runs in $O(LD)$ time. Thus, using $R$ background samples and a model containing $T$ trees, yields a complexity of $O(TRLD)$.

15.3.2 Path dependent feature perturbation

Another way to approximate equation (2) to compute $f(S)$ given an instance $x$ and a set of missing features $\tilde{S}$ is to recursively follow the decision path through the tree and:

- return the node value if a split on a feature $i \in S$ is performed
- take a weighted average of the values returned by children if $i \in \tilde{S}$, where the weighing factor is equal to the proportion of training examples flowing down each branch. This proportion is a property of each node, sometimes referred to as weight or cover and measures how important is that node with regard to classifying the training data.

Therefore, in the path-dependent perturbation method, we compute the expectations with respect to the training data distribution by weighting the leaf values according to the proportion of the training examples that flow to that leaf.

To avoid repeating the above recursion $M^2$ times, one first notices that for a single decision tree, applying a perturbation would result in the sample ending up in a different leaf. Therefore, following each path from the root to a leaf in the tree is equivalent to perturbing subsets of features of varying cardinalities. Consequently, each leaf will contain a certain proportion of all possible subsets $S \subseteq F$. Therefore, to compute the shap values, the following quantities are computed at each leaf:

- the proportion of subsets $S$ at the leaf that contain $i$ and the proportion of subsets $S$ that do not contain $i$
- for each cardinality, the proportion of the sets of that cardinality contained at the leaf. Tracking each cardinality as opposed to a single count of subsets falling into a given leaf is necessary since it allows to apply the weighting factor in equation (1), which depends on the subset size, $|S|$.

This intuition can be summarised as follows:

$$\phi_i := \sum_{j=1}^{L} \sum_{P \in S_j} \frac{w(|P|, j)}{M_j(|P|)} (p_o^{i,j} - p_z^{i,j}) v_j$$

where $S_j$ is the set of present feature subsets at leaf $j$, $M_j$ is the length of the path and $w(|P|, j)$ is the proportion of all subsets of cardinality $P$ at leaf $j$, $p_o^{i,j}$ and $p_z^{i,j}$ represent the fractions of subsets that contain or do not contain feature $i$ respectively.

Computational complexity

Using the above quantities, one can compute the contribution of each leaf to the Shapley value of every feature. This algorithm has complexity $O(TLD^2)$ for an ensemble of trees where $L$ is the number of leaves, $T$ the number of trees in the ensemble and $D$ the maximum tree depth. If the tree is balanced, then $D = \log L$ and the complexity of our algorithm is $O(TL \log^2 L)$.
Expected value for the path-dependent perturbation algorithm

Note that although a background dataset is not provided, the expected value is computed using the node cover information, stored at each node. The computation proceeds recursively, starting at the root. The contribution of a node to the expected value of the tree is a function of the expected values of the children and is computed as follows:

\[ c_j = \frac{c_{r(j)} r_j + c_{l(j)} l_j}{r_j} \]

where \( j \) denotes the node index, \( c_j \) denotes the node expected value, \( r_j \) is the cover of the \( j \)th node and \( r(j) \) and \( l(j) \) represent the indices of the right and left children, respectively. The expected value used by the tree is simply \( c_{\text{root}} \).

The cover depends on the objective function and the model chosen. For example, in a gradient boosted tree trained with squared loss objective, \( r_j \) is simply the number of training examples flowing through \( j \). For an arbitrary objective, this is the sum of the Hessian of the loss function evaluated at each point flowing through \( j \), as explained here.

15.3.3 Shapley interaction values

While the Shapley values provide a solution to the problem of allocating a function variation to the input features, in practice it might be of interest to understand how the importance of a feature depends on the other features. The Shapley interaction values can solve this problem, by allocating the change in the function amongst the individual features (main effects) and all pairs of features (interaction effects). Thus, they are defined as

\[ \Phi_{i,j}(f, x) = \frac{1}{2^{|F\setminus\{i,j\}|}} \sum_{S \subseteq \{i,j\}} \nabla_{ij}(f, x, S), \ i \neq j \]

and

\[ \nabla_{ij}(f, x, S) = \left( f_x(S \cup \{i, j\}) - f_x(S \cup \{j\}) - [f_x(S \cup \{i\}) - f_x(S)] \right). \]

Therefore, the interaction of features \( i \) and \( j \) can be computed by taking the difference between the shap values of \( i \) when \( j \) is present and when \( j \) is not present. The main effects are defined as

\[ \Phi_{i,i}(f, x) = \phi_i(f, x) - \sum_{i \neq j} \Phi_{i,j}(f, x), \]

Setting \( \Phi_{0,0} = f_x(\emptyset) \) yields the local accuracy property for Shapley interaction values:

\[ f(x) = \sum_{i=0}^{M} \sum_{j=0}^{M} \Phi_{i,j}(f, x) \]

The interaction is split equally between feature \( i \) and \( j \), which is why the division by two appears in equation (7). The total interaction effect is defined as \( \Phi_{i,j}(f, x) = \Phi_{j,i}(f, x) \).
**Computational complexity**

According to equation (8), the interaction values can be computed by applying either the interventional or path-dependent feature perturbation algorithm twice: once by fixing the value of feature $j$ to $x_j$ and computing the shapley value for feature $i$ in this configuration, and once by fixing $x_j$ to a “missing” value and performing the same computation. Thus, the interaction values can be computed in $O(T M L D^2)$ with the path-dependent perturbation algorithm and $O(T M L D R)$ with the interventional feature perturbation algorithm.

**15.3.4 Comparison to other methods**

Tree-based models are widely used in areas where model interpretability is of interest because node-level statistics gathered from the training data can be used to provide insights into the behaviour of the model across the training dataset, providing a global explanation technique. As shown in our example, considering different statistics gives rise to different importance rankings. As discussed in [1] and [3], depending on the statistic chosen, feature importances derived from trees are not consistent, meaning that a model where a feature is known to have a bigger impact might fail to have a larger importance. As such, feature importances cannot be compared across models. In contrast, both the path-dependent and interventional perturbation algorithms tackle this limitation.

In contrast to feature importances derived from tree statistics, the Tree SHAP algorithms can also provide local explanations, allowing the identification of features that are globally “not important”, but can affect specific outcomes significantly, as might be the case in healthcare applications. Additionally, it provides a means to succinctly summarise the effect magnitude and direction (positive or negative) across potentially large samples. Finally, as shown in [1] (see here, p. 26), averaging the instance-level shap values importance to derive a global score for each feature can result in improvements in feature selection tasks.

Another method to derive instance-level explanations for tree-based model has been proposed by Sabaas here. This feature attribution method is similar in spirit to Shapley value, but does not account for the effect of variable order as explained here (pp. 10-11) as well as not satisfying consistency ([3]).

Finally, both Tree SHAP algorithms exploit model structure to provide exact Shapley values computation albeit using different estimates for the effect of missing features, achieving explanations in low-order polynomial time. The KernelShap method relies on post-hoc (black-box) function modelling and approximations to approximate the same quantities and given enough samples has been shown to to the exact values (see experiments here and our example). Our Kernel SHAP documentation provides comparisons of feature attribution methods based on Shapley values with other algorithms such as LIME and anchors.

**15.4 References**


15.5 Examples

15.5.1 Path-dependent Feature Perturbation Tree SHAP

Explaining tree models with path-dependent feature perturbation Tree SHAP

15.5.2 Interventional Feature Perturbation Tree SHAP

Explaining tree models with path-dependent feature perturbation Tree SHAP
ACCUMULATED LOCAL EFFECTS FOR PREDICTING HOUSE PRICES

In this example we will explain the behaviour of regression models on the Boston housing dataset. We will show how to calculate accumulated local effects (ALE) for determining the feature effects on a model and how these vary on different kinds of models (linear and non-linear models).

16.1 Load and prepare the dataset

[2]: data = load_boston()
   feature_names = data.feature_names
   X = data.data
   y = data.target
   print(feature_names)
   ['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO' 'B' 'LSTAT']

Shuffle the data and define the train and test set:

[3]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
16.2 Fit and evaluate models

Fit and evaluate a linear regression model:

```python
[4]: lr = LinearRegression()

[5]: lr.fit(X_train, y_train)

[5]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

[6]: mean_squared_error(y_test, lr.predict(X_test))

[6]: 22.098694827098424
```

Fit and evaluate a random forest model:

```python
[7]: rf = RandomForestRegressor()
    rf.fit(X_train, y_train)

[7]: RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                         max_depth=None, max_features='auto', max_leaf_nodes=None,
                         max_samples=None, min_impurity_decrease=0.0,
                         min_impurity_split=None, min_samples_leaf=1,
                         min_samples_split=2, min_weight_fraction_leaf=0.0,
                         n_estimators=100, n_jobs=None, oob_score=False,
                         random_state=None, verbose=0, warm_start=False)

[8]: mean_squared_error(y_test, rf.predict(X_test))

[8]: 10.387403700787404
```

16.3 Feature Effects: Motivation

Here we develop an intuition for calculating feature effects. We start by illustrating the calculation of feature effects for the linear regression model.

For our regression model, the conditional mean or the prediction function \( \mathbb{E}(y|x) = f(x) \) is linear:

\[
f(x) = w_0 + w_1 x_1 + \cdots + w_k x_k.
\]

Because the model is additive and doesn’t include feature interactions, we can read off individual feature effects immediately: the effect of any feature \( x_i \) is just \( w_i x_i \), so the effect is a linear function of \( x_i \) and the sign of the coefficient \( w_i \) determines whether the effect is positive or negative as \( x_i \) changes.

Now suppose we don’t know the true effect of the feature \( x_i \) which is usually the case when using a more complex model. How might we approach the problem of estimating the effect? Let’s focus on one feature - average number of rooms per dwelling (\( RM \)). The following is a scatterplot of model predictions versus the feature:

```python
[9]: FEATURE = 'RM'
    index = np.where(feature_names==FEATURE)[0][0]

    fig, ax = plt.subplots()
    ax.scatter(X_train[:, index], lr.predict(X_train));

    ax.set_xlabel(FEATURE);
    ax.set_ylabel('Value in $1000\'s');
```
As we can see, there is a strong positive correlation as one might expect. However the feature effects for \( RM \) cannot be read off immediately because the prediction function includes the effects of all features not just \( RM \). What we need is a procedure to block out the effects of all other features to uncover the true effect of \( RM \) only. This is exactly what the ALE approach does by averaging the differences of predictions across small intervals of the feature.

## 16.4 Calculate Accumulated Local Effects

Here we initialize the ALE object by passing it the predictor function which in this case is the `clf.predict` method for both models. We also pass in feature names and target name for easier interpretation of the resulting explanations.

```python
[10]: lr_ale = ALE(lr.predict, feature_names=feature_names, target_names=['Value in $1000\'s'])
    rf_ale = ALE(rf.predict, feature_names=feature_names, target_names=['Value in $1000\'s'])
```

We now call the `explain` method on the explainer objects which will compute the ALE’s and return an `Explanation` object which is ready for inspection and plotting. Since ALE is a global explanation method it takes in a batch of data for which the model feature effects are computed, in this case we pass in the training set.

```python
[11]: lr_exp = lr_ale.explain(X_train)
    rf_exp = rf_ale.explain(X_train)
[12]: lr_exp.feature_names
    array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
          'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')
```

The resulting `Explanation` objects contain the ALE’s for each feature under the `ale_values` attribute - this is a list of numpy arrays, one for each feature. The easiest way to interpret the ALE values is by plotting them against the feature values for which we provide a built-in function `plot_ale`. By calling the function without arguments, we will plot the effects of every feature, so in this case we will get 13 different subplots. To fit them all on the screen we pass in options for the figure size.
16.5 ALE for the linear regression model

The ALE plots show the main effects of each feature on the prediction function.

The interpretation of the ALE plot is that given a feature value, the ALE value corresponding to that feature value is difference to the mean effect of that feature. Put differently, the ALE value is the relative feature effect on the prediction at that feature value.
### 16.5.1 Effect of the average number of rooms

Let’s look at the ALE plot for the feature RM (average number of rooms per dwelling) in more detail:

```python
plot_ale(lr_exp, features=['RM']);
```

The ALE on the y-axes of the plot above is in the units of the prediction variable which, in this case, is the value of the house in $1000’s. The ALE value for the point RM=8 is ~7.5 which has the interpretation that for neighbourhoods for which the average number of rooms is ~8 the model predicts an up-lift of ~$7500 due to feature RM with respect to the average prediction. On the other hand, for neighbourhoods with an average number of rooms lower than ~6.25, the feature effect on the prediction becomes negative, i.e. a smaller number of rooms brings the predicted value down with respect to the average prediction.

Let’s find the neighbourhoods for which the average number of rooms are close to 8:

```python
lower_index = np.where(lr_exp.feature_values[5] < 8)[0][-1]
upper_index = np.where(lr_exp.feature_values[5] > 8)[0][0]
subset = X_train[(X_train[:, 5] > lr_exp.feature_values[5][lower_index]) & (X_train[:, 5] < lr_exp.feature_values[5][upper_index])]
print(subset.shape)
(6, 13)
```

The mean prediction on this subset is:

```python
subset_pred = lr.predict(subset).mean()
subset_pred
```

```
[16]: 38.71963302852489
```

The zeroth order effects is the mean prediction averaged across the whole dataset:

```python
mean_pred = lr.predict(X_train).mean()
mean_pred
```

```
[17]: 22.907915567282302
```

The difference between these two values is:}

### 16.5. ALE for the linear regression model
This is the total expected uplift in $1000’s for neighbourhoods with the average room number close to 8. The ALE value for \( RM=8 \) tells us that this feature is responsible for roughly $7.5K or almost half of the uplift while the combination of other features is responsible for the rest.

### 16.5.2 Effect of the crime level

An additional feature of the ALE plot is that it shows feature deciles on the x-axis. This helps understand in which regions there is no data so the ALE plot is interpolating. For example, for the CRIM feature (per capita crime rate by town), there is very little to no data in the feature interval 30-85, so the ALE plot in that region is just linearly interpolating:

```
[19]: plot_ale(lr_exp, features=['CRIM']);
```

For linear models this is not an issue as we know the effect is linear across the whole range of the feature, however for non-linear models linear interpolation in feature areas with no data could be unreliable. This is why the use of deciles can help assess in which areas of the feature space the estimated feature effects are more reliable.

### 16.5.3 Linearity of ALE

It is no surprise that the ALE plots for the linear regression model are linear themselves—the feature effects are after all linear by definition. In fact, the slopes of the ALE lines are exactly the coefficients of the linear regression:

```
[20]: lr.coef_
```

```
[20]: array([-1.28322638e-01, 2.95517751e-02, 4.88590934e-02, 2.77350326e+00,
   -1.62388292e+01, 4.36875476e+00, -9.24808158e-03, -1.40086668e+00,
   2.57761243e-01, -9.95694820e-03, -9.23122944e-01, 1.31854199e-02,
   -5.17639519e-01])
```

```
[21]: slopes = np.array([(v[-1]-v[0])/(f[-1]-f[0])).item() for v, f in zip(lr_exp.ale_values, lr_exp.feature_values)])
```
Thus the slopes of the ALE plots for linear regression have exactly the same interpretation as the coefficients of the learnt model—global feature effects. In fact, we can calculate the ALE effect of linear regression analytically (with some assumptions on the conditional feature distributions) to show that the effect of feature \( x_i \) is
\[
\text{ALE}(x_i) = w_i x_i - w_i \overline{x}(x_i)
\]
which is the familiar effect \( w_i x_i \) relative to the mean effect of the feature.

### 16.6 ALE for the random forest model

Now let’s look at the ALE plots for the non-linear random forest model:

```python
axes = plot_ale(rf_exp, fig_kw={'figwidth':10, 'figheight': 10});
```
Because the model is no longer linear, the ALE plots are non-linear also and in some cases also non-monotonic. The interpretation of the plots is still the same—the ALE value at a point is the relative feature effect with respect to the mean feature effect, however the non-linear model used shows that the feature effects differ both in shape and magnitude when compared to the linear model.

From these plots, it seems that the feature $\text{RM}$ has the biggest impact on the prediction. Checking the feature importances of the random forest classifier confirms this:

```
[24]: feature_names[rf.feature_importances_.argmax()]

[24]: 'RM'
```

Let’s explore the feature $\text{DIS}$ (weighted distances to five Boston employment centres) and how its effects are different between the two models. To do this, we can pass in matplotlib `axes` objects for the `plot_ale` function to plot on:
From this plot we can gather several interesting insights:

- Linear regression puts a higher emphasis on the \texttt{DIS} feature, as evidenced by the relative magnitudes of the ALE scores across the feature range.
- Whilst the linear regression feature effects of \texttt{DIS} are negatively correlated (the higher the distance to employment centres, the lower the predicted value), the random forest feature effects are not monotonic.
- In particular, at the start of the range of \texttt{DIS} there seems to be switching between positive and negative effects.

To compare multiple models and multiple features we can plot the ALE's on a common axis that is big enough to accommodate all features of interest:
Chapter 16. Accumulated Local Effects for predicting house prices
ACCUMULATED LOCAL EFFECTS FOR CLASSIFYING FLOWERS

In this example we will explain the behaviour of classification models on the Iris dataset. It is recommended to first read the *ALE regression example* to familiarize yourself with how to interpret ALE plots in a simpler setting. Interpreting ALE plots for classification problems become more complex due to a few reasons:

- Instead of one ALE line for each feature we now have one for each class to explain the feature effects towards predicting each class.
- There are two ways to choose the prediction function to explain:
  - Class probability predictions (e.g. `clf.predict_proba` in `sklearn`)
  - Margin or logit predictions (e.g. `clf.decision_function` in `sklearn`)

We will see the implications of explaining each of these prediction functions.

```python
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from alibi.explainers.ale import ALE, plot_ale
```

### 17.1 Load and prepare the dataset

```python
data = load_iris()
feature_names = data.feature_names
target_names = data.target_names
X = data.data
y = data.target
print(feature_names)
print(target_names)

['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
['setosa' 'versicolor' 'virginica']
```

Shuffle the data and define the train and test set:

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```
17.2 Fit and evaluate a logistic regression model

```python
[4]: lr = LogisticRegression(max_iter=200)

[5]: lr.fit(X_train, y_train)
```

```python
[5]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                 intercept_scaling=1, l1_ratio=None, max_iter=200,
                 multi_class='auto', n_jobs=None, penalty='l2',
                 random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                 warm_start=False)
```

```python
[6]: accuracy_score(y_test, lr.predict(X_test))
[6]: 1.0
```

17.3 Calculate Accumulated Local Effects

There are several options for explaining the classifier predictions using ALE. We define two prediction functions, one in the unnormalized logit space and the other in probability space, and look at how the resulting ALE plot interpretation changes.

```python
[7]: logit_fun_lr = lr.decision_function
    proba_fun_lr = lr.predict_proba

[8]: logit_ale_lr = ALE(logit_fun_lr, feature_names=feature_names, target_names=target_names)
    proba_ale_lr = ALE(proba_fun_lr, feature_names=feature_names, target_names=target_names)

[9]: logit_exp_lr = logit_ale_lr.explain(X_train)
    proba_exp_lr = proba_ale_lr.explain(X_train)
```

17.4 ALE in logit space

We first look at the ALE plots for explaining the feature effects towards the unnormalized logit scores:

```python
[10]: plot_ale(logit_exp_lr, n_cols=2, fig_kw=('figwidth': 8, 'figheight': 5), sharey=None);
```
We see that the feature effects are linear for each class and each feature. This is exactly what we expect because the logistic regression is a linear model in the logit space.

Furthermore, the units of the ALE plots here are in logits, which means that the feature effect at some feature value will be a positive or negative contribution to the logit of each class with respect to the mean feature effect.

Let's look at the interpretation of the feature effects for “petal length” in more detail:

```python
[11]: plot_ale(logit_exp_lr, features=[2]);
```

The ALE lines cross the 0 mark at ~3.8cm which means that for instances of petal length around ~3.8cm the feature effect on the prediction is the same as the average feature effect. On the other hand, going towards the extreme values of the feature, the model assigns a large positive/negative penalty towards classifying instances as “setosa” and vice
versa for “virginica”.

We can go into a bit more detail about the “mean response” at the petal length around ~3.8cm. First, we calculate the mean response (in logit space) of the model on the training set:

```
[12]: mean_logits = logit_fun_lr(X_train).mean(axis=0)
mean_logits
```

```
array([-0.64214307, 2.26719121, -1.62504814])
```

Next, we find instances for which the feature “petal length” is close to ~3.8cm and look at the predictions for these:

```
[13]: lower_index = np.where(logit_exp_lr.feature_values[2] < 3.8)[0][-1]
upper_index = np.where(logit_exp_lr.feature_values[2] > 3.8)[0][0]
subset = X_train[(X_train[:, 2] > logit_exp_lr.feature_values[2][lower_index]) & (X_train[:, 2] < logit_exp_lr.feature_values[2][upper_index])]
print(subset.shape)
```

```
(8, 4)
```

```
[14]: subset_logits = logit_fun_lr(subset).mean(axis=0)
subset_logits
```

```
array([-1.33625605, 2.32669999, -0.99044394])
```

Now if we subtract the logits of the instances for which petal length is ~3.8 from the mean logits, because ALE(3.8) = 0 for petal length, any difference from zero must be due to the combined effect of all other features (except petal length):

```
[15]: mean_logits - subset_logits
```

```
array([ 0.69411298, -0.05950878, -0.6346042])
```

For example, the remaining 3 features combined must be responsible for a positive effect of around ~0.69 on predicting instances with petal length ~3.8cm to be of class setosa.

This is true only because the model is linear, so, by calculating the ALE of a feature, we account for all of the effects of that feature. For non-linear models, however, there might be higher order interaction effects of the feature in question with other features responsible for the difference from the mean effects.

We can gain even more insight into the ALE plot by looking at the class histograms for the feature petal length:

```
[16]: fig, ax = plt.subplots()
    for target in range(3):
        ax.hist(X_train[y_train==target][:,2], label=target_names[target]);
    ax.set_xlabel(feature_names[2])
    ax.legend();
```
Here we see that the three classes are very well separated by this feature. This confirms that the ALE plot is behaving as expected—the feature effects of small value of “petal length” are that of increasing the the logit values for the class “setosa” and decreasing for the other two classes. Also note that the range of the ALE values for this feature is particularly high compared to other features which can be interpreted as the model attributing more importance to this feature as it separates the classes well on its own.

### 17.5 ALE in probability space

We now turn to interpreting the ALE plots for explaining the feature effects on the probabilities of each class.
As expected, the ALE plots are no longer linear which reflects the non-linear nature due to the softmax transformation applied to the logits.

Note that, in this case, the ALE are in the units of relative probability mass, i.e. given a feature value how much more (less) probability does the model assign to each class relative to the mean prediction. This also means that any increase in relative probability of one class must result into a decrease in probability of another class. In fact, the ALE curves summed across classes result in 0 as a direct consequence of conservation of probability:

```
[18]: for feature in range(4):
    print(proba_exp_lr.ale_values[feature].sum())
-5.551115123125783e-17
1.734723475976807e-17
-6.661338147750939e-16
4.440892098500626e-16
```

By transforming the ALE plots into probability space we can gain additional insight into the model behaviour. For example, the ALE curve for the feature petal width and class setosa is virtually flat. This means that the model does not use this feature to assign higher or lower probability to class setosa with respect to the average prediction. This is not readily seen in logit space as the ALE curve has negative slope which would lead us to the opposite conclusion. The interpretation here is that even though the ALE curve in the logit space shows a negative effect with feature value, the effect in the logit space is not significant enough to translate into a tangible effect in the probability space.

Finally, the feature sepal width does not offer significant information to the model to prefer any class over the other (with respect to the mean effect of sepal_width that is). If we plot the marginal distribution of sepal_width it explains why that is—the overlap in the class conditional histograms of this feature show that it does not increase the model discriminative power:

```
[19]: fig, ax = plt.subplots()
for target in range(3):
    ax.hist(X_train[y_train==target][:,1], label=target_names[target]);
ax.set_xlabel(feature_names[1])
ax.legend();
```

![Histogram of sepal width for three classes](image)
17.6 ALE for gradient boosting

Finally, we look at the resulting ALE plots for a highly non-linear model—a gradient boosted classifier.

```python
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier()
gb.fit(X_train, y_train)
gb = GradientBoostingClassifier(ccp_alpha=0.0, criterion='friedman_mse', init=None, learning_rate=0.1, loss='deviance', max_depth=3, max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=100, n_iter_no_change=None, presort='deprecated', random_state=None, subsample=1.0, tol=0.0001, validation_fraction=0.1, verbose=0, warm_start=False)
```

```python
accuracy_score(y_test, gb.predict(X_test))
```

As before, we explain the feature contributions in both logit and probability space.

```python
logit_fun_gb = gb.decision_function
proba_fun_gb = gb.predict_proba
logit_ale_gb = ALE(logit_fun_gb, feature_names=feature_names, target_names=target_names)
proba_ale_gb = ALE(proba_fun_gb, feature_names=feature_names, target_names=target_names)
logit_exp_gb = logit_ale_gb.explain(X_train)
proba_exp_gb = proba_ale_gb.explain(X_train)
```

17.6.1 ALE in logit space

```python
plot_ale(logit_exp_gb, n_cols=2, fig_kw={'figwidth': 8, 'figheight': 5})
```
The ALE curves are no longer linear as the model used is non-linear. Furthermore, we’ve plotted the ALE curves of different features on the same scale on the $y$-axis which suggests that the features petal length and petal width are more discriminative for the task. Checking the feature importances of the classifier confirms this:

```python
[27]: gb.feature_importances_
```

```python
[27]: array([0.00220678, 0.01643393, 0.53119079, 0.4501685])
```

### 17.6.2 ALE in probability space

```python
[28]: plot_ale(proba_exp_gb, n_cols=2, fig_kw=('figwidth': 8, 'figheight': 5));
```
Because of the non-linearity of the gradient boosted model the ALE curves in probability space are very similar to the curves in the logit space just on a different scale.

### 17.6.3 Comparing ALE between models

We have seen that for both logistic regression and gradient boosting models the features *petal length* and *petal width* have a high feature effect on the classifier predictions. We can explore this in more detail by comparing the ALE curves for both models. In the following we plot the ALE curves of the two features for predicting the class *setosa* in probability space:

```python
[29]: fig, ax = plt.subplots(1, 2, figsize=(8, 4), sharey='row');
    plot_ale(proba_exp_lr, features=[2, 3], targets=['setosa'], ax=ax, line_kw={'label': 'LR'});
    plot_ale(proba_exp_gb, features=[2, 3], targets=['setosa'], ax=ax, line_kw={'label': 'GB'});
```

17.6. ALE for gradient boosting 105
From this plot we can draw a couple of conclusions:

- Both models have similar feature effects of petal length—a high positive effect for predicting setosa for small feature values and a high negative effect for large values (over >3cm).

- While the logistic regression model does not benefit from the petal width feature to discriminate the setosa class, the gradient boosted model does exploit this feature to discern between different classes.
ANCHOR EXPLANATIONS FOR INCOME PREDICTION

In this example, we will explain predictions of a Random Forest classifier whether a person will make more or less than $50k based on characteristics like age, marital status, gender or occupation. The features are a mixture of ordinal and categorical data and will be pre-processed accordingly.

```python
[1]: import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from alibi.explainers import AnchorTabular
from alibi.datasets import fetch_adult

18.1 Load adult dataset

The `fetch_adult` function returns a Bunch object containing the features, the targets, the feature names and a mapping of categorical variables to numbers which are required for formatting the output of the Anchor explainer.

```python
[2]: adult = fetch_adult()
adult.keys()

[2]: dict_keys(['data', 'target', 'feature_names', 'target_names', 'category_map'])

[3]: data = adult.data
target = adult.target
feature_names = adult.feature_names
category_map = adult.category_map

Note that for your own datasets you can use our utility function `gen_category_map` to create the category map:

```python
[4]: from alibi.utils.data import gen_category_map

Define shuffled training and test set

```python
[5]: np.random.seed(0)
data_perm = np.random.permutation(np.c_[data, target])
data = data_perm[:, :-1]
target = data_perm[:, -1]
18.2 Create feature transformation pipeline

Create feature pre-processor. Needs to have ‘fit’ and ‘transform’ methods. Different types of pre-processing can be applied to all or part of the features. In the example below we will standardize ordinal features and apply one-hot-encoding to categorical features.

Ordinal features:

```python
ordinal_features = [x for x in range(len(feature_names)) if x not in list(category_map.keys())]
ordinal_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')), ('scaler', StandardScaler())])
```

Categorical features:

```python
categorical_features = list(category_map.keys())
categorical_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')), ('onehot', OneHotEncoder(handle_unknown='ignore'))])
```

Combine and fit:

```python
preprocessor = ColumnTransformer(transformers=[('num', ordinal_transformer, ordinal_features), ('cat', categorical_transformer, categorical_features)])
preprocessor.fit(X_train)
```

(continues on next page)
18.3 Train Random Forest model

Fit on pre-processed (imputing, OHE, standardizing) data.

```python
[10]: np.random.seed(0)
clf = RandomForestClassifier(n_estimators=50)
clf.fit(preprocessor.transform(X_train), Y_train)
```

Define predict function

```python
[11]: predict_fn = lambda x: clf.predict(preprocessor.transform(x))
print('Train accuracy: ', accuracy_score(Y_train, predict_fn(X_train)))
print('Test accuracy: ', accuracy_score(Y_test, predict_fn(X_test)))
```

Train accuracy: 0.9655333333333334
Test accuracy: 0.855859375

18.4 Initialize and fit anchor explainer for tabular data

```python
[12]: explainer = AnchorTabular(predict_fn, feature_names, categorical_names=category_map, seed=1)
```

Discretize the ordinal features into quartiles

```python
[13]: explainer.fit(X_train, disc_perc=[25, 50, 75])
```
18.5 Getting an anchor

Below, we get an anchor for the prediction of the first observation in the test set. An anchor is a sufficient condition - that is, when the anchor holds, the prediction should be the same as the prediction for this instance.

```python
idx = 0
class_names = adult.target_names
print('Prediction: ', class_names[explainer.predictor(X_test[idx].reshape(1, -1))[0]])
Prediction: <=50K
```

We set the precision threshold to 0.95. This means that predictions on observations where the anchor holds will be the same as the prediction on the explained instance at least 95% of the time.

```python
explanation = explainer.explain(X_test[idx], threshold=0.95)
print('Anchor: %s
Precision: %.2f
Coverage: %.2f
' % (' AND '.join(explanation.anchor), explanation.precision, explanation.coverage))
Anchor: Marital Status = Separated AND Sex = Female
Precision: 0.95
Coverage: 0.18
```

18.6 ...or not?

Let’s try getting an anchor for a different observation in the test set - one for which the prediction is >50K.

```python
idx = 6
class_names = adult.target_names
print('Prediction: ', class_names[explainer.predictor(X_test[idx].reshape(1, -1))[0]])
explanation = explainer.explain(X_test[idx], threshold=0.95)
print('Anchor: %s
Precision: %.2f
Coverage: %.2f
' % (' AND '.join(explanation.anchor), explanation.precision, explanation.coverage))
Prediction: >50K
Anchor: Capital Loss > 0.00 AND Relationship = Husband AND Marital Status = Married
AND Age > 37.00 AND Race = White AND Country = United-States AND Sex = Male
Precision: 0.71
Coverage: 0.05
```

Notice how no anchor is found!
This is due to the imbalanced dataset (roughly 25:75 high:low earner proportion), so during the sampling stage feature ranges corresponding to low-earners will be oversampled. This is a feature because it can point out an imbalanced dataset, but it can also be fixed by producing balanced datasets to enable anchors to be found for either class.
CHAPTER NINETEEN

ANCHOR EXPLANATIONS ON THE IRIS DATASET

19.1 Load iris dataset

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from alibi.explainers import AnchorTabular

dataset = load_iris()
feature_names = dataset.feature_names
class_names = list(dataset.target_names)
```

Define training and test set

```python
idx = 145
X_train, Y_train = dataset.data[:idx, :], dataset.target[:idx]
X_test, Y_test = dataset.data[idx+1:, :], dataset.target[idx+1:]
```

19.2 Train Random Forest model

```python
np.random.seed(0)
clf = RandomForestClassifier(n_estimators=50)
clf.fit(X_train, Y_train)
```

```python
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
criterion='gini', max_depth=None, max_features='auto',
max_leaf_nodes=None, max_samples=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, n_estimators=50,
n_jobs=None, oob_score=False, random_state=None,
verbose=0, warm_start=False)
```

Define predict function

```python
predict_fn = lambda x: clf.predict_proba(x)
```
19.3 Initialize and fit anchor explainer for tabular data

```python
explainer = AnchorTabular(predict_fn, feature_names)
```

Discretize the ordinal features into quartiles

```python
explainer.fit(X_train, disc_perc=(25, 50, 75))
```

```python
AnchorTabular(meta={
    'name': 'AnchorTabular',
    'type': ['blackbox'],
    'explanations': ['local'],
    'params': {'seed': None, 'disc_perc': (25, 50, 75)}
})
```

19.4 Getting an anchor

Below, we get an anchor for the prediction of the first observation in the test set. An anchor is a sufficient condition - that is, when the anchor holds, the prediction should be the same as the prediction for this instance.

```python
idx = 0
print('Prediction: ', class_names[explainer.predictor(X_test[idx].reshape(1, -1))[0]])
Prediction: virginica
```

We set the precision threshold to 0.95. This means that predictions on observations where the anchor holds will be the same as the prediction on the explained instance at least 95% of the time.

```python
explanation = explainer.explain(X_test[idx], threshold=0.95)
print('Anchor: %s
' % (' AND '.join(explanation.anchor)))
print('Precision: %.2f' % explanation.precision)
print('Coverage: %.2f' % explanation.coverage)
```

Anchor: petal width (cm) > 1.80 AND sepal width (cm) <= 2.80
Precision: 0.98
Coverage: 0.32
ANCHOR EXPLANATIONS FOR MOVIE SENTIMENT

In this example, we will explain why a certain sentence is classified by a logistic regression as having negative or positive sentiment. The logistic regression is trained on negative and positive movie reviews.

```python
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
import spacy
from alibi.explainers import AnchorText
from alibi.datasets import fetch_movie_sentiment
from alibi.utils.download import spacy_model
```

20.1 Load movie review dataset

The `fetch_movie_sentiment` function returns a `Bunch` object containing the features, the targets and the target names for the dataset.

```python
movies = fetch_movie_sentiment()
movies.keys()
```

```python
dict_keys(['data', 'target', 'target_names'])
```

```python
data = movies.data
labels = movies.target
target_names = movies.target_names
```

Define shuffled training, validation and test set

```python
train, test, train_labels, test_labels = train_test_split(data, labels, test_size=.2, random_state=42)
train, val, train_labels, val_labels = train_test_split(train, train_labels, test_size=.1, random_state=42)
train_labels = np.array(train_labels)
test_labels = np.array(test_labels)
val_labels = np.array(val_labels)
```
20.2 Apply CountVectorizer to training set

```python
vectorizer = CountVectorizer(min_df=1)
vectorizer.fit(train)
```

```python
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
               dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
               lowercase=True, max_df=1.0, max_features=None, min_df=1,
               ngram_range=(1, 1), preprocessor=None, stop_words=None,
               strip_accents=None, token_pattern='(?u)\b\w\w+\b',
               tokenizer=None, vocabulary=None)
```

20.3 Fit model

```python
np.random.seed(0)
clf = LogisticRegression(solver='liblinear')
clf.fit(vectorizer.transform(train), train_labels)
```

```python
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                    intercept_scaling=1, l1_ratio=None, max_iter=100,
                    multi_class='auto', n_jobs=None, penalty='l2',
                    random_state=None, solver='liblinear', tol=0.0001, verbose=0,
                    warm_start=False)
```

20.4 Define prediction function

```python
predict_fn = lambda x: clf.predict(vectorizer.transform(x))
```

20.5 Make predictions on train and test sets

```python
preds_train = predict_fn(train)
preds_val = predict_fn(val)
preds_test = predict_fn(test)
print('Train accuracy', accuracy_score(train_labels, preds_train))
print('Validation accuracy', accuracy_score(val_labels, preds_val))
print('Test accuracy', accuracy_score(test_labels, preds_test))
```

Train accuracy 0.9801624284382905
Validation accuracy 0.7544910179640718
Test accuracy 0.7589841878294202
20.6 Load spaCy model

English multi-task CNN trained on OntoNotes, with GloVe vectors trained on Common Crawl. Assigns word vectors, context-specific token vectors, POS tags, dependency parse and named entities.

```python
[9]: model = 'en_core_web_md'
    spacy_model(model=model)
    nlp = spacy.load(model)
```

20.7 Initialize anchor text explainer

```python
[10]: explainer = AnchorText(nlp, predict_fn)
```

20.8 Explain a prediction

```python
[11]: class_names = movies.target_names

[12]: text = data[4]
    print(text)
    a visually flashy but narratively opaque and emotionally vapid exercise in style and mystification.

Prediction:

```python
[13]: pred = class_names[predict_fn([text])[0]]
    alternative = class_names[1 - predict_fn([text])[0]]
    print('Prediction: %s' % pred)
    print('Prediction: negative

Explanation:

```python
[14]: np.random.seed(0)
    explanation = explainer.explain(text, threshold=0.95, use_unk=True)

use_unk=True means we will perturb examples by replacing words with UNKS. Let us now take a look at the anchor. The word ‘exercise’ basically guarantees a negative prediction.

```python
[15]: print('Anchor: %s' % (' AND '.join(explanation.anchor)))
    print('Precision: %.2f' % explanation.precision)
    print('
Examples where anchor applies and model predicts %s:' % pred)
    print('
Examples where anchor applies and model predicts %s:' % alternative)
    print('Anchor: flashy
    Precision: 0.99

Examples where anchor applies and model predicts negative:
    a UNK flashy UNK UNK opaque and emotionally vapid exercise in style UNK mystification.'
20.9 Changing the perturbation distribution

Let’s try this with another perturbation distribution, namely one that replaces words by similar words instead of UNKs.

Explanation:

```python
np.random.seed(0)
explanation = explainer.explain(text, threshold=0.95, use_unk=False, sample_proba=0.5)
```

The anchor now shows that we need more to guarantee the negative prediction:

```python
Anchor: exercise AND emotionally
Precision: 0.97
```

Examples where anchor applies and model predicts negative:

- a instantly flashy but narratively opaque and emotionally vapid exercise than style and mystification.
- some visually quirky but narratively dainty and emotionally patronising exercise until style and hopelessness.
- another visually decorate but narratively opaque and emotionally vapid exercise in outfit and mystification.
- both curiously unwieldy but wonderfully hollow and emotionally ridiculous exercise in style and mystification.
- a visually artsy but gloriously opaque and emotionally vapid exercise in style and mystification.
- a visually eclectic but narratively opaque and emotionally vapid exercise in vogue and mystification.
- another stunningly flashy but narratively bright and emotionally unscientific exercise on style and falsehood.

(continues on next page)
a exceptionally whimsical but disturbingly opaque and emotionally vapid exercise about vibe and mystification.

Examples where anchor applies and model predicts positive:
both visually unconventional but socially opaque and emotionally caricature exercise around style and woe.
built wonderfully artsy but similarly opaque and emotionally vapid exercise in style and mystification.
a perfectly groovy but supremely smooth and emotionally vapid exercise towards style and mystification.
a visually stylish but narratively truthful and emotionally moronic exercise in style and oxymoron.
any remarkably inventive but narratively opaque and emotionally babble exercise despite style and naivety.

We can make the token perturbation distribution sample words that are more similar to the ground truth word via the top_n argument. Smaller values (default=100) should result in sentences that are more coherent and thus more in the distribution of natural language which could influence the returned anchor. By setting the use_probability_proba to True, the sampling distribution for perturbed tokens is proportional to the similarity score between the possible perturbations and the original word. We can also put more weight on similar words via the temperature argument. Lower values of temperature increase the sampling weight of more similar words. The following example will perturb tokens in the original sentence with probability equal to sample_proba. The sampling distribution for the perturbed tokens is proportional to the similarity score between the ground truth word and each of the top_n words.

```python
np.random.seed(0)
explanation = explainer.explain(text, threshold=0.95, use_similarity_proba=True,
--sample_proba=0.5,
    use_unk=False, top_n=20, temperature=.2)
```

Anchor: exercise AND emotionally
Precision: 0.98

Examples where anchor applies and model predicts negative:
a visually exquisite but narratively opaque and emotionally vapid exercise before style and mystification.
each mechanically eccentric but narratively transparent and emotionally unremarkable exercise in style and falsehood.
a incredibly extravagant but artistically bright and emotionally vapid exercise of style and mystification.
any visually shiny but artistically glide and emotionally vapid exercise around temperament and materialism.
another clearly flashy but aesthetically opaque and emotionally vapid exercise whether flair and mystification.
a visually snazzy but narratively opaque and emotionally mindless exercise within style and negation.
a visually ingenious but narratively opaque and emotionally unimaginative exercise in artistry and mystification.
a visually flashy but narratively colorful and emotionally vapid exercise than style and mystification.
a graphically punchy but narratively opaque and emotionally vapid exercise of vibe and insanity.

a artistically flashy but narratively opaque and emotionally vapid exercise in ballroom and mystification.

Examples where anchor applies and model predicts positive:
any vividly outlandish but supremely opaque and emotionally vapid exercise throughout streetwear and mystification.
another precisely elaborate but delightfully realistic and emotionally muddled exercise in brevity and paranoia.
ANCHOR EXPLANATIONS FOR IMAGENET

21.1 Load InceptionV3 model pre-trained on ImageNet

```python
[2]: model = InceptionV3(weights='imagenet')
```

21.2 Download and preprocess some images from ImageNet

The `fetch_imagenet` function takes as arguments any of the 1000 ImageNet categories as well as the number of images to return and the target size of the image.

```python
[3]: category = 'Persian cat'
    image_shape = (299, 299, 3)
    data, labels = fetch_imagenet(category, nb_images=10, target_size=image_shape[:2],
    →seed=2, return_X_y=True)
    print(f'Images shape: {data.shape}')
```

Images shape: (10, 299, 299, 3)

Apply image preprocessing, make predictions and map predictions back to categories. The output label is a tuple which consists of the class name, description and the prediction probability.

```python
[4]: images = preprocess_input(data)
    preds = model.predict(images)
    label = decode_predictions(preds, top=3)
    print(label[0])
```

[('n02123394', 'Persian_cat', 0.909348), ('n03207941', 'dishwasher', 0.0027691855), ('n03832673', 'notebook', 0.0020055235)]
21.3 Define prediction function

```python
predict_fn = lambda x: model.predict(x)
```

21.4 Initialize anchor image explainer

The segmentation function will be used to generate superpixels. It is important to have meaningful superpixels in order to generate a useful explanation. Please check scikit-image’s segmentation methods (felzenszwalb, slic and quickshift built in the explainer) for more information.

In the example, the pixels not in the proposed anchor will take the average value of their superpixel. Another option is to superimpose the pixel values from other images which can be passed as a numpy array to the `images_background` argument.

```python
segmentation_fn = 'slic'
kwars = {'n_segments': 15, 'compactness': 20, 'sigma': .5}
explainer = AnchorImage(predict_fn, image_shape, segmentation_fn=segmentation_fn,
segmentation_kwargs=kwars, images_background=None)
```

21.5 Explain a prediction

The explanation of the below image returns a mask with the superpixels that constitute the anchor.

```python
i = 0
plt.imshow(data[i]);
```

The `threshold`, `p_sample` and `tau` parameters are also key to generate a sensible explanation and ensure fast enough convergence. The `threshold` defines the minimum fraction of samples for a candidate anchor that need to lead to the same prediction as the original instance. While a higher threshold gives more confidence in the anchor, it also leads to longer computation time. `p_sample` determines the fraction of superpixels that are changed to either the average value of the superpixel or the pixel value for the superimposed image. The pixels in the proposed anchors are of course unchanged. The parameter `tau` determines when we assume convergence. A bigger value for `tau` means faster convergence but also looser anchor restrictions.
```python
import numpy as np

np.random.seed(0)

explanation = explainer.explain(image, threshold=.95, p_sample=.5, tau=0.25)
```

Superpixels in the anchor:
```python
plt.imshow(explanation.anchor);
```

A visualization of all the superpixels:
```python
plt.imshow(explanation.segments);
```
CHAPTER TWENTYTWO

ANCHOR EXPLANATIONS FOR FASHION MNIST

22.1 Load and prepare fashion MNIST data

```
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, Dense, Dropout, Flatten, MaxPooling2D,
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical
from alibi.explainers import AnchorImage

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
print('x_train shape:', x_train.shape, 'y_train shape:', y_train.shape)

# Scale, reshape and categorize data
```

```
x_train shape: (60000, 28, 28) y_train shape: (60000,)
```

```
idx = 0
plt.imshow(x_train[idx]);
```

Scale, reshape and categorize data
22.2 Define CNN model

```python
def model():
    x_in = Input(shape=(28, 28, 1))
    x = Conv2D(filters=64, kernel_size=2, padding='same', activation='relu')(x_in)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)
    x = Conv2D(filters=32, kernel_size=2, padding='same', activation='relu')(x)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)
    x = Flatten()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.5)(x)
    x_out = Dense(10, activation='softmax')(x)
    cnn = Model(inputs=x_in, outputs=x_out)
    cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return cnn
```

```bash
cnn = model()
cnn.summary()
```

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_1 (InputLayer)</td>
<td>[(None, 28, 28, 1)]</td>
<td>0</td>
</tr>
<tr>
<td>conv2d (Conv2D)</td>
<td>(None, 28, 28, 64)</td>
<td>320</td>
</tr>
<tr>
<td>max_pooling2d (MaxPooling2D)</td>
<td>(None, 14, 14, 64)</td>
<td>0</td>
</tr>
<tr>
<td>dropout (Dropout)</td>
<td>(None, 14, 14, 64)</td>
<td>0</td>
</tr>
<tr>
<td>conv2d_1 (Conv2D)</td>
<td>(None, 14, 14, 32)</td>
<td>8224</td>
</tr>
<tr>
<td>max_pooling2d_1 (MaxPooling2D)</td>
<td>(None, 7, 7, 32)</td>
<td>0</td>
</tr>
<tr>
<td>dropout_1 (Dropout)</td>
<td>(None, 7, 7, 32)</td>
<td>0</td>
</tr>
</tbody>
</table>
22.3 Train model

[7]: `cnn.fit(x_train, y_train, batch_size=64, epochs=3)`

Train on 60000 samples
Epoch 1/3
60000/60000 [==============================] - 29s 481us/sample - loss: 0.5932 - acc: 0.7819
Epoch 2/3
60000/60000 [==============================] - 33s 542us/sample - loss: 0.4066 - acc: 0.8506
Epoch 3/3
60000/60000 [==============================] - 32s 525us/sample - loss: 0.3624 - acc: 0.8681

[7]: `<tensorflow.python.keras.callbacks.History at 0x7fae6dd5cb70>`

[8]: `# Evaluate the model on test set`  
    `score = cnn.evaluate(x_test, y_test, verbose=0)`  
    `print('Test accuracy: ', score[1])`

Test accuracy:  0.8867

22.4 Define superpixels

Function to generate rectangular superpixels for a given image. Alternatively, use one of the built in methods. It is important to have meaningful superpixels in order to generate a useful explanation. Please check scikit-image's segmentation methods (felzenszwalb, slic and quickshift built in the explainer) for more information on the built in methods.

[9]: `def superpixel(image, size=(4, 7)):`  
    `segments = np.zeros([image.shape[0], image.shape[1]])`  
    `row_idx, col_idx = np.where(segments == 0)`  
    `for i, j in zip(row_idx, col_idx):`  
        `segments[i, j] = int((image.shape[1]/size[1]) * (i//size[0]) + j//size[1])`  
    `return segments

22.3. Train model
22.5 Define prediction function

\[\textbf{11}: \quad \text{predict\_fn} = \textbf{lambda} \ x: \text{cnn.predict}(x)\]

22.6 Initialize anchor image explainer

\[\textbf{12}: \quad \text{image\_shape} = \text{x\_train}[\text{id}_x].\text{shape} \quad \text{explainer} = \text{AnchorImage}(\text{predict\_fn}, \text{image\_shape}, \text{segmentation\_fn=}\text{superpixel})\]

22.7 Explain a prediction

The explanation returns a mask with the superpixels that constitute the anchor.

Image to be explained:

\[\textbf{13}: \quad i = 1 \quad \text{image} = \text{x\_test}[i] \quad \text{plt.imshow(image[:,:,0]);}\]
Model prediction:

```python
[14]: cnn.predict(image.reshape(1, 28, 28, 1)).argmax()
[14]: 2
```

The predicted category correctly corresponds to the class **Pullover**:

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T-shirt/top</td>
</tr>
<tr>
<td>1</td>
<td>Trouser</td>
</tr>
<tr>
<td>2</td>
<td>Pullover</td>
</tr>
<tr>
<td>3</td>
<td>Dress</td>
</tr>
<tr>
<td>4</td>
<td>Coat</td>
</tr>
<tr>
<td>5</td>
<td>Sandal</td>
</tr>
<tr>
<td>6</td>
<td>Shirt</td>
</tr>
<tr>
<td>7</td>
<td>Sneaker</td>
</tr>
<tr>
<td>8</td>
<td>Bag</td>
</tr>
<tr>
<td>9</td>
<td>Ankle boot</td>
</tr>
</tbody>
</table>

Generate explanation:

```python
[15]: explanation = explainer.explain(image, threshold=.95, p_sample=.8, seed=0)
```

Show anchor:

```python
[16]: plt.imshow(explanation.anchor[:,:,0]);
```
From the example, it looks like the end of the sleeve alone is sufficient to predict a pullover.
CONTRASTIVE EXPLANATIONS METHOD (CEM) APPLIED TO MNIST

The Contrastive Explanation Method (CEM) can generate black box model explanations in terms of pertinent positives (PP) and pertinent negatives (PN). For PP, it finds what should be minimally and sufficiently present (e.g. important pixels in an image) to justify its classification. PN on the other hand identify what should be minimally and necessarily absent from the explained instance in order to maintain the original prediction.

The original paper where the algorithm is based on can be found on arXiv.

```python
import tensorflow as tf
tf.get_logger().setLevel(40) # suppress deprecation messages
tf.compat.v1.disable_v2_behavior() # disable TF2 behaviour as alibi code still relies on TF1 constructs
import tensorflow.keras as keras
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D,
    Input, UpSampling2D
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.utils import to_categorical
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
from alibi.explainers import CEM

print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # False

TF version: 2.2.0
Eager execution enabled: False
```

23.1 Load and prepare MNIST data

```python
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
print('x_train shape:', x_train.shape, 'y_train shape:', y_train.shape)
plt.gray()
plt.imshow(x_test[4]);
print('x_train shape: (60000, 28, 28) y_train shape: (60000,)
```
Prepare data: scale, reshape and categorize

```python
[3]: x_train = x_train.astype('float32') / 255
ten_test = x_test.astype('float32') / 255	x_train = np.reshape(x_train, x_train.shape + (1,))
ten_test = np.reshape(x_test, x_test.shape + (1,))
print('x_train shape:', x_train.shape, 'x_test shape:', x_test.shape)
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
print('y_train shape:', y_train.shape, 'y_test shape:', y_test.shape)
x_train shape: (60000, 28, 28, 1) x_test shape: (10000, 28, 28, 1)
y_train shape: (60000, 10) y_test shape: (10000, 10)
```

```python
[4]: xmin, xmax = -.5, .5
x_train = ((x_train - x_train.min()) / (x_train.max() - x_train.min())) * (xmax - xmin) + xmin
x_test = ((x_test - x_test.min()) / (x_test.max() - x_test.min())) * (xmax - xmin) + xmin
```

### 23.2 Define and train CNN model

```python
[5]: def cnn_model():
    x_in = Input(shape=(28, 28, 1))
x = Conv2D(filters=64, kernel_size=2, padding='same', activation='relu')(x_in)
x = MaxPooling2D(pool_size=2)(x)
x = Dropout(0.3)(x)
x = Conv2D(filters=32, kernel_size=2, padding='same', activation='relu')(x)
x = MaxPooling2D(pool_size=2)(x)
x = Dropout(0.3)(x)
x = Conv2D(filters=32, kernel_size=2, padding='same', activation='relu')(x)
x = MaxPooling2D(pool_size=2)(x)
x = Dropout(0.3)(x)
x = Flatten()(x)
(continues on next page)
```
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
x_out = Dense(10, activation='softmax')(x)

cnn = Model(inputs=x_in, outputs=x_out)
cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
return cnn

[c6]:
cnn = cnn_model()
cnn.summary()
cnn.fit(x_train, y_train, batch_size=64, epochs=5, verbose=1)
cnn.save('mnist_cnn.h5', save_format='h5')

Evaluate the model on test set

[c7]:
cnn = load_model('mnist_cnn.h5')
score = cnn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy: ', score[1])
23.3 Define and train auto-encoder

```python
[8]:
def ae_model():
    x_in = Input(shape=(28, 28, 1))
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x_in)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)
    encoded = Conv2D(1, (3, 3), activation=None, padding='same')(x)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(encoded)
    x = UpSampling2D((2, 2))(x)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    decoded = Conv2D(1, (3, 3), activation=None, padding='same')(x)

    autoencoder = Model(x_in, decoded)
    autoencoder.compile(optimizer='adam', loss='mse')
    return autoencoder

[9]:

ae = ae_model()
ae.summary()
ae.fit(x_train, x_train, batch_size=128, epochs=4, validation_data=(x_test, x_test),
       verbose=0)
ae.save('mnist_ae.h5', save_format='h5')

Compare original with decoded images
```
23.4 Generate contrastive explanation with pertinent negative

Explained instance:

```python
[11]: idx = 15
X = x_test[idx].reshape((1,) + x_test[idx].shape)
```

```python
[12]: plt.imshow(X.reshape(28, 28));
```
Model prediction:

```
[13]: cnn.predict(X).argmax(), cnn.predict(X).max()
[13]: (5, 0.99959975)
```

CEM parameters:

```
[14]: mode = 'PN' # 'PN' (pertinent negative) or 'PP' (pertinent positive)
shape = (1,) + x_train.shape[1:] # instance shape
kappa = 0. # minimum difference needed between the prediction probability for the
→perturbed instance on the
  # class predicted by the original instance and the max probability on the
→other classes
    # in order for the first loss term to be minimized
beta = .1 # weight of the L1 loss term
gamma = 100 # weight of the optional auto-encoder loss term
c_init = 1. # initial weight c of the loss term encouraging to predict a different
→class (PN) or
  # the same class (PP) for the perturbed instance compared to the
→original instance to be explained
c_steps = 10 # nb of updates for c
max_iterations = 1000 # nb of iterations per value of c
feature_range = (x_train.min(), x_train.max()) # feature range for the perturbed
→instance
clip = (-1000., 1000.) # gradient clipping
lr = 1e-2 # initial learning rate
no_info_val = -1. # a value, float or feature-wise, which can be seen as containing
→no info to make a prediction
    # perturbations towards this value means removing features, and
→away means adding features
    # for our MNIST images, the background (-0.5) is the least
→informative,
    # so positive/negative perturbations imply adding/removing features

Generate pertinent negative:

```
[15]: cem = CEM(cnn, mode, shape, kappa=kappa, beta=beta, feature_range=feature_range,
  gamma=gamma, ae_model=ae, max_iterations=max_iterations,
  c_init=c_init, c_steps=c_steps, learning_rate_init=lr, clip=clip, no_info_
→val=no_info_val)

explanation = cem.explain(X)

Pertinent negative:

```
[16]: print(f'Pertinent negative prediction: {explanation.PN_pred}')
plt.imshow(explanation.PN.reshape(28, 28));
```

Pertinent negative prediction: 3
23.5 Generate pertinent positive

```python
mode = 'PP'

cem = CEM(cnn, mode, shape, kappa=kappa, beta=beta, feature_range=feature_range,
          gamma=gamma, ae_model=ae, max_iterations=max_iterations,
          c_init=c_init, c_steps=c_steps, learning_rate_init=lr, clip=clip, no_info_
          →val=no_info_val)

explanation = cem.explain(X)

Pertinent positive:

print(f'Pertinent positive prediction: {explanation.PP_pred}"
plt.imshow(explanation.PP.reshape(28, 28));

Pertinent positive prediction: 5

Clean up:
```
```python
[ ]: os.remove('mnist_cnn.h5')
os.remove('mnist_ae.h5')
```
The Contrastive Explanation Method (CEM) can generate black box model explanations in terms of pertinent positives (PP) and pertinent negatives (PN). For PP, it finds what should be minimally and sufficiently present (e.g. important pixels in an image) to justify its classification. PN on the other hand identify what should be minimally and necessarily absent from the explained instance in order to maintain the original prediction.

The original paper where the algorithm is based on can be found on arXiv.

```python
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
from sklearn.datasets import load_iris
from alibi.explainers import CEM
```

**24.1 Load and prepare Iris dataset**

```python
dataset = load_iris()
feature_names = dataset.feature_names
class_names = list(dataset.target_names)
```

Scale data

```python
dataset.data = (dataset.data - dataset.data.mean(axis=0)) / dataset.data.std(axis=0)
```
Define training and test set

```python
idx = 145
x_train, y_train = dataset.data[:idx, :], dataset.target[:idx]
x_test, y_test = dataset.data[idx+1:, :], dataset.target[idx+1:]
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

### 24.2 Define and train logistic regression model

```python
def lr_model():
    x_in = Input(shape=(4,))
    x_out = Dense(3, activation='softmax')(x_in)
    lr = Model(inputs=x_in, outputs=x_out)
    lr.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
    return lr
```

```python
lr = lr_model()
lr.summary()
lr.fit(x_train, y_train, batch_size=16, epochs=500, verbose=0)
lr.save('iris_lr.h5', save_format='h5')
```

*Model: "model"

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_1 (InputLayer)</td>
<td>[(None, 4)]</td>
<td>0</td>
</tr>
<tr>
<td>dense (Dense)</td>
<td>(None, 3)</td>
<td>15</td>
</tr>
<tr>
<td>Total params:</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Trainable params:</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Non-trainable params:</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

### 24.3 Generate contrastive explanation with pertinent negative

Explain instance:

```python
idx = 0
X = x_test[idx].reshape((1,) + x_test[idx].shape)
print('Prediction on instance to be explained: {}\n'.format(class_names[np.argmax(lr.predict(X))]))
print('Prediction probabilities for each class on the instance: {}\n'.format(lr.predict(X)))
```

Prediction on instance to be explained: virginica
Prediction probabilities for each class on the instance: [[2.2735458e-04 2.4420770e-01 7.5556499e-01]]

CEM parameters:
alibi Documentation, Release 0.5.6

[8]:
mode = 'PN' # 'PN' (pertinent negative) or 'PP' (pertinent positive)
shape = (1,) + x_train.shape[1:] # instance shape
kappa = .2 # minimum difference needed between the prediction probability for the
- perturbed instance on the
  # class predicted by the original instance and the max probability on the
  # other classes
# in order for the first loss term to be minimized
beta = .1 # weight of the L1 loss term
beta = .1 # weight of the L1 loss term
c_init = 10. # initial weight c of the loss term encouraging to predict a different
- class (PN) or
  # the same class (PP) for the perturbed instance compared to the
- original instance to be explained
# nb of updates for c
max_iterations = 1000 # nb of iterations per value of c
feature_range = (x_train.min(axis=0).reshape(shape)-.1,
# feature range for the
  # perturbed instance
  x_train.max(axis=0).reshape(shape)+.1) # can be either a float or
  # array of shape (1xfeatures)
clip = (-1000.,1000.) # gradient clipping
lr_init = 1e-2 # initial learning rate

Generate pertinent negative:

[9]:
# define model
lr = load_model('iris_lr.h5')
# initialize CEM explainer and explain instance
cem = CEM(lr, mode, shape, kappa=kappa, beta=beta, feature_range=feature_range,
  max_iterations=max_iterations, c_init=c_init, c_steps=c_steps,
  learning_rate_init=lr_init, clip=clip)
cem.fit(x_train, no_info_type='median') # we need to define what feature values
- contain the least
  # info wrt predictions
  # here we will naively assume that the
- feature-wise median
  # contains no info; domain knowledge helps!
explanation = cem.explain(X, verbose=False)

[10]:
print(f'Original instance: {explanation.X}')
print('Predicted class: {}
'.format(class_names[explanation.X_pred]))
Original instance: [[ 0.55333328 -1.28296331 0.70592084 0.92230284]]
Predicted class: virginica

[11]:
print(f'Pertinent negative: {explanation.PN}')
print('Predicted class: {}
'.format(class_names[explanation.PN_pred]))
Pertinent negative: [[ 0.5533333 -1.2829633 -0.5391252 0.92230284]]
Predicted class: versicolor

Store explanation to plot later on:

[12]:
expl = {}
expl['PN'] = explanation.PN
expl['PN_pred'] = explanation.PN_pred

24.3. Generate contrastive explanation with pertinent negative 141
24.4 Generate pertinent positive

```python
mode = 'PP'

Generate pertinent positive:

```java
# define model
lr = load_model('iris_lr.h5')

cem = CEM(lr, mode, shape, kappa=kappa, beta=beta, feature_range=feature_range,
          max_iterations=max_iterations, c_init=c_init, c_steps=c_steps,
          learning_rate_init=lr_init, clip=clip)
cem.fit(x_train, no_info_type='median')
explanation = cem.explain(X, verbose=False)
```

```markdown
Pertinent positive: 
\[-7.4469730e-09 -3.47054341e-08 2.68840638e-01 9.17062904e-01\]

Predicted class: virginica
```

```python
expl['PP'] = explanation.PP
expl['PP_pred'] = explanation.PP_pred
```

24.5 Visualize PN and PP

Let’s visualize the generated explanations to check if the perturbed instances make sense.

Create dataframe from standardized data:

```python
df = pd.DataFrame(dataset.data, columns=dataset.feature_names)
df['species'] = np.array([dataset.target_names[i] for i in dataset.target])
```

Highlight explained instance and add pertinent negative and positive to the dataset:

```python
pn = pd.DataFrame(expl['PN'], columns=dataset.feature_names)
pn['species'] = 'PN_ ' + class_names[expl['PN_pred']]
pp = pd.DataFrame(expl['PP'], columns=dataset.feature_names)
pp['species'] = 'PP_ ' + class_names[expl['PP_pred']]
orig_inst = pd.DataFrame(explanation.X, columns=dataset.feature_names)
orig_inst['species'] = 'orig_ ' + class_names[explanation.X_pred]
```

Pair plots between the features show that the pertinent negative is pushed from the original instance (versicolor) into the virginica distribution while the pertinent positive moved away from the virginica distribution.

```python
fig = sns.pairplot(df, hue='species', diag_kind='hist');```
24.6 Use numerical gradients in CEM

If we do not have access to the Keras or TensorFlow model weights, we can use numerical gradients for the first term in the loss function that needs to be minimized (eq. 1 and 4 in the paper).

CEM parameters:

[20]: mode = 'PN'

If numerical gradients are used to compute:

\[
\frac{\partial L}{\partial x} = \frac{\partial L}{\partial p} \cdot \frac{\partial p}{\partial x}
\]

with \( L \) = loss function; \( p \) = predict function and \( x \) the parameter to optimize, then the tuple \( eps \) can be used to define the perturbation used to compute the derivatives. \( eps[0] \) is used to calculate the first partial derivative term and \( eps[1] \) is used for the second term. \( eps[0] \) and \( eps[1] \) can be a combination of float values or numpy arrays. For \( eps[0] \), the array dimension should be \((1 x nb of prediction categories)\) and for \( eps[1] \) it should be \((1 x nb of features)\).
For complex models with a high number of parameters and a high dimensional feature space (e.g. Inception on ImageNet), evaluating numerical gradients can be expensive as they involve multiple prediction calls for each perturbed instance. The `update_num_grad` parameter allows you to set a batch size on which to evaluate the numerical gradients, drastically reducing the number of prediction calls required.

```python
update_num_grad = 1
```

Generate pertinent negative:

```python
# define model
lr = load_model('iris_lr.h5')
predict_fn = lambda x: lr.predict(x)  # only pass the predict fn which takes numpy arrays to CEM
# explainer can no longer minimize wrt model weights

cem = CEM(predict_fn, mode, shape, kappa=kappa, beta=beta,
          feature_range=feature_range, max_iterations=max_iterations,
          eps=eps, c_init=c_init, c_steps=c_steps, learning_rate_init=lr_init,
          clip=clip, update_num_grad=update_num_grad)
cem.fit(x_train, no_info_type='median')
explanation = cem.explain(X, verbose=False)
```
CHAPTER TWENTYFIVE

COUNTERFACTUAL INSTANCES ON MNIST

Given a test instance $X$, this method can generate counterfactual instances $X'$ given a desired counterfactual class $t$ which can either be a class specified upfront or any other class that is different from the predicted class of $X$.

The loss function for finding counterfactuals is the following:

$$L(X'|X) = (f_t(X') - p_t)^2 + \lambda L_1(X', X).$$

The first loss term, guides the search towards instances $X'$ for which the predicted class probability $f_t(X')$ is close to a pre-specified target class probability $p_t$ (typically $p_t = 1$). The second loss term ensures that the counterfactuals are close in the feature space to the original test instance.

In this notebook we illustrate the usage of the basic counterfactual algorithm on the MNIST dataset.

```python
import tensorflow as tf
tf.get_logger().setLevel(40) # suppress deprecation messages
tf.compat.v1.disable_v2_behavior() # disable TF2 behaviour as alibi code still relies on TF1 constructs
from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D,
    Input
from tensorflow.keras.models import Model, load_model
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
from time import time
from alibi.explainers import CounterFactual
print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # False

25.1 Load and prepare MNIST data

```
Prepare data: scale, reshape and categorize

```python
[3]: x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = np.reshape(x_train, x_train.shape + (1,))
x_test = np.reshape(x_test, x_test.shape + (1,))
print('x_train shape:', x_train.shape, 'x_test shape:', x_test.shape)
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
print('y_train shape:', y_train.shape, 'y_test shape:', y_test.shape)
x_train shape: (60000, 28, 28, 1) x_test shape: (10000, 28, 28, 1)
y_train shape: (60000, 10) y_test shape: (10000, 10)
```

```python
[4]: xmin, xmax = -.5, .5
x_train = ((x_train - x_train.min()) / (x_train.max() - x_train.min())) * (xmax - xmin) + xmin
x_test = ((x_test - x_test.min()) / (x_test.max() - x_test.min())) * (xmax - xmin) + xmin
```

## 25.2 Define and train CNN model

```python
[5]: def cnn_model():
    x_in = Input(shape=(28, 28, 1))
x = Conv2D(filters=64, kernel_size=2, padding='same', activation='relu')(x_in)
x = MaxPooling2D(pool_size=2)(x)
x = Dropout(0.3)(x)

    x = Conv2D(filters=32, kernel_size=2, padding='same', activation='relu')(x)
x = MaxPooling2D(pool_size=2)(x)
x = Dropout(0.3)(x)

    x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
    x_out = Dense(10, activation='softmax')(x)
```

(continues on next page)
cnn = Model(inputs=x_in, outputs=x_out)
cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
return cnn

[6]:
cnn = cnn_model()
cnn.summary()
cnn.fit(x_train, y_train, batch_size=64, epochs=3, verbose=0)
cnn.save('mnist_cnn.h5')

Model: "model"

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_1 (InputLayer)</td>
<td>(None, 28, 28, 1)</td>
<td>0</td>
</tr>
<tr>
<td>conv2d (Conv2D)</td>
<td>(None, 28, 28, 64)</td>
<td>320</td>
</tr>
<tr>
<td>max_pooling2d (MaxPooling2D)</td>
<td>(None, 14, 14, 64)</td>
<td>0</td>
</tr>
<tr>
<td>dropout (Dropout)</td>
<td>(None, 14, 14, 64)</td>
<td>0</td>
</tr>
<tr>
<td>conv2d_1 (Conv2D)</td>
<td>(None, 14, 14, 32)</td>
<td>8224</td>
</tr>
<tr>
<td>max_pooling2d_1 (MaxPooling2)</td>
<td>(None, 7, 7, 32)</td>
<td>0</td>
</tr>
<tr>
<td>dropout_1 (Dropout)</td>
<td>(None, 7, 7, 32)</td>
<td>0</td>
</tr>
<tr>
<td>flatten (Flatten)</td>
<td>(None, 1568)</td>
<td>0</td>
</tr>
<tr>
<td>dense (Dense)</td>
<td>(None, 256)</td>
<td>401664</td>
</tr>
<tr>
<td>dropout_2 (Dropout)</td>
<td>(None, 256)</td>
<td>0</td>
</tr>
<tr>
<td>dense_1 (Dense)</td>
<td>(None, 10)</td>
<td>2570</td>
</tr>
</tbody>
</table>

Total params: 412,778
Trainable params: 412,778
Non-trainable params: 0

Evaluate the model on test set

[7]:
cnn = load_model('mnist_cnn.h5')
score = cnn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy: ', score[1])

Test accuracy: 0.9871
25.3 Generate counterfactuals

Original instance:

```
[8]: X = x_test[0].reshape((1,) + x_test[0].shape)
plt.imshow(X.reshape(28, 28));
```

Counterfactual parameters:

```
[9]: shape = (1,) + x_train.shape[1:]
target_proba = 1.0
tol = 0.01 # want counterfactuals with p(class)>0.99
target_class = 'other' # any class other than 7 will do
max_iter = 1000
lam_init = 1e-1
max_lam_steps = 10
learning_rate_init = 0.1
feature_range = (x_train.min(), x_train.max())
```

Run counterfactual:

```
[10]: # initialize explainer
cf = CounterFactual(cnn, shape=shape, target_proba=target_proba, tol=tol,
                    target_class=target_class, max_iter=max_iter, lam_init=lam_init,
                    max_lam_steps=max_lam_steps, learning_rate_init=learning_rate_init,
                    feature_range=feature_range)

start_time = time()
explanation = cf.explain(X)
print('Explanation took {:.3f} sec'.format(time() - start_time))
Explanation took 8.407 sec
```

Results:

```
[11]: pred_class = explanation.cf['class']
proba = explanation.cf['proba'][0][pred_class]
```
Counterfactual prediction: 9 with probability 0.9924006462097168

The counterfactual starting from a 7 moves towards the closest class as determined by the model and the data - in this case a 9. The evolution of the counterfactual during the iterations over $\lambda$ can be seen below (note that all of the following examples satisfy the counterfactual condition):

```
[12]: n_cfs = np.array([len(explanation.all[iter_cf]) for iter_cf in range(max_lam_steps)])
examples = {}
for ix, n in enumerate(n_cfs):
    if n>0:
        examples[ix] = {'ix': ix, 'lambda': explanation.all[ix][0]['lambda'],
                         'X': explanation.all[ix][0]['X']}
columns = len(examples) + 1
rows = 1
fig = plt.figure(figsize=(16,6))
for i, key in enumerate(examples.keys()):
    ax = plt.subplot(rows, columns, i+1)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    plt.imshow(examples[key]['X'].reshape(28,28))
    plt.title(f'Iteration: {key}')</n
Typically, the first few iterations find counterfactuals that are out of distribution, while the later iterations make the counterfactual more sparse and interpretable.

Let’s now try to steer the counterfactual to a specific class:

```
[13]: target_class = 1
```
cf = CounterFactual(cnn, shape=shape, target_proba=target_proba, tol=tol, 
                   target_class=target_class, max_iter=max_iter, lam_init=lam_init, 
                   max_lam_steps=max_lam_steps, learning_rate_init=learning_rate_init, 
                   feature_range=feature_range)

explanation = start_time = time()
explanation = cf.explain(X)
print('Explanation took {:.3f} sec'.format(time() - start_time))

Explanation took 6.249 sec

Results:

[14]: pred_class = explanation.cf['class']
    proba = explanation.cf['proba'][0][pred_class]

    print(f'Counterfactual prediction: {pred_class} with probability {proba:.6f}"
    plt.imshow(explanation.cf['X'].reshape(28, 28));

Counterfactual prediction: 1 with probability 0.9999160766601562

As you can see, by specifying a class, the search process can't go towards the closest class to the test instance (in this case a 9 as we saw previously), so the resulting counterfactual might be less interpretable. We can gain more insight by looking at the difference between the counterfactual and the original instance:

[15]: plt.imshow((explanation.cf['X'] - X).reshape(28, 28));
This shows that the counterfactual is stripping out the top part of the 7 to make to result in a prediction of 1 - not very surprising as the dataset has a lot of examples of diagonally slanted 1’s.

Clean up:

```
[16]: os.remove('mnist_cnn.h5')
```
COUNTERFACTUALS GUIDED BY PROTOTYPES ON MNIST

This method is described in the Interpretable Counterfactual Explanations Guided by Prototypes paper and can generate counterfactual instances guided by class prototypes. It means that for a certain instance $X$, the method builds a prototype for each prediction class using either an autoencoder or k-d trees. The nearest prototype class other than the originally predicted class is then used to guide the counterfactual search. For example, in MNIST the closest class to a 7 could be a 9. As a result, the prototype loss term will try to minimize the distance between the proposed counterfactual and the prototype of a 9. This speeds up the search towards a satisfactory counterfactual by steering it towards an interpretable solution from the start of the optimization. It also helps to avoid out-of-distribution counterfactuals with the perturbations driven to a prototype of another class.

The loss function to be optimized is the following:

$$\text{Loss} = cL_{\text{pred}} + \beta L_1 + L_2 + L_{AE} + L_{\text{proto}}$$

The first loss term relates to the model’s prediction function, the following 2 terms define the elastic net regularization while the last 2 terms are optional. The aim of $L_{AE}$ is to penalize out-of-distribution counterfactuals while $L_{\text{proto}}$ guides the counterfactual to a prototype. When we only have access to the model’s prediction function and cannot fully enjoy the benefits of automatic differentiation, the prototypes allow us to drop the prediction function loss term $L_{\text{pred}}$ and still generate high quality counterfactuals. This drastically reduces the number of prediction calls made during the numerical gradient update step and again speeds up the search.

Other options include generating counterfactuals for specific classes or including trust score constraints to ensure that the counterfactual is close enough to the newly predicted class compared to the original class. Different use cases are illustrated throughout this notebook.
26.1 Load and prepare MNIST data

```
[2]: (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
print('x_train shape:', x_train.shape, 'y_train shape:', y_train.shape)
plt.imshow(x_test[1]);
plt.gray()
plt.imshow(x_test[1]);
x_train shape: (60000, 28, 28) y_train shape: (60000,)
```

Prepare data: scale, reshape and categorize

```
[3]: x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = np.reshape(x_train, x_train.shape + (1,))
x_test = np.reshape(x_test, x_test.shape + (1,))
print('x_train shape:', x_train.shape, 'x_test shape:', x_test.shape)
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
print('y_train shape:', y_train.shape, 'y_test shape:', y_test.shape)
x_train shape: (60000, 28, 28, 1) x_test shape: (10000, 28, 28, 1)
y_train shape: (60000, 10) y_test shape: (10000, 10)
```

```
[4]: xmin, xmax = -.5, .5
x_train = ((x_train - x_train.min()) / (x_train.max() - x_train.min())) * (xmax - xmin) + xmin
x_test = ((x_test - x_test.min()) / (x_test.max() - x_test.min())) * (xmax - xmin) + xmin
```
26.2 Define and train CNN model

```python
[5]: def cnn_model():
    x_in = Input(shape=(28, 28, 1))
    x = Conv2D(filters=32, kernel_size=2, padding='same', activation='relu')(x_in)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)
    x = Conv2D(filters=64, kernel_size=2, padding='same', activation='relu')(x)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)
    x = Flatten()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.5)(x)
    x_out = Dense(10, activation='softmax')(x)
    cnn = Model(inputs=x_in, outputs=x_out)
    cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return cnn

[6]:
cnn = cnn_model()
cnn.fit(x_train, y_train, batch_size=32, epochs=3, verbose=0)
cnn.save('mnist_cnn.h5', save_format='h5')

Evaluate the model on test set

[7]:
cnn = load_model('mnist_cnn.h5')
score = cnn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy: ', score[1])

Test accuracy: 0.9871

26.3 Define and train auto-encoder

```python
[8]:
def ae_model():
    # encoder
    x_in = Input(shape=(28, 28, 1))
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x_in)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(pool_size=2, padding='same')(x)
    encoded = Conv2D(1, (3, 3), activation=None, padding='same')(x)
    encoder = Model(x_in, encoded)

    # decoder
    dec_in = Input(shape=(14, 14, 1))
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(dec_in)
    x = UpSampling2D((2, 2))(x)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    decoded = Conv2D(1, (3, 3), activation=None, padding='same')(x)
    decoder = Model(dec_in, decoded)

(continues on next page)
26.4 Generate counterfactual guided by the nearest class prototype

Original instance:

```python
X = x_test[0].reshape((1,) + x_test[0].shape)
plt.imshow(X.reshape(28, 28));
```
Counterfactual parameters:

```python
[12]: shape = (1,) + x_train.shape[1:]
gamma = 100.
theta = 100.
c_init = 1.
c_steps = 2
max_iterations = 1000
feature_range = (x_train.min(),x_train.max())
```

Run counterfactual:

```python
[13]: # initialize explainer, fit and generate counterfactual
cf = CounterFactualProto(cnn, shape, gamma=gamma, theta=theta,
                          ae_model=ae, enc_model=enc, max_iterations=max_iterations,
                          feature_range=feature_range, c_init=c_init, c_steps=c_steps)
start_time = time()
cf.fit(x_train)  # find class prototypes
print('Time to find prototypes each class: {:.3f} sec'.format(time() - start_time))
start_time = time()
explanation = cf.explain(X)
print('Explanation took {:.3f} sec'.format(time() - start_time))

Time to find prototypes each class: 14.580 sec
Explanation took 9.269 sec
```

Results:

```python
[14]: print('Counterfactual prediction: {0}',format(explanation.cf['class']))
print('Closest prototype class: {0}',format(explanation.id_proto))
plt.imshow(explanation.cf['X'].reshape(28, 28));
```

Counterfactual prediction: 9
Closest prototype class: 9
The counterfactual starting from a 7 moves towards its closest prototype class: a 9. The evolution of the counterfactual during the first iteration can be seen below:

```python
[15]: iter_cf = 0
    print(f'iteration c {iter Cf}
    n = len(explanation['all'][iter_cf])
    plt.figure(figsize=(20, 4))
    for i in range(n):
        ax = plt.subplot(1, n+1, i+1)
        plt.imshow(explanation['all'][iter Cf][i].reshape(28, 28))
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
    plt.show()

iteration c 0
```

Typically, the first few iterations already steer the 7 towards a 9, while the later iterations make the counterfactual more sparse.

### 26.5 Prototypes defined by the $k$ nearest encoded instances

In the above example, the class prototypes are defined by the average encoding of all instances belonging to the specific class. Instead, we can also select only the $k$ nearest encoded instances of a class to the encoded instance to be explained and use the average over those $k$ encodings as the prototype.

```python
[16]: # initialize explainer, fit and generate counterfactuals
cf = CounterFactualProto(cnn, shape, gamma=gamma, theta=theta,
    ae_model=ae, enc_model=enc, max_iterations=max_iterations,
    feature_range=feature_range, c_init=c_init, c_steps=c_steps)
    cf.fit(x_train)
    explanation_k1 = cf.explain(X, k=1, k_type='mean')
    explanation_k20 = cf.explain(X, k=20, k_type='mean')
```

Results for $k$ equals 1:
Counterfactual prediction: 9
Closest prototype class: 9

Results for \(k\) equals 20:

Counterfactual prediction: 9
Closest prototype class: 9

A lower value of \(k\) typically leads to counterfactuals that look more like the original instance and less like an average instance of the counterfactual class.
26.6 Remove the autoencoder loss term $L_{AE}$

In the previous example, we used both an autoencoder loss term to penalize a counterfactual which falls outside of the training data distribution as well as an encoder loss term to guide the counterfactual to the nearest prototype class. In the next example we get rid of the autoencoder loss term to speed up the counterfactual search and still generate decent counterfactuals:

```
[19]: # initialize explainer, fit and generate counterfactuals
cf = CounterFactualProto(cnn, shape, gamma=gamma, theta=theta,
                        enc_model=enc, max_iterations=max_iterations,
                        feature_range=feature_range, c_init=c_init, c_steps=c_steps)

cf.fit(x_train)
start_time = time()
explanation = cf.explain(X, k=1)
print('Explanation took {:.3f} sec'.format(time() - start_time))

Explanation took 6.443 sec
```

**Results:**

```
[20]: print('Counterfactual prediction: ').format(explanation.cf['class']))
    print('Closest prototype class: ' .format(explanation.id_proto))
    plt.imshow(explanation.cf['X'].reshape(28, 28));

Counterfactual prediction: 9
Closest prototype class: 9
```

![Counterfactual prediction image](image)

26.7 Specify prototype classes

For multi-class predictions, we might be interested to generate counterfactuals for certain classes while avoiding others. The following example illustrates how to do this:

```
[21]: X = x_test[12].reshape((1,) + x_test[1].shape)
    plt.imshow(X.reshape(28, 28));
```

```
```
```
# initialize explainer, fit and generate counterfactuals

cf = CounterFactualProto(cnn, shape, gamma=gamma, theta=theta,
    ae_model=ae, enc_model=enc, max_iterations=max_iterations,
    feature_range=feature_range, c_init=c_init, c_steps=c_steps)

cf.fit(x_train)
explanation_1 = cf.explain(X, k=5, k_type='mean')
proto_1 = explanation_1.id_proto
explanation_2 = cf.explain(X, k=5, k_type='mean', target_class=[7])
proto_2 = explanation_2.id_proto

The closest class to the 9 is 4. This is evident by looking at the first counterfactual below. For the second counterfactual, we specified that the prototype class used in the search should be a 7. As a result, a counterfactual 7 instead of a 4 is generated.

Counterfactual prediction: 4
Closest prototype class: 4
26.8 Speed up the counterfactual search by removing the predict function loss loss term

We can also remove the prediction loss term and still obtain an interpretable counterfactual. This is especially relevant for fully black box models. When we provide the counterfactual search method with a Keras or TensorFlow model, it is incorporated in the TensorFlow graph and evaluated using automatic differentiation. However, if we only have access to the model’s prediction function, the gradient updates are numerical and typically require a large number of prediction calls because of the prediction loss term $L_{pred}$. These prediction calls can slow the search down significantly and become a bottleneck. We can represent the gradient of the loss term as follows:

$$\frac{\partial L_{pred}}{\partial x} = \frac{\partial L_{pred}}{\partial p} \frac{\partial p}{\partial x}$$

where $L_{pred}$ is the prediction loss term, $p$ the prediction function and $x$ the input features to optimize. For a 28 by 28 MNIST image, the $\frac{\partial p}{\partial x}$ term alone would require a prediction call with batch size $28 \times 28 \times 2 = 1568$. By using the prototypes to guide the search however, we can remove the prediction loss term and only make a single prediction at the end of each gradient update to check whether the predicted class on the proposed counterfactual is different from the original class. We do not necessarily need a Keras or TensorFlow auto-encoder either and can use k-d trees to find the nearest class prototypes. Please check out this notebook for a practical example.

The first example below removes $L_{pred}$ from the loss function to bypass the bottleneck. It illustrates the drastic speed improvements over the black box alternative with numerical gradient evaluation while still producing interpretable counterfactual instances.
# weight on prediction loss term set to 0

c_init = 0.

c_steps = 1

# no need to find optimal values for c

# define a black-box model
predict_fn = lambda x: cnn.predict(x)

# initialize explainer, fit and generate counterfactuals
cf = CounterFactualProto(predict_fn, shape, gamma=gamma, theta=theta,
                         ae_model=ae, enc_model=enc, max_iterations=max_iterations,
                         feature_range=feature_range, c_init=c_init, c_steps=c_steps)

cf.fit(x_train)

start_time = time()
explanation = cf.explain(X, k=1)

print('Explanation took {:.3f} sec'.format(time() - start_time))

Explanation took 7.257 sec

print('Counterfactual prediction: {}'.format(explanation.cf['class']))
print('Closest prototype class: {}'.format(explanation.id_proto))

plt.imshow(explanation.cf['X'].reshape(28, 28));

Counterfactual prediction: 6
Closest prototype class: 6
Let us know add the $L_{pred}$ loss term back in the objective function and observe how long it takes to generate a black box counterfactual:

[29]:
```python
c_init = 1.
c_steps = 2
```

[30]:
```python
# define a black-box model
predict_fn = lambda x: cnn.predict(x)

# initialize explainer, fit and generate counterfactuals
cf = CounterFactualProto(predict_fn, shape, gamma=gamma, theta=theta,
                           ae_model=ae, enc_model=enc, max_iterations=max_iterations,
                           feature_range=feature_range, c_init=c_init, c_steps=c_steps)

cf.fit(x_train)
start_time = time()
explanation = cf.explain(X, k=1)
print('Explanation took {:.3f} sec'.format(time() - start_time))
```
Explanation took 966.342 sec

[31]:
```python
print('Counterfactual prediction: {}, format(explanation.cf['class'])
print(f'Closest prototype class: {explanation.id_proto')
plt.imshow(explanation.cf['X'].reshape(28, 28))
```
Counterfactual prediction: 6
Closest prototype class: 6
Clean up:

```python
os.remove('mnist_cnn.h5')
os.remove('mnist_ae.h5')
os.remove('mnist_enc.h5')
```
CHAPTER TWENTYSEVEN

COUNTERFACTUALS GUIDED BY PROTOTYPES ON BOSTON HOUSING DATASET

This notebook goes through an example of prototypical counterfactuals using k-d trees to build the prototypes. Please check out this notebook for a more in-depth application of the method on MNIST using (auto-)encoders and trust scores.

In this example, we will train a simple neural net to predict whether house prices in the Boston area are above the median value or not. We can then find a counterfactual to see which variables need to be changed to increase or decrease a house price above or below the median value.

```python
import tensorflow as tf
# suppress deprecation messages
# disable TF2 behaviour as alibi code still relies on TF1 constructs
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.utils import to_categorical
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
from sklearn.datasets import load_boston
from alibi.explainers import CounterFactualProto

print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # False

TF version:  2.2.0
Eager execution enabled:  False
```

### 27.1 Load and prepare Boston housing dataset

```python
boston = load_boston()
data = boston.data
target = boston.target
feature_names = boston.feature_names

Transform into classification task: target becomes whether house price is above the overall median or not

```python
y = np.zeros((target.shape[0],))
y[np.where(target > np.median(target))[0]] = 1
```
Remove categorical feature

```python
[4]:
data = np.delete(data, 3, 1)
feature_names = np.delete(feature_names, 3)
```

Explanation of remaining features:

- CRIM: per capita crime rate by town
- ZN: proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS: proportion of non-retail business acres per town
- RM: average number of rooms per dwelling
- AGE: proportion of owner-occupied units built prior to 1940
- DIS: weighted distances to five Boston employment centres
- RAD: index of accessibility to radial highways
- TAX: full-value property-tax rate per USD10,000
- PTRATIO: pupil-teacher ratio by town
- B: 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT: % lower status of the population

Standardize data

```python
[5]:
mu = data.mean(axis=0)
sigma = data.std(axis=0)
data = (data - mu) / sigma
```

Define train and test set

```python
[6]:
idx = 475
x_train, y_train = data[:idx, :], y[:idx]
x_test, y_test = data[idx:, :], y[idx:]
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

### 27.2 Train model

```python
[7]:
np.random.seed(42)
tf.random.set_seed(42)
```

```python
[8]:
def nn_model():
    x_in = Input(shape=(12,))
x = Dense(40, activation='relu')(x_in)
x = Dense(40, activation='relu')(x)
x_out = Dense(2, activation='softmax')(x)
nn = Model(inputs=x_in, outputs=x_out)
nn.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
    return nn
```
27.3 Generate counterfactual guided by the nearest class prototype

Original instance:

[11]: X = x_test[1].reshape((1,) + x_test[1].shape)
    shape = X.shape

Run counterfactual:

[12]: # define model
    nn = load_model('nn_boston.h5')

    # initialize explainer, fit and generate counterfactual
    cf = CounterFactualProto(nn, shape, use_kdtree=True, theta=10., max_iterations=1000,
                             feature_range=(x_train.min(axis=0), x_train.max(axis=0)),
                             c_init=1., c_steps=10)

    cf.fit(x_train)
    explanation = cf.explain(X)

The prediction flipped from 0 (value below the median) to 1 (above the median):

[13]: print(f'Original prediction: {explanation.orig_class}')</n    print(f'Counterfactual prediction: {explanation.cf['class']}.format(explanation.cf['class']))
Let’s take a look at the counterfactual. To make the results more interpretable, we will first undo the pre-processing step and then check where the counterfactual differs from the original instance:

```python
[14]: orig = X * sigma + mu
counterfactual = explanation.cf['X'] * sigma + mu
delta = counterfactual - orig
for i, f in enumerate(feature_names):
    if np.abs(delta[0][i]) > 1e-4:
        print('{}: {}'.format(f, delta[0][i]))

AGE: -6.526239960695747
LSTAT: -4.799340540220259
```

So in order to increase the house price, the proportion of owner-occupied units built prior to 1940 should decrease by ~11-12%. This is not surprising since the proportion for the observation is very high at 93.6%. Furthermore, the % of the population with “lower status” should decrease by ~5%.

```python
[15]: print('% owner-occupied units built prior to 1940: {}'.format(orig[0][5]))
    print('% lower status of the population: {}'.format(orig[0][11]))

% owner-occupied units built prior to 1940: 93.6
% lower status of the population: 18.68
```

Clean up:

```python
[16]: os.remove('nn_boston.h5')
```
Real world machine learning applications often handle data with categorical variables. Explanation methods which rely on perturbations of the input features need to make sure those perturbations are meaningful and capture the underlying structure of the data. This becomes tricky for categorical features. For instance random perturbations across possible categories or enforcing a ranking between categories based on frequency of occurrence in the training data do not capture this structure. Our method captures the relation between categories of a variable numerically through the context given by the other features in the data and/or the predictions made by the model. First it captures the pairwise distances between categories and then applies multi-dimensional scaling. More details about the method can be found in the documentation. The example notebook illustrates this approach on the adult dataset, which contains a mixture of categorical and numerical features used to predict whether a person’s income is above or below $50k.

```
import tensorflow as tf
tf.get_logger().setLevel(40)  # suppress deprecation messages
tf.compat.v1.disable_v2_behavior()  # disable TF2 behaviour as alibi code still relies on TF1 constructs
from tensorflow.keras.layers import Dense, Dropout, Input
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical

%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import os
from sklearn.preprocessing import OneHotEncoder
from time import time
from alibi.datasets import fetch_adult
from alibi.explainers import CounterFactualProto
from alibi.utils.mapping import ohe_to_ord, ord_to_ohe

print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly())  # False
```

TF version: 2.2.0
Eager execution enabled: False
28.1 Load adult dataset

The `fetch_adult` function returns a `Bunch` object containing the features, the targets, the feature names and a mapping of the categories in each categorical variable.

```python
adult = fetch_adult()
data = adult.data
target = adult.target
feature_names = adult.feature_names
category_map_tmp = adult.category_map
target_names = adult.target_names
```

Define shuffled training and test set:

```python
def set_seed(s=0):
    np.random.seed(s)
    tf.random.set_seed(s)
set_seed()
data_perm = np.random.permutation(np.c_[data, target])
X = data_perm[:,:-1]
y = data_perm[:,-1]
idx = 30000
y_train, y_test = y[:idx], y[idx+1:]
```

Reorganize data so categorical features come first:

```python
X = np.c_[X[:, 1:8], X[:, 11], X[:, 0], X[:, 8:11]]
```

Adjust `feature_names` and `category_map` as well:

```python
                feature_names[8:11]
print(feature_names)
['Workclass', 'Education', 'Marital Status', 'Occupation', 'Relationship', 'Race', 
 'Sex', 'Country', 'Age', 'Capital Gain', 'Capital Loss', 'Hours per week']
```

```python
category_map = {}
for i, (_, v) in enumerate(category_map_tmp.items()):
    category_map[i] = v
```

Create a dictionary with as keys the categorical columns and values the number of categories for each variable in the dataset:

```python
cat_vars_ord = {}
n_categories = len(list(category_map.keys()))
for i in range(n_categories):
    cat_vars_ord[i] = len(np.unique(X[:, i]))
print(cat_vars_ord)
{0: 9, 1: 7, 2: 4, 3: 9, 4: 6, 5: 5, 6: 2, 7: 11}
```

Since we will apply one-hot encoding (OHE) on the categorical variables, we convert `cat_vars_ord` from the ordinal to OHE format. `alibi.utils.mapping` contains utility functions to do this. The keys in `cat_vars_ord`
now represent the first column index for each one-hot encoded categorical variable. This dictionary will later be used in the counterfactual explanation.

```python
[10]: cat_vars_ohe = ord_to_ohe(X, cat_vars_ord)
    print(cat_vars_ohe)
{0: 9, 9: 7, 16: 4, 20: 9, 29: 6, 35: 5, 40: 2, 42: 11}
```

### 28.2 Preprocess data

Scale numerical features between -1 and 1:

```python
[11]: X_num = X[:, -4:].astype(np.float32, copy=False)
xmin, xmax = X_num.min(axis=0), X_num.max(axis=0)
rng = (-1., 1.)
X_num_scaled = (X_num - xmin) / (xmax - xmin) * (rng[1] - rng[0]) + rng[0]
X_num_scaled_train = X_num_scaled[:idx, :]
X_num_scaled_test = X_num_scaled[idx+1:, :]
```

Apply OHE to categorical variables:

```python
[12]: X_cat = X[:, :-4].copy()
ohe = OneHotEncoder(categories='auto')
ohe.fit(X_cat)
X_cat_ohe = ohe.transform(X_cat)
```

Combine numerical and categorical data:

```python
[13]: X = np.c_[X_cat_ohe.todense(), X_num_scaled].astype(np.float32, copy=False)
X_train, X_test = X[:idx, :], X[idx+1:, :]
print(X_train.shape, X_test.shape)
(30000, 57) (2560, 57)
```

### 28.3 Train neural net

```python
[14]: def nn_ohe():
    x_in = Input(shape=(57,))
x = Dense(60, activation='relu')(x_in)
x = Dropout(.2)(x)
x = Dense(60, activation='relu')(x)
x = Dropout(.2)(x)
x = Dense(60, activation='relu')(x)
x = Dropout(.2)(x)
x_out = Dense(2, activation='softmax')(x)

    nn = Model(inputs=x_in, outputs=x_out)
nn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

    return nn
```
```python
[15]:
set_seed()
nn = nn_ohe()
nn.summary()
nn.fit(X_train, to_categorical(y_train), batch_size=256, epochs=30, verbose=0)
```

Model: "model"

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_1 (InputLayer)</td>
<td>[None, 57]</td>
<td>0</td>
</tr>
<tr>
<td>dense (Dense)</td>
<td>(None, 60)</td>
<td>3480</td>
</tr>
<tr>
<td>dropout (Dropout)</td>
<td>(None, 60)</td>
<td>0</td>
</tr>
<tr>
<td>dense_1 (Dense)</td>
<td>(None, 60)</td>
<td>3660</td>
</tr>
<tr>
<td>dropout_1 (Dropout)</td>
<td>(None, 60)</td>
<td>0</td>
</tr>
<tr>
<td>dense_2 (Dense)</td>
<td>(None, 60)</td>
<td>3660</td>
</tr>
<tr>
<td>dropout_2 (Dropout)</td>
<td>(None, 60)</td>
<td>0</td>
</tr>
<tr>
<td>dense_3 (Dense)</td>
<td>(None, 2)</td>
<td>122</td>
</tr>
</tbody>
</table>

Total params: 10,922
Trainable params: 10,922
Non-trainable params: 0

```python
[15]: <tensorflow.python.keras.callbacks.History at 0x7f2d7cc5e410>
```

## 28.4 Generate counterfactual

Original instance:

```python
[16]:
X = X_test[0].reshape((1,) + X_test[0].shape)
```

Initialize counterfactual parameters. The feature perturbations are applied in the numerical feature space, after transforming the categorical variables to numerical features. As a result, the dimensionality and values of `feature_range` are defined in the numerical space.

```python
[17]:
shape = X.shape
beta = .01
c_init = 1.
c_steps = 5
max_iterations = 500
rng = (-1., 1.)  # scale features between -1 and 1
rng_shape = (1,) + data.shape[1:]
feature_range = ((np.ones(rng_shape) * rng[0]).astype(np.float32),
                 (np.ones(rng_shape) * rng[1]).astype(np.float32))
```

Initialize explainer:
```python
[18]: def set_seed(s=0):
    np.random.seed(s)
    tf.random.set_seed(s)

[19]: set_seed()
    cf = CounterFactualProto(nn,
        shape,
        beta=beta,
        cat_vars=cat_vars_ohe,
        ohe=True,  # OHE flag
        max_iterations=max_iterations,
        feature_range=feature_range,
        c_init=c_init,
        c_steps=c_steps
    )
```

Fit explainer. `d_type` refers to the distance metric used to convert the categorical to numerical values. Valid options are `abdm`, `mvdm` and `abdm-mvdm`. `abdm` infers the distance between categories of the same variable from the context provided by the other variables. This requires binning of the numerical features as well. `mvdm` computes the distance using the model predictions, and `abdm-mvdm` combines both methods. More info on both distance measures can be found in the documentation.

```python
[20]: cf.fit(X_train, d_type='abdm', disc_perc=[25, 50, 75]);
```

We can now visualize the transformation from the categorical to numerical values for each category. The example below shows that the **Education** feature is ordered from **High School Dropout** to having obtained a **Doctorate** degree. As a result, if we perturb an instance representing a person that has obtained a **Bachelors** degree, the nearest perturbations will result in a counterfactual instance with either a **Masters** or an **Associates** degree.

```python
[21]: def plot_bar(dist, cols, figsize=(10,4)):
    dist = dist.reshape(dist.shape[0])
    idx = np.argsort(dist)
    fig, ax = plt.subplots(figsize=figsize)
    plt.bar(cols[idx], dist[idx])
    print(cols[idx])

[22]: cat = 'Education'
    idx = feature_names.index(cat)
    plot_bar(cf.d_abs[idx], np.array(category_map[idx]), figsize=(20,4))

['Dropout' 'High School grad' 'Associates' 'Bachelors' 'Masters' 'Prof-School' 'Doctorate']
```

Explain instance:

```python
[23]: explanation = cf.explain(X)
```

Helper function to more clearly describe explanations:
def describe_instance(X, explanation, eps=1e-2):
    print('Original instance: {} -- proba: {}'.format(target_names[explanation.orig_→class], explanation.orig_proba[0]))
    print('Counterfactual instance: {} -- proba: {}'.format(target_names[explanation.cf['class']], explanation.cf['proba']))
    print('Counterfactual perturbations...')
    print('Categorical:')
    X_orig_ord = ohe_to_ord(X, cat_vars_ohe)[0]
    X_cf_ord = ohe_to_ord(explanation.cf['X'], cat_vars_ohe)[0]
    delta_cat = {}
    for i, (_, v) in enumerate(category_map.items()):
        cat_orig = v[int(X_orig_ord[0, i])]
        cat_cf = v[int(X_cf_ord[0, i])]
        if cat_orig != cat_cf:
            delta_cat[feature_names[i]] = [cat_orig, cat_cf]
    if delta_cat:
        for k, v in delta_cat.items():
            print('{}: {} --> {}'.format(k, v[0], v[1]))
    print('Numerical:')
    delta_num = X_cf_ord[0, -4:] - X_orig_ord[0, -4:]
    n_keys = len(list(cat_vars_ord.keys()))
    for i in range(delta_num.shape[1]):
        if np.abs(delta_num[0, i]) > eps:
            print('{}: {:.2f} --> {:.2f}'.format(feature_names[i+n_keys], X_orig_ord[0,i+n_keys], X_cf_ord[0,i+n_keys]))

describe_instance(X, explanation)

By obtaining a higher level of education the income is predicted to be above $50k.

### 28.5 Change the categorical distance metric

Instead of abdm, we now use mvdm as our distance metric.

set_seed()

set_seed()

cf.fit(X_train, d_type='mvdm')

explanation = cf.explain(X)

describe_instance(X, explanation)

Original instance: <=50K -- proba: [0.70744723 0.29255277]
Counterfactual instance: >50K -- proba: [0.37736374 0.62263626]

Counterfactual perturbations...

Categorical:

Education: Associates --> Bachelors

Numerical:

By obtaining a higher level of education the income is predicted to be above $50k.

### 28.5 Change the categorical distance metric

Instead of abdm, we now use mvdm as our distance metric.
Counterfactual perturbations...

Categorical:
Education: Associates --> Bachelors

Numerical:

The same conclusion hold using a different distance metric.

## 28.6 Use k-d trees to build prototypes

We can also use k-d trees to build class prototypes to guide the counterfactual to nearby instances in the counterfactual class as described in Interpretable Counterfactual Explanations Guided by Prototypes.

```
[27]: use_kdtree = True
    theta = 10.  # weight of prototype loss term

Initialize, fit and explain instance:
```

```
[28]: set_seed()
    X = X_test[7].reshape((1,) + X_test[0].shape)
    cf = CounterFactualProto(nn,
        shape,
        beta=beta,
        theta=theta,
        cat_vars=cat_vars_ohe,
        ohe=True,
        use_kdtree=use_kdtree,
        max_iterations=max_iterations,
        feature_range=feature_range,
        c_init=c_init,
        c_steps=c_steps
    
    cf.fit(X_train, d_type='abdm')
    explanation = cf.explain(X)
    describe_instance(X, explanation)
```

Original instance: <=50K -- proba: [0.5211548 0.47884512]
Counterfactual instance: >50K -- proba: [0.49958408 0.500416 ]

Counterfactual perturbations...

Categorical:

Numerical:
Age: -0.53 --&gt; -0.51

By slightly increasing the age of the person the income would be predicted to be above $50k.
28.7 Use an autoencoder to build prototypes

Another option is to use an autoencoder to guide the perturbed instance to the counterfactual class. We define and train the autoencoder:

```python
[29]: def ae_model():
    # encoder
    x_in = Input(shape=(57,))
    x = Dense(60, activation='relu')(x_in)
    x = Dense(30, activation='relu')(x)
    x = Dense(15, activation='relu')(x)
    encoded = Dense(10, activation=None)(x)
    encoder = Model(x_in, encoded)

    # decoder
    dec_in = Input(shape=(10,))
    x = Dense(15, activation='relu')(dec_in)
    x = Dense(30, activation='relu')(x)
    x = Dense(60, activation='relu')(x)
    decoded = Dense(57, activation=None)(x)
    decoder = Model(dec_in, decoded)

    # autoencoder = encoder + decoder
    x_out = decoder(encoder(x_in))
    autoencoder = Model(x_in, x_out)
    autoencoder.compile(optimizer='adam', loss='mse')
    return autoencoder, encoder, decoder

[30]: set_seed()
ae, enc, dec = ae_model()
ae.summary()
ae.fit(X_train, X_train, batch_size=128, epochs=100, validation_data=(X_test, X_test), verbose=0)

Model: "model_3"

Layer (type)                  Output Shape    Param #   
=================================================================
input_2 (InputLayer)         [(None, 57)]    0
model_1 (Model)              (None, 10)      5935
model_2 (Model)              (None, 57)      5982
=================================================================
Total params: 11,917
Trainable params: 11,917
Non-trainable params: 0

[30]: <tensorflow.python.keras.callbacks.History at 0x7f2d783aff90>

Weights for the autoencoder and prototype loss terms:

```python
[31]: beta = .1  # L1
gamma = 10.  # autoencoder
theta = .1   # prototype
```
Initialize, fit and explain instance:

```
[32]: set_seed()
X = X_test[19].reshape((1,) + X_test[0].shape)
cf = CounterFactualProto(nn,
    shape,
    beta=beta,
    enc_model=enc,
    ae_model=ae,
    gamma=gamma,
    theta=theta,
    cat_vars=cat_vars_ohe,
    ohe=True,
    max_iterations=max_iterations,
    feature_range=feature_range,
    c_init=c_init,
    c_steps=c_steps
)
```

```
cf.fit(X_train, d_type='abdm')
explanation = cf.explain(X)
describe_instance(X, explanation)
```

Original instance: >50K -- proba: [0.48656026 0.5134398 ]
Counterfactual instance: <=50K -- proba: [0.71456206 0.28543794]

Counterfactual perturbations...

Categorical:
Education: High School grad --> Dropout

Numerical:

### 28.8 Black box model with k-d trees

Now we assume that we only have access to the model’s prediction function and treat it as a black box. The k-d trees are again used to define the prototypes.

```
[33]: use_kdtree = True
theta = 10.  # weight of prototype loss term
```

Initialize, fit and explain instance:

```
[34]: set_seed()
X = X_test[24].reshape((1,) + X_test[0].shape)

# define predict function
predict_fn = lambda x: nn.predict(x)
```

```
cf = CounterFactualProto(predict_fn,
    shape,
    beta=beta,
    theta=theta,
    cat_vars=cat_vars_ohe,
    ohe=True,
    use_kdtree=use_kdtree,
```

(continues on next page)
max_iterations=max_iterations,
feature_range=feature_range,
c_init=c_init,
c_steps=c_steps
)

cf.fit(X_train, d_type='abdm')
explanation = cf.explain(X)
describe_instance(X, explanation)

Original instance: >50K -- proba: [0.20676644 0.79323356]  
Counterfactual instance: <=50K -- proba: [0.50484160 0.49515834]

Counterfactual perturbations...

Categorical:

Numerical:
Age: -0.15 --> -0.19 
Hours per week: -0.20 --> -0.51

If the person was younger and worked less, he or she would have a predicted income below $50k.
COUNTERFACTUAL EXPLANATIONS WITH ORDINALLY ENCODED CATEGORICAL VARIABLES

This example notebook illustrates how to obtain counterfactual explanations for instances with a mixture of ordinally encoded categorical and numerical variables. A more elaborate notebook highlighting additional functionality can be found here. We generate counterfactuals for instances in the adult dataset where we predict whether a person’s income is above or below $50k.

```python
[1]: import tensorflow as tf
tf.get_logger().setLevel(40)  # suppress deprecation messages
tf.compat.v1.disable_v2_behavior()  # disable TF2 behaviour as alibi code still relies on TF1 constructs
from tensorflow.keras.layers import Dense, Input, Embedding, Concatenate, Reshape, Dropout, Lambda
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import os
from sklearn.preprocessing import OneHotEncoder
from time import time
from alibi.datasets import fetch_adult
from alibi.explainers import CounterFactualProto

print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly())  # False
TF version:  2.2.0
Eager execution enabled:  False
```

### 29.1 Load adult dataset

The `fetch_adult` function returns a Bunch object containing the features, the targets, the feature names and a mapping of the categories in each categorical variable.

```python
[2]: adult = fetch_adult()
data = adult.data
target = adult.target
feature_names = adult.feature_names
```
Define shuffled training and test set:

```python
[3]: `def set_seed(s=0):
    np.random.seed(s)
    tf.random.set_seed(s)
```

```python
[4]: set_seed()

data_perm = np.random.permutation(np.c_[data, target])
X = data_perm[:,:-1]
y = data_perm[:, -1]
```

```python
[5]: idx = 30000
y_train, y_test = y[:idx], y[idx+1:]
```

Reorganize data so categorical features come first:

```python
[6]: X = np.c_[X[:, 1:8], X[:, 11], X[:, 0], X[:, 8:11]]
```

Adjust `feature_names` and `category_map` as well:

```python
      feature_names[8:11]
print(feature_names)

['Workclass', 'Education', 'Marital Status', 'Occupation', 'Relationship', 'Race',
 'Sex', 'Country', 'Age', 'Capital Gain', 'Capital Loss', 'Hours per week']
```

```python
[8]: category_map = {}
    for i, (_, v) in enumerate(category_map_tmp.items()):
        category_map[i] = v
```

Create a dictionary with as keys the categorical columns and values the number of categories for each variable in the dataset. This dictionary will later be used in the counterfactual explanation.

```python
[9]: cat_vars_ord = {}
    n_categories = len(list(category_map.keys()))
    for i in range(n_categories):
        cat_vars_ord[i] = len(np.unique(X[:, i]))
print(cat_vars_ord)

{0: 9, 1: 7, 2: 4, 3: 9, 4: 6, 5: 5, 6: 2, 7: 11}
```

## 29.2 Preprocess data

Scale numerical features between -1 and 1:

```python
[10]: X_num = X[:, :-4].astype(np.float32, copy=False)
xmin, xmax = X_num.min(axis=0), X_num.max(axis=0)
rng = (-1., 1.)
X_num_scaled = (X_num - xmin) / (xmax - xmin) * (rng[1] - rng[0]) + rng[0]
```
Combine numerical and categorical data:

```python
[11]: X = np.c_[X[:, :-4], X_num_scaled].astype(np.float32, copy=False)
X_train, X_test = X[:idx, :], X[idx+1:, :]
print(X_train.shape, X_test.shape)
(30000, 12) (2560, 12)
```

### 29.3 Train a neural net

The neural net will use entity embeddings for the categorical variables.

```python
[12]: def nn_ord():
    x_in = Input(shape=(12,))
layers_in = []

    # embedding layers
    for i, (_, v) in enumerate(cat_vars_ord.items()):
        emb_in = Lambda(lambda x: x[:, i:i+1])(x_in)
        emb_dim = int(max(min(np.ceil(0.5 * v), 50), 2))
        emb_layer = Embedding(input_dim=v+1, output_dim=emb_dim, input_length=1)(emb_in)
        emb_layer = Reshape(target_shape=(emb_dim,))(emb_layer)
layers_in.append(emb_layer)

    # numerical layers
    num_in = Lambda(lambda x: x[:, -4:])(x_in)
    num_layer = Dense(16)(num_in)
layers_in.append(num_layer)

    # combine
    x = Concatenate()(layers_in)
    x = Dense(60, activation='relu')(x)
    x = Dropout(.2)(x)
    x = Dense(60, activation='relu')(x)
    x = Dropout(.2)(x)
    x = Dense(60, activation='relu')(x)
    x = Dropout(.2)(x)
    x_out = Dense(2, activation='softmax')(x)
    nn = Model(inputs=x_in, outputs=x_out)
    nn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return nn

[13]: set_seed()
nn = nn_ord()
nn.summary()
nn.fit(X_train, to_categorical(y_train), batch_size=128, epochs=30, verbose=0)
```
Model: "model"

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
<th>Connected to</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_1 (InputLayer)</td>
<td>(None, 12)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>lambda (Lambda)</td>
<td>(None, 1)</td>
<td>0</td>
<td>input_1[0][0]</td>
</tr>
<tr>
<td>lambda_1 (Lambda)</td>
<td>(None, 1)</td>
<td>0</td>
<td>input_1[0][0]</td>
</tr>
<tr>
<td>lambda_2 (Lambda)</td>
<td>(None, 1)</td>
<td>0</td>
<td>input_1[0][0]</td>
</tr>
<tr>
<td>lambda_3 (Lambda)</td>
<td>(None, 1)</td>
<td>0</td>
<td>input_1[0][0]</td>
</tr>
<tr>
<td>lambda_4 (Lambda)</td>
<td>(None, 1)</td>
<td>0</td>
<td>input_1[0][0]</td>
</tr>
<tr>
<td>lambda_5 (Lambda)</td>
<td>(None, 1)</td>
<td>0</td>
<td>input_1[0][0]</td>
</tr>
<tr>
<td>lambda_6 (Lambda)</td>
<td>(None, 1)</td>
<td>0</td>
<td>input_1[0][0]</td>
</tr>
<tr>
<td>lambda_7 (Lambda)</td>
<td>(None, 1)</td>
<td>0</td>
<td>input_1[0][0]</td>
</tr>
<tr>
<td>embedding (Embedding)</td>
<td>(None, 1, 5)</td>
<td>50</td>
<td>lambda[0][0]</td>
</tr>
<tr>
<td>embedding_1 (Embedding)</td>
<td>(None, 1, 4)</td>
<td>32</td>
<td>lambda_1[0][0]</td>
</tr>
<tr>
<td>embedding_2 (Embedding)</td>
<td>(None, 1, 2)</td>
<td>10</td>
<td>lambda_2[0][0]</td>
</tr>
<tr>
<td>embedding_3 (Embedding)</td>
<td>(None, 1, 5)</td>
<td>50</td>
<td>lambda_3[0][0]</td>
</tr>
<tr>
<td>embedding_4 (Embedding)</td>
<td>(None, 1, 3)</td>
<td>21</td>
<td>lambda_4[0][0]</td>
</tr>
<tr>
<td>embedding_5 (Embedding)</td>
<td>(None, 1, 3)</td>
<td>18</td>
<td>lambda_5[0][0]</td>
</tr>
<tr>
<td>embedding_6 (Embedding)</td>
<td>(None, 1, 2)</td>
<td>6</td>
<td>lambda_6[0][0]</td>
</tr>
<tr>
<td>embedding_7 (Embedding)</td>
<td>(None, 1, 6)</td>
<td>72</td>
<td>lambda_7[0][0]</td>
</tr>
<tr>
<td>lambda_8 (Lambda)</td>
<td>(None, 4)</td>
<td>0</td>
<td>input_1[0][0]</td>
</tr>
</tbody>
</table>

(continues on next page)
(continued from previous page)

```plaintext
29.3. Train a neural net

(continues on next page)
```
29.4 Generate counterfactual

Original instance:

```python
X = X_test[0].reshape((1,) + X_test[0].shape)
```

Initialize counterfactual parameters:

```python
shape = X.shape
beta = .01
c_init = 1.
c_steps = 5
max_iterations = 500
rng = (-1., 1.)  # scale features between -1 and 1
rng_shape = (1,) + data.shape[1:]
feature_range = ((np.ones(rng_shape) * rng[0]).astype(np.float32),
                 (np.ones(rng_shape) * rng[1]).astype(np.float32))
```

Initialize explainer. Since the Embedding layers in `tf.keras` do not let gradients propagate through, we will only make use of the model’s predict function, treat it as a black box and perform numerical gradient calculations.

```python
set_seed()

# define predict function
predict_fn = lambda x: nn.predict(x)

cf = CounterFactualProto(predict_fn,
                         shape,
                         beta=beta,
                         cat_vars=cat_vars_ord,
                         max_iterations=max_iterations,
                         feature_range=feature_range,
                         c_init=c_init,
                         c_steps=c_steps,
                         eps=(.01, .01))  # perturbation size for numerical gradients
```

Fit explainer. Please check the documentation for more info about the optional arguments.

```python
cf.fit(X_train, d_type='abdm', disc_perc=[25, 50, 75]);
```

Explain instance:

```python
set_seed()

explanation = cf.explain(X)
```
Helper function to more clearly describe explanations:

```python
[19]: def describe_instance(X, explanation, eps=1e-2):
    print('Original instance: /{} -- proba: /{}'\n          .format(target_names[explanation.orig_class],
                   explanation.orig_proba[0]))
    print('Counterfactual instance: /{} -- proba: /{}'\n          .format(target_names[explanation.cf['class']],
                   explanation.cf['proba'][0]))

    print('nCounterfactual perturbations...')
    print('nCategorical: ')
    X_orig_ord = X
    X_cf_ord = explanation.cf['X']
    delta_cat = {}  
    for i, (_, v) in enumerate(category_map.items()):
        cat_orig = v[int(X_orig_ord[0, i])]
        cat_cf = v[int(X_cf_ord[0, i])]
        if cat_orig != cat_cf:
            delta_cat[feature_names[i]] = [cat_orig, cat_cf]
            if delta_cat:
                for k, v in delta_cat.items():
                    print('{}: {}/ --> {}/'.format(k, v[0], v[1]))
    print('nNumerical: ')
    delta_num = X_cf_ord[0, -4:] - X_orig_ord[0, -4:]
    n_keys = len(list(cat_vars_ord.keys()))
    for i in range(delta_num.shape[0]):
        if np.abs(delta_num[i]) > eps:
            print('{}: {:.2f} --> {:.2f}'.format(feature_names[i+n_keys],
                                             X_orig_ord[0,i+n_keys],
                                             X_cf_ord[0,i+n_keys]))

[20]: describe_instance(X, explanation)

Original instance: <=50K -- proba: [0.6976237 0.30237624]  
Counterfactual instance: >50K -- proba: [0.49604183 0.5039582 ] 

Counterfactual perturbations...

Categorical:

Numerical:

Capital Gain: -1.00 --> -0.88

The person’s incomce is predicted to be above $50k by increasing his or her capital gain.

29.4. Generate counterfactual
CHAPTER THIRTY

KERNEL SHAP EXPLANATION FOR SVM MODELS

30.1 Introduction

In this example, we show how to explain a multi-class classification model based on the SVM algorithm using the KernelSHAP method. We show how to perform instance-level (or local) explanations on this model as well as how to draw insights about the model behaviour in general by aggregating information from explanations across many instances (that is, perform global explanations).

```python
import shap
shap.initjs()
import matplotlib.pyplot as plt
import numpy as np
from alibi.explainers import KernelShap
from sklearn import svm
from sklearn.datasets import load_wine
from sklearn.metrics import confusion_matrix, plot_confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
<IPython.core.display.HTML object>
```

30.2 Data preparation

```python
wine = load_wine()
wine.keys()
```

```python
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names'])
```

```python
data = wine.data
target = wine.target
target_names = wine.target_names
feature_names = wine.feature_names
```

Split data into testing and training sets and normalize it.

```python
X_train, X_test, y_train, y_test = train_test_split(data,
                                                  target,
                                                  test_size=0.2,
```
30.3 Fitting a support vector classifier (SVC) to the Wine dataset

30.3.1 Training

SVM, is a binary classifier, so multiple classifiers are fitted in order to support multiclass classification. The algorithm output is explained here.

```python
np.random.seed(0)
classifier = SVC(
    kernel = 'rbf',
    C=1,
    gamma = 0.1,
    decision_function_shape='ovr',  # n_cls trained with data from one class as postive and remainder of data as neg
    random_state = 0,
)
classifier.fit(X_train_norm, y_train)
```

30.3.2 Model assessment

Look at confusion matrix.

```python
y_pred = classifier.predict(X_test_norm)
cm = confusion_matrix(y_test, y_pred)
title = 'Confusion matrix for SVC classifier'
disp = plot_confusion_matrix(classifier,
    X_test_norm,
    y_test,
    display_labels=target_names,
    cmap=plt.cm.Blues,
    normalize=None,
)
disp.ax_.set_title(title)
```
The confusion matrix show the classifier is perfect - let’s understand what patterns in the data help the SVC perform so well!

### 30.4 Apply KernelSHAP to explain the model

The model needs access to a function that takes as an input samples and returns predictions to be explained. For an input $z$ the decision function of an binary SVM classifier is given by:

$$\text{class}(z) = \text{sign}(\beta z + b)$$

where $\beta$ is the best separating hyperplane (linear combination of support vectors, the training points closest to the separating hyperplane) and $b$ is the bias of the model.

For the ‘one-vs-rest’ SVM, $n_{\text{class}}$ binary SVM algorithms are fitted using each class as the positive class and the remainder as negative class. The classification decision is taken by assigning the label from the classifier with the maximum absolute decision score. Therefore, to explain our model we could consider explaining the SVM model which outputs the highest decision score. Click here to go back to source.

To do so, the KernelSHAP explainer must receive a callable that returns a set of scores when called with an input $X$, in this case the decision_function attribute of our classifier.

```python
[10]: pred_fcn = classifier.decision_function

[11]: np.random.seed(0)
svm_explainer = KernelShap(pred_fcn)
svm_explainer.fit(X_train_norm)
```

Using 142 background data samples could cause slower run times. Consider using `shap.sample` or `shap.kmeans` to summarize the background as $K$ samples.

```python
[11]: KernelShap(meta={
    'name': 'KernelShap',
    'type': ['blackbox'],
    'task': 'classification',
    'explanations': ['local', 'global'],
(continues on next page)```
Note that the explainer is fit to the classifier training set. This training set is used for two purposes:

- To determine the model output when all inputs are missing ($\phi_0$ in eq. (5) of [I]). Because the SVM model does not accept arbitrary inputs, this quantity is approximated by averaging the decision score for each class, across the samples in $X_{\text{train norm}}$ as shown below and it is stored as the expected_value attribute of the explainer.

- The values of the features in the $N \times D$ $X_{\text{train norm}}$ dataset are used to replace the values missing during the feature attribution ($\phi_i$) estimation process. Specifically, $n_{\text{samples}}$ copies of $X_{\text{train norm}}$ are tiled to create a dataset where, for each copy, a subset of features $z'$ of size $s = |z'|$ are replaced by the values in the instance to be explained and the complement of this subset is left to the background dataset value. These background values simulate the effect of missing values, since most models cannot cope with arbitrary patterns of missing values at inference time. Therefore, when computing the shap value of a particular feature, $\phi_i$, $n_{\text{samples}}$ regression targets ($f(h_{\text{test}}(z'))$ in eq (5) of [I]) are computed as the expected prediction of the model to be explained when a given subset of features is missing as opposed to replacing the missing feature with a single value. Note that the averaging operation can be replaced by weighted averaging by specifying the weights argument to the fit method. (a)

For the above reason, this is sometimes referred to as the background dataset; a larger dataset increases the runtime of the algorithm, so for large datasets, a subset of it should be used. An option to deal with the runtime issue while still providing meaningful values for missing values is to summarise the dataset using the shap.kmeans function. This function wraps the sklearn k-means clustering implementation, while ensuring that the clusters returned have values that are found in the training data. In addition, the samples are weighted according to the cluster sizes.

```
[12]: # expected_values attribute stores average scores across training set for every...
    mean_scores_train = pred_fcn(X_train_norm).mean(axis=0)
    # are stored in the expected_value attribute of the explainer ...
    print(mean_scores_train - svm_explainer.expected_value)
    [-1.11022302e-16  4.44089210e-16 -4.44089210e-16]

[13]: svm_explanation = svm_explainer.explain(X_test_norm, l1_reg=False)
```

In cases where the feature space has higher dimensionality, only a small fraction of the missing subsets can be enumerated for a given number of samples $n_{\text{samples}}$. If the the fraction of the subsets enumerated falls below a fraction (0.2 for version 0.3.2) and the regularisation is set to auto, a least angle regression with the AIC information criterion for selecting the regularisation coefficient $\alpha$ is performed in order to select features. The regularisation has no effect if the fraction is greater than this threshold and l1_reg is not set to auto. Other options for regularisations are: - l1_reg="num_features(10)"; in this case, the LARS algorithm [2] is used to which 10 features to estimate the shap values for. - l1_reg="bic"; in this case, the least angle regression is run with the Bayes Information Criterion - l1_reg=0.02: if a float is specified, the $\ell_1$-regularised regression coefficient is set to this value.
30.4.1 Local explanation

Because the SVM algorithm returns a score for each of the 3 classes, the shap_values are computed for each class in turn. Moreover, the attributions are computed for each data point to be explained and for each feature, resulting in a $N_e \times D$ matrix of shap values for each class, where $N_e$ is the number of instances to be explained and $D$ is the number of features.

```python
[14]: print("Output type:", type(svm_explanation.shap_values))
print("Output size:", len(svm_explanation.shap_values))
print("Class output size:", svm_explanation.shap_values[0].shape)
Output type: <class 'list'>
Output size: 3
Class output size: (36, 13)
```

For a given instance, we can visualise the attributions using a force plot. Let’s choose the first example in the testing set as an example.

```python
[15]: idx = 0
instance = X_test_norm[idx][None, :]
pred = classifier.predict(instance)
scores = classifier.decision_function(instance)
class_idx = pred.item()
print("The predicted class for the X_test_norm[{} is {}\].".format(idx, *pred))
print("OVR decision function values are \{}\].".format(*scores))
The predicted class for the X_test_norm[0] is 0.
OVR decision function values are [ 2.24071294 0.85398239 -0.21510456].
```

We see that class 0 is predicted because the SVM model trained with class 0 as a positive class and classes 1 and 2 combined as a negative class returned the largest score.

To create this force plot, we have provided the plotting function with four inputs: - the expected predicted score by the class-0 SVM assuming all inputs are missing. This is marked as the base value on the force plot - the feature attributions for the instance to be explained - the instance to be explained - the feature names

```python
[16]: shap.force_plot(
    svm_explainer.expected_value[class_idx],
    svm_explanation.shap_values[class_idx][idx, :],
    instance,
    feature_names,
)
```

```python
[16]: <shap.plots._force.AdditiveForceVisualizer at 0x7fe4f51bb2d0>
```

The force plot depicts the contribution of each feature to the process of moving the value of the decision score from the base value (estimation of the decision score if all inputs were missing) to the value predicted by the classifier. We see that all features contribute to increasing the decision score, and that the largest increases are due to the proline feature with a value of 1.049 and the flavanoids feature with a value of 0.9778. The lengths of the bars are the corresponding feature attributions.

Similarly, below we see that the proline and alcohol features contribute to decreasing the decision score of the SVM predicting class 1 as positive and that the malic_acid feature increases the decision score.

```python
[17]: shap.force_plot(
    svm_explainer.expected_value[1],
    svm_explanation.shap_values[1][idx, :],
    instance,
```

(continues on next page)
An alternative way to visualise local explanations for multi-output models is a *multioutput decision plot*. This plot can be especially useful when the number of features is large and the force plot might not be readable.

```python
def class_labels(classifier, instance, class_names=None):
    """
    Creates a set of legend labels based on the decision scores of a classifier and, optionally, the class names.
    """
    decision_scores = classifier.decision_function(instance)
    if not class_names:
        class_names = [f'Class {i}' for i in range(decision_scores.shape[1])]
    for i, score in enumerate(np.nditer(decision_scores)):
        class_names[i] = class_names[i] + ' ({}).format(round(score.item(),3))
    return class_names

legend_labels = class_labels(classifier, instance)

r = shap.multioutput_decision_plot(svm_explainer.expected_value.tolist(),
                   svm_explanation.shap_values,
                   idx,
                   feature_names=feature_names,
                   feature_order='importance',
                   highlight=[class_idx],
                   legend_labels=legend_labels,
                   return_objects=True,
                   legend_location='lower right')
```

194 Chapter 30. Kernel SHAP explanation for SVM models
The decision plots show how the individual features influence the contribution to the classification into each of the three classes (a prediction path). One sees that, for this example, the model can easily separate the three classes. It also shows, for example, that a wine with the given alcohol content is typical of class 0 (because the alcohol feature contributes positively to a classification of class 0 as negatively to classification in classes 1 and 2).

Note that the feature ordering is determined by summing the shap value magnitudes corresponding to each feature across classes and then ordering the feature_names in descending order of cumulative magnitude. The plot origin, marked by the gray vertical line, is the average base values across the classes. The dashed line represents the model prediction - in general we can highlight a particular class by passing the class index in the highlight list.

Suppose now that we want to analyse instance 5 but realise that the feature importances are different for this instance.

```python
[21]: idx = 5
instance = X_test_norm[idx][None, :]
pred = classifier.predict(instance)

[22]: instance_shap = np.array(svm_explanation.shap_values)[:, idx, :]
feature_order = np.argsort(np.sum(np.abs(instance_shap),axis=0))[::-1]
feat_importance = [feature_names[i] for i in feature_order]

[23]: print(feat_importance)
['flavanoids', 'alcalinity_of_ash', 'od280/od315_of_diluted_wines', 'alcohol', 'ash',
→'total_phenols', 'proline', 'magnesium', 'proanthocyanins', 'hue', 'malic_acid',
→'nonflavanoid_phenols', 'color_intensity']
```

We want to create a multi-output decision plot with the same feature order and scale. This is possible, since by passing the return_objects=True to the plotting function in the example above, we retrieved the feature indices and the axis limits and can reuse them to display the decision plot for instance 5 with the same feature order as above.

30.4. Apply KernelSHAP to explain the model
30.4.2 Global explanation

As shown above, the force plot allows us to understand how the individual features contribute to a classification output given an instance. However, the particular explanation does not tell us about the model behaviour in general. Below, we show how such insights can be drawn.
Stacked force plots

The simplest way we can do this is to stack the force plot for a number of instances, which can be achieved by calling the `force_plot` function with the same arguments as before but replacing `instance` with the whole testing set, `X_test_norm`.

```python
[26]: class_idx = 0  # we explain the predicted label
    shap.force_plot(
        svm_explainer.expected_value[class_idx],
        svm_explanation.shap_values[class_idx],
        X_test_norm,
        feature_names,
    )

[26]: <shap.plots._force.AdditiveForceArrayVisualizer at 0x7fe4f476d990>
```

In the default configuration, the \(x\) axis is represented by the 36 instances in `X_test_norm` whereas the \(y\) axis represents the decision score. Note that, like before, the decision score is for class 0. For a given instance, the height in between two horizontal lines is equal to the shap value of the feature, and hovering over a plot shows a list of the features along with their values, sorted by shap values. As before, the blue shading shows a negative contribution to the decision score (moves the score away from the baseline value) whereas the pink shading shows a positive contribution to the decision score. Hovering over the plot, tells us, for example, that to achieve a high decision score (equivalent to class 0 membership) the features proline and flavanoids are generally the most important and that positive proline values lead to higher decision scores for belonging to this class whereas negative proline values provide evidence against this one belonging to 0 class.

To see the relationship between decision scores and the values more clearly, we can permute the \(x\) axis so that the instances are sorted according to the value of the proline feature by selecting proline from the horizontal drop down menu.

```python
[28]: shap.force_plot(
    svm_explainer.expected_value[class_idx],
    svm_explanation.shap_values[class_idx],
    X_test_norm,
    feature_names,
)

[28]: <shap.plots._force.AdditiveForceArrayVisualizer at 0x7fe4f4f757910>
```

You can also explore the effect of a particular feature across the testing dataset. For example, in the plot below, by selecting flavanoids from the top drop-down, the instances are ordered on the \(x\) axis in increasing value of the flavanoids feature.

Similarly, selecting flavanoids effects from the side drop-down will plot the shap value as opposed to the model output. The effect of this feature generally increases as its value increases and the large negative values of this feature reduce the decision score for classification as 0. Note that the shap values are respresented with resepect to the base value for this class (0.798, as shown below).

```python
[29]: shap.force_plot(
    svm_explainer.expected_value[0],
    svm_explanation.shap_values[0],
    X_test_norm,
    feature_names,
)

[29]: <shap.plots._force.AdditiveForceArrayVisualizer at 0x7fe4f4f757f10>
```
Summary plots

To visualise the impact of the features on the decision scores associated with class 0, we can use a summary plot. In this plot, the features are sorted by the sum of their SHAP values magnitudes across all instances in \( X_{\text{test norm}} \). Therefore, the features with the highest impact on the decision score for class \( \text{class idx} \) are displayed at the top of the plot.

In this case, the proline and flavanoids have the most impact on the model output; as the values of the features increase, their impact also increases and the model is more likely to predict class 0. On the other hand, high values of the nonflavonoid phenols have a negative impact on the model output, potentially contributing to the classification of the particular wine in a different class. To see this, we do a summary plot with respect to \( \text{class}\_2 \).

```python
[31]: shap.summary_plot(svm_explanation.shap_values[1], X_test_norm, feature_names)
```
We see that, indeed, a higher value of the `nonflavonoid_phenols` feature contributes to a sample being classified as class 1, but that this effect is rather limited compared to features such as `proline` or `alcohol`.

To visualise the impact of the feature across all classes, that is, the importance of a particular feature for the model, we simply pass all the shap values to the `summary_plot` functions. We see, that, for example, the `color_intensity` feature is much more important for deciding whether an instance should be classified as class 2 then in class 0.

```python
[33]: shap.summary_plot(svm_explanation.shap_values, X_test_norm, feature_names)
```
Another way to visualise the model dependence on a particular feature is through a dependence plot. This plot shows the impact of the feature value on its importance for classification with respect to class 0.

```python
[34]: feature = 'flavanoids'
shap.dependence_plot(
    feature,
    svm_explanation.shap_values[0],
    X_test_norm,
    feature_names=feature_names,
    interaction_index='auto',
)
```
The colour of the individual instances is represented by the value of the feature nonflavanoid_phenols. By specifying interaction_index=auto, the nonflavanoid_phenols was estimated as the feature with the strongest interaction with the flavanoids_feature; this interaction is approximate, and is estimate by computing the Pearson Correlation Coefficient between the shap values of the reference feature (flavanoids in this case) and the value of each feature in turn on bins along the feature value.

We see that, for class 0 wines, a higher value for nonflavanoid_phenols is generally associated with a low value in flavanoids and that they have a negative impact on the score for class 0 classification.

30.4.3 Footnotes

(a) The weights are applied to each point in a copy, so the number of weights should be the same as the number of samples in the data.

30.4.4 References


CHAPTER THIRTYONE

KERNEL SHAP EXPLANATION FOR MULTINOMIAL LOGISTIC REGRESSION MODELS

31.1 Introduction

In a previous example, we showed how the KernelSHAP algorithm can be applied to explain the output of an arbitrary classification model so long the model outputs probabilities or operates in margin space. We also showcased the powerful visualisations in the shap library that can be used for model investigation. In this example we focus on understanding, in a simple setting, how conclusions drawn from the analysis of the KernelShap output relate to conclusions drawn from interpreting the model directly. To make this possible, we fit a logistic regression model on the Wine dataset.

```python
[1]: import shap
    shap.initjs()

    import matplotlib.pyplot as plt
    import numpy as np

    from alibi.explainers import KernelShap
    from scipy.special import logit
    from sklearn.datasets import load_wine
    from sklearn.metrics import confusion_matrix, plot_confusion_matrix
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import StandardScaler
    from sklearn.linear_model import LogisticRegression

<IPython.core.display.HTML object>
```

31.2 Data preparation: load and split Wine dataset

```python
[2]: wine = load_wine()
    wine.keys()

[2]: dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names'])

[3]: data = wine.data
    target = wine.target
    target_names = wine.target_names
    feature_names = wine.feature_names

Split data into testing and training sets and normalize it.
```
31.3 Fitting a multinomial logistic regression classifier to the Wine dataset

31.3.1 Training

```python
[6]: classifier = LogisticRegression(multi_class='multinomial',
                               random_state=0,
                           )
classifier.fit(X_train_norm, y_train)
```

31.3.2 Model assessment

```python
[7]: y_pred = classifier.predict(X_test_norm)
[8]: cm = confusion_matrix(y_test, y_pred)
[9]: title = 'Confusion matrix for the logistic regression classifier'
disp = plot_confusion_matrix(classifier, 
                          X_test_norm,
                          y_test,
                          display_labels=target_names,
                          cmap=plt.cm.Blues,
                          normalize=None,
                      )
disp.ax_.set_title(title)
[9]: Text(0.5, 1.0, 'Confusion matrix for the logistic regression classifier')
```
One way to arrive at the multinomial logistic regression model is to consider modelling a categorical response variable \( y \sim \text{Cat}(y|\beta x) \) where \( \beta \) is \( K \times D \) matrix of distribution parameters with \( K \) being the number of classes and \( D \) the feature dimensionality. Because the probability of outcome \( k \) being observed given \( x \), \( p_k = p(y = k|x, \beta) \), is bounded by \([0, 1]\), the logistic regression assumes that a linear relationship exists between the logit transformation of the output and the input. This can be formalised as follows:

\[
\log\left( \frac{p_k}{1 - p_k} \right) = \beta_{0,k} + \beta_{1,k} x_1 + \beta_{2,k} x_2 + \cdots + \beta_{D,k} x_D = \beta_k \cdot x
\]  

(31.1)

The RHS is a function of the expected value of the categorical distribution (sometimes referred to as a link function in the literature). The coefficients \( \beta \) of the linear relations used to fit the logit transformation are estimated jointly given a set of training examples \( \mathcal{D} = \{(x_i, y_i)\}_{i=1}^N \).

For each class, the vector of coefficients \( \beta_k \) can be used to interpret the model globally; in the absence of interaction terms, the coefficient of a predictor (i.e., independent variable) represents the change in log odds when the predictor changes by one unit while all other variables are kept at fixed values. Equivalently, the exponentiated coefficient is equivalent to a change in odds. Since the transformation from odds to outcome probabilities is monotonic, a change in odds also implies a change in the outcome probability in the same direction. Thus, the magnitudes of the feature coefficients measure the effect of a predictor on the output and thus one can globally interpret the logistic regression model.

However, the log odds ratios and odds ratios are known to be sensitive to unobserved heterogeneity, that is, omission of a variable with good explanatory power from a logistic regression model assumed true. While we will not be concerned directly with this issue and refer the interested reader to [2], we will be using the estimated percentage unit effect (or the marginal effect)

\[
\beta_{j,k} \times p_{i,k}(1 - p_{i,k})
\]

as a means of estimating the effect of a predictor \( j \) on individual \( i \) \((x_{i,j})\) with respect to predicting the \( k^{th} \) class and thus locally interpret the model. The average marginal effect is more robust measure of effects in situations where effects are compared across different groups or models. Consider a logistic model where an independent variable \( x_1 \)
is used to predict an outcome and a logistic model where \( x_2 \), known to be uncorrelated with \( x_1 \), is also included. Since the two models assign different probabilities to the different outcomes and since the distribution of the outcome across values of \( x_1 \) should be the same across the two models (due to the independence assumption), we expected the second model will scale the coefficient of \( \beta_1 \). Hence, the log-odds and odds ratios are not robust to unobserved heterogeneity so directly comparing the two across models or groups can be misleading. As discussed in [2], the marginal effect is generally robust to the effect.

The average marginal effect (AME) of a predictor

\[
\frac{1}{n} \sum_{i=1}^{n} \beta_{j,k} \times p_{i,k}(1 - p_{i,k})
\]

is equivalent to simply using \( \beta_{j,k} \) to globally explain the model.

```python
[10]:
def issorted(arr, reverse=False):
    """
    Checks if a numpy array is sorted.
    """
    if reverse:
        return np.all(arr[::-1][:-1] <= arr[::-1][1:])
    return np.all(arr[:-1] <= arr[1:])
def get_importance(class_idx, beta, feature_names, intercepts=None):
    """
    Retrive and sort abs magnitude of coefficients from model.
    """
    # sort the absolute value of model coef from largest to smallest
    srt_beta_k = np.argsort(np.abs(beta[class_idx, :]))[::-1]
    feat_names = [feature_names[idx] for idx in srt_beta_k]
    feat_imp = beta[class_idx, srt_beta_k]
    # include bias among feat importances
    if intercepts is not None:
        intercept = intercepts[class_idx]
        bias_idx = len(feat_imp) - (np.searchsorted(np.abs(feat_imp)[::-1], np.abs(intercept)) + 1)
        feat_imp = np.insert(feat_imp, bias_idx, intercept.item(), )
        intercept_idx = np.where(feat_imp == intercept)[0][0]
        feat_names.insert(intercept_idx, 'bias')
    return feat_imp, feat_names
def plot_importance(feat_imp, feat_names, **kwargs):
    """
    Create a horizontal barchart of feature effects, sorted by their magnitude.
    """
    left_x, right_x = kwargs.get("left_x"), kwargs.get("right_x")
eps_factor = kwargs.get("eps_factor", 4.5)
xlabel = kwargs.get("xlabel", None)
ylabel = kwargs.get("ylabel", None)
labels_fontsize = kwargs.get("labels_fontsize", 15)
tick_labels_fontsize = kwargs.get("tick_labels_fontsize", 15)
```

(continues on next page)
# plot
fig, ax = plt.subplots(figsize=(10, 5))
y_pos = np.arange(len(feat_imp))
ax.barh(y_pos, feat_imp)

# set lables
ax.set_yticks(y_pos)
ax.set_yticklabels(feat_names, fontsize=tick_labels_fontsize)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel(xlabel, fontsize=labels_fontsize)
ax.set_ylabel(ylabel, fontsize=labels_fontsize)
ax.set_xlim(left=left_x, right=right_x)

# add text
for i, v in enumerate(feat_imp):
    eps = 0.03
    if v < 0:
        eps = -eps_factor*eps
    ax.text(v + eps, i + .25, str(round(v, 3)))

return ax, fig

We now retrieve the estimated coefficients, and plot them sorted by their magnitude.

[11]: beta = classifier.coef_
     intercepts = classifier.intercept_
     all_coefs = np.concatenate((beta, intercepts[:, None]), axis=1)

[12]: class_idx = 0
     feat_imp, feat_names = get_importance(class_idx, beta, feature_names,)

[13]: _, class_0_fig = plot_importance(feat_imp, feat_names, left_x=-1.,
                                right_x=1.25, xlabel = f"Feature effects (class {class_idx})",
                                ylabel = "Features")
Note that these effects are with respect to the model bias (displayed below).

This plot shows that features such as proline, flavanoids, od280/od315_of_diluted_wines, alcohol increase the odds of any sample being classified as class_0 whereas the alcalinity_of_ash decreases them.

The plot below shows that, however, alcalinity_of_ash increases the odds of a wine being in class_1. Predictors such as proline, alcohol or ash, which increase the odds of predicting a wine as a member of class_0, decrease the odds of predicting it as a member of class_1.
Finally, for class 2, the color_intensity, ash are the features that increase the class 2 odds.
31.5 Apply KernelSHAP to explain the model

Note that the *local accuracy* property of SHAP (eq. (5) in [1]) requires

\[ f(x) = g(x') = \phi_0 + \sum_{j=1}^{D} \phi_j x'_j. \]

Hence, sum of the feature importances, \( \phi_j \), should be equal to the model output, \( f(x) \). By passing `link='logit'` to the explainer, we ensure that \( \phi_0 \), the *base value* (see *Local explanation* section here) will be calculated in the correct units. Note that here \( x' \in \mathbb{R}^D \) represents a *simplified input* for which the shap value is computed. A simple example of a simplified input in the image domain, justified by the dimensionality of the input space, is a *superpixel mask*: we formulate the task of explaining the outcome of an image prediction task as determining the effects of each superpixel in a segmented image upon the outcome. The interested reader is referred to [1] for more details about simplified inputs.

```python
[19]: pred_fcn = classifier.predict_proba
    lr_explainer = KernelShap(pred_fcn, link='logit')
    lr_explainer.fit(X_train_norm)

Using 142 background data samples could cause slower run times. Consider using shap.sample(data, K) or shap.kmeans(data, K) to summarize the background as K samples.

[19]: KernelShap(meta={
    'name': 'KernelShap',
    'type': ['blackbox'],
    'explanations': ['local', 'global'],
    'params': {
        'groups': None,
        'group_names': None,
        'weights': None,
        'summarise_background': False
    }
})
```

```python
[20]: # passing the logit link function to the explainer ensures the units are consistent ..
    mean_scores_train = logit(pred_fcn(X_train_norm).mean(axis=0))
    print(mean_scores_train - lr_explainer.expected_value)

[-6.66133815e-16 0.00000000e+00 4.44089210e-16]
```

```python
[21]: lr_explanation = lr_explainer.explain(X_test_norm, ll_reg=False)

HBox(children=(IntProgress(value=0, max=36), HTML(value='')))
```

Because the dimensionality of the feature space is relatively small, we opted not to regularise the regression that computes the Shapley values. For more information about the regularisation options available for higher dimensional data see the introductory example here.
31.5.1 Locally explaining multi-output models with KernelShap

31.5.2 Explaining the logistic regression model globally with KernelSHAP

Summary plots

To visualise the impact of the features on the decision scores associated with class \texttt{class_idx}, we can use a summary plot. In this plot, the features are sorted by the sum of their SHAP values magnitudes across all instances in \texttt{X_test_norm}. Therefore, the features with the highest impact on the decision score for class \texttt{class_idx} are displayed at the top of the plot.

![Summary plot of feature importance](image)

Because the logistic regression model uses a linear predictor function, the exact shap values for each class \(k\) can be computed exactly according to (11)

\[
\phi_{i,j}(f, x_i) = \beta_{j,k}(x_{i,j} - E_D[x_j]).
\]

Here we introduced an additional index \(i\) to emphasize that we compute a shap value for each predictor and each instance in a set to be explained. This allows us to check the accuracy of the SHAP estimate. Note that we have already applied the normalisation so the expectation is not subtracted below.

![Plot of exact shap values](image)
alibi Documentation, Release 0.5.6

The plot below shows that the exact shap values and the estimate values give rise to similar ranking of the features, and only the order of the flavanoids and alcohol features is swapped.

```python
shap.summary_plot(exact_shap[class_idx, ...], X_test_norm, feature_names)
```

212 Chapter 31. Kernel SHAP explanation for multinomial logistic regression models
An similar plot can be create for the logistic regression model by plotting the marginal effects. Note that the plot labelling cannot be changed, so the x axis is incorrectly labeled as SHAP value below.

```python
[26]: p = classifier.predict_proba(X_test_norm)
    prb = p * (1. - p)
    marg_effects = all_coefs[:,:,:] * prb[..., None] # nb: ranking of the feature coefs should be preserved
    avg_marg_effects = np.mean(marg_effects, axis=1)
    mask = np.ones_like(X_test_norm) # the effect (positive vs negative) on the output depend on the sign of the input
    mask[X_test_norm < 0] = -1

[27]: shap.summary_plot(marg_effects[class_idx, :, :-1]*mask, X_test_norm, feature_names) # exclude bias
```

31.5. Apply KernelSHAP to explain the model 213
As expected, the ranking of the marginal effects is the same as that provided the ranking the raw coefficients (see below). However, this effect measure allows us to assess the effects at instance level. Note that both the approximate computation and the exact method yield the same group of features as the most important, although their rankings are not identical. It is important to note that the exact effects ranking and absolute values is a function of the entire data (due to the dependence of the model coefficients) whereas the approximate computation is local: the explanation model is fitted locally around each instance. We also notice that the approximate and exact shap value computation both identify the same relationship between the feature value and the effect on the evidence of a sample belonging to class_idx.

Looking at the 6 most important features for this classification in class_0, we see that both the KernelSHAP method and the logistic regression rank the proline feature as the one with the most significant effect. While the
order of the subsequent 5 features is permuted, the effects of these features are also very similar so, in effect, similar conclusions would be drawn from analysing either output.

31.5.3 References


32.1 Introduction

In this example, we show how the KernelSHAP method can be used for tabular data, which contains both numerical (continuous) and categorical attributes. Using a logistic regression model fitted to the Adult dataset, we examine the performance of the KernelSHAP algorithm against the exact shap values. We investigate the effect of the background dataset size on the estimated shap values and present two ways of handling categorical data.

```python
import shap
shap.initjs()

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from alibi.explainers import KernelShap
from alibi.datasets import fetch_adult
from scipy.special import logit
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, plot_confusion_matrix
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

32.2 Data preparation

32.2.1 Load and split

The `fetch_adult` function returns a Bunch object containing the features, the targets, the feature names and a mapping of categorical variables to numbers.

```python
adult = fetch_adult()
adult.keys()
```

```python
dict_keys(['data', 'target', 'feature_names', 'target_names', 'category_map'])
```
Note that for your own datasets you can use our utility function `gen_category_map` to create the category map.

```python
from alibi.utils.data import gen_category_map
```

```python
np.random.seed(0)
data_perm = np.random.permutation(np.c_[data, target])
data = data_perm[:, :-1]
target = data_perm[:, -1]
```

```python
idx = 30000
X_train, y_train = data[:idx, :], target[:idx]
X_test, y_test = data[idx+1:, :], target[idx+1:]
```

### 32.2.2 Create feature transformation pipeline

#### 32.2.3 Create feature transformation pipeline

Create feature pre-processor. Needs to have ‘fit’ and ‘transform’ methods. Different types of pre-processing can be applied to all or part of the features. In the example below we will standardize ordinal features and apply one-hot-encoding to categorical features.

Ordinal features:

```python
ordinal_features = [x for x in range(len(feature_names)) if x not in list(category_map.keys())]
ordinal_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')), ('scaler', StandardScaler())])
```

Categorical features:

```python
categorical_features = list(category_map.keys())
categorical_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')), ('onehot', OneHotEncoder(drop='first', handle_unknown='error'))])
```

Note that in order to be able to interpret the coefficients corresponding to the categorical features, the option `drop='first'` has been passed to the `OneHotEncoder`. This means that for a categorical variable with $n$ levels, the length of the code will be $n-1$. This is necessary in order to avoid introducing feature multicolinearity, which would skew the interpretation of the results. For more information about the issue about multicolinearity in the context of linear modelling see [1].

Combine and fit:

```python
preprocessor = ColumnTransformer(transformers=[('num', ordinal_transformer, ordinal_features), ('cat', categorical_transformer, categorical_features)])
preprocessor.fit(X_train)
```
32.2.4 Preprocess the data

[10]: X_train_proc = preprocessor.transform(X_train)
    X_test_proc = preprocessor.transform(X_test)

32.3 Fit a binary logistic regression classifier to the Adult dataset

32.3.1 Training

[11]: classifier = LogisticRegression(multi_class='multinomial',
random_state=0,
max_iter=500,
verbose=0,
(continues on next page)
```python
classifier.fit(X_train_proc, y_train)

[11]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
   intercept_scaling=1, l1_ratio=None, max_iter=500,
   multi_class='multinomial', n_jobs=None, penalty='l2',
   random_state=0, solver='lbfgs', tol=0.0001, verbose=0,
   warm_start=False)
```

### 32.3.2 Model assessment

```python
y_pred = classifier.predict(X_test_proc)

cm = confusion_matrix(y_test, y_pred)

title = 'Confusion matrix for the logistic regression classifier'
disp = plot_confusion_matrix(classifier,
   X_test_proc,
   y_test,
   display_labels=target_names,
   cmap=plt.cm.Blues,
   normalize=None,
)
disp.ax_.set_title(title)

[14]: Text(0.5, 1.0, 'Confusion matrix for the logistic regression classifier')
```

Confusion matrix for the logistic regression classifier

```
Confusion matrix for the logistic regression classifier

<table>
<thead>
<tr>
<th></th>
<th>&lt;=50K</th>
<th>&gt;50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;=50K</td>
<td>18e+03</td>
<td>1.3e+02</td>
</tr>
<tr>
<td>&gt;50K</td>
<td>2.4e+02</td>
<td>3.7e+02</td>
</tr>
</tbody>
</table>

[15]: print('Test accuracy: ', accuracy_score(y_test, classifier.predict(X_test_proc)))
```
32.4 Intepreting the logistic regression model

In order to interpret the logistic regression model, we need to first recover the encoded feature names. The feature effect of a categorical variable is computed by summing the coefficients of the encoded variables. Hence, we first understand how the preprocessing transformation acts on the data and then obtain the overall effects from the model coefficients.

First, we are concerned with understanding the dimensionality of a preprocessed record and what it is comprised of.

```python
idx = 0
print("The dimensionality of a preprocessed record is {}.".format(X_train_proc[idx: ~idx+1, :].shape))
print("Then number of continuos features in the original data is {}.". ~format(len(ordinal_features)))
The dimensionality of a preprocessed record is (1, 49).
Then number of continuos features in the original data is 4.
```

Therefore, of 49, 45 of the dimensions of the original data are encoded categorical features. We obtain `feat_enc_dim`, an array with the lengths of the encoded dimensions for each categorical variable that will be use for processing the results later on.

```python
fts = [feature_names[x] for x in categorical_features]
# get feature names for the encoded categorical features
ohe = preprocessor.transformers_[1][1].named_steps['onehot']
cat_enc_feat_names = ohe.get_feature_names(fts)
    # compute encoded dimension; -1 as ohe is setup with drop='first'
feat_enc_dim = [len(cat_enc) - 1 for cat_enc in ohe.categories_]
d = {'feature_names': fts, 'encoded_dim': feat_enc_dim}
df = pd.DataFrame(data=d)
print(df)
total_dim = df['encoded_dim'].sum()
print("The dimensionality of the encoded categorical features is {}.")
assert total_dim == len(cat_enc_feat_names)
```

<table>
<thead>
<tr>
<th>feature_names</th>
<th>encoded_dim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workclass</td>
<td>8</td>
</tr>
<tr>
<td>Education</td>
<td>6</td>
</tr>
<tr>
<td>Marital Status</td>
<td>3</td>
</tr>
<tr>
<td>Occupation</td>
<td>8</td>
</tr>
<tr>
<td>Relationship</td>
<td>5</td>
</tr>
<tr>
<td>Race</td>
<td>4</td>
</tr>
<tr>
<td>Sex</td>
<td>1</td>
</tr>
<tr>
<td>Country</td>
<td>10</td>
</tr>
</tbody>
</table>

The dimensionality of the encoded categorical features is 45.

By analysing an encoded record, we can recover the mapping of column indices to the features they represent.

```python
print(X_train_proc[0, :])
```

(continues on next page)
(continued from previous page)

\[
\begin{pmatrix}
0, 7 & 1.0 \\
0, 15 & 1.0 \\
0, 19 & 1.0 \\
0, 21 & 1.0 \\
0, 32 & 1.0 \\
0, 37 & 1.0 \\
0, 47 & 1.0 \\
\end{pmatrix}
\]

```
[19]: numerical_feats_idx = preprocessor.transformers_[0][2]
categorical_feats_idx = preprocessor.transformers_[1][2]
scaler = preprocessor.transformers_[0][1].named_steps[\'scaler\']
print((X_train[idx, numerical_feats_idx] - scaler.mean_)/scaler.scale_)
num_feats_names = [feature_names[i] for i in numerical_feats_idx]
cat_feats_names = [feature_names[i] for i in categorical_feats_idx]
print(num_feats_names)
```

\[-0.84644563 -0.14513572 -0.21784552 0.28898152\]

\['Age', 'Capital Gain', 'Capital Loss', 'Hours per week'\]

Therefore, the first four columns of the encoded data represent the **Age**, **Capital Gain**, **Capital Loss** and **Hours per week** features. Notice these features have a different index in the dataset prior to processing `X_train`.

The remainder of the columns encode the encoded categorical features, as shown below.

```
[20]: print(cat_enc_feat_names)
```

\['Workclass_1.0' 'Workclass_2.0' 'Workclass_3.0' 'Workclass_4.0'
 'Workclass_5.0' 'Workclass_6.0' 'Workclass_7.0' 'Workclass_8.0'
 'Education_1.0' 'Education_2.0' 'Education_3.0' 'Education_4.0'
 'Education_5.0' 'Education_6.0' 'Marital Status_1.0' 'Marital Status_2.0'
 'Marital Status_3.0' 'Occupation_1.0' 'Occupation_2.0' 'Occupation_3.0'
 'Occupation_4.0' 'Occupation_5.0' 'Occupation_6.0' 'Occupation_7.0'
 'Occupation_8.0' 'Relationship_1.0' 'Relationship_2.0' 'Relationship_3.0'
 'Relationship_4.0' 'Relationship_5.0' 'Race_1.0' 'Race_2.0' 'Race_3.0'
 'Race_4.0' 'Sex_1.0' 'Country_1.0' 'Country_2.0' 'Country_3.0'
 'Country_4.0' 'Country_5.0' 'Country_6.0' 'Country_7.0' 'Country_8.0'
 'Country_9.0' 'Country_10.0'\]

To obtain a single coefficient for each categorical variable, we pass a list with the indices where each encoded categorical variable starts and the encodings dimensions to the `sum_categories` function.

```
[21]: from alibi.explainers.shap_wrappers import sum_categories
```

Compute the start index of each categorical variable knowing that the categorical variables are adjacent and follow the continuous features.

```
[22]: start=len(ordinal_features)
cat_feat_start = [start]
for dim in feat_enc_dim[:-1]:
    cat_feat_start.append(dim + cat_feat_start[-1])
```

```
[23]: beta = classifier.coef_
beta = np.concatenate((-beta, beta), axis=0)
intercepts = classifier.intercept_
intercepts = np.concatenate((-intercepts, intercepts), axis=0)
all_coef = sum_categories(beta, cat_feat_start, feat_enc_dim)
```
Extract and plot feature importances. Please see this example for background on interpreting logistic regression coefficients.

```python
[24]: def get_importance(class_idx, beta, feature_names, intercepts=None):
    """
    Retrive and sort abs magnitude of coefficients from model.
    """
    # sort the absolute value of model coef from largest to smallest
    srt_beta_k = np.argsort(np.abs(beta[class_idx, :]))[::-1]
    feat_names = [feature_names[idx] for idx in srt_beta_k]
    feat_imp = beta[class_idx, srt_beta_k]
    # include bias among feat importances
    if intercepts is not None:
        intercept = intercepts[class_idx]
        bias_idx = len(feat_imp) - np.searchsorted(np.abs(feat_imp)[::-1], np.abs(intercept))
        feat_imp = np.insert(feat_imp, bias_idx, intercept.item(),)
        intercept_idx = np.where(feat_imp == intercept)[0][0]
        feat_names.insert(intercept_idx, 'bias')
    return feat_imp, feat_names

[25]: def plot_importance(feat_imp, feat_names, class_idx, **kwargs):
    """
    Create a horizontal barchart of feature effects, sorted by their magnitude.
    """
    left_x, right_x = kwargs.get("left_x"), kwargs.get("right_x")
    eps_factor = kwargs.get("eps_factor", 4.5)
    fig, ax = plt.subplots(figsize=(10, 5))
    y_pos = np.arange(len(feat_imp))
    ax.barh(y_pos, feat_imp)
    ax.set_yticks(y_pos)
    ax.set_yticklabels(feat_names, fontsize=15)
    ax.invert_yaxis()  # labels read top-to-bottom
    ax.set_xlabel(f'Feature effects for class {class_idx}', fontsize=15)
    ax.set_xlim(left=left_x, right=right_x)
    for i, v in enumerate(feat_imp):
        eps = 0.03
        if v < 0:
            eps = -eps_factor*eps
        ax.text(v + eps, i + .25, str(round(v, 3)))
    return ax, fig

[26]: class_idx = 0
perm_feat_names = num_feats_names + cat_feats_names

[26]: perm_feat_names  # feats are reordered by preprocessor
['Age',
'Capital Gain',
'Capital Loss',
'Hours per week',
'Workclass',

(continues on next page)
'Education',
'Marital Status',
'Occupation',
'Relationship',
'Race',
'Sex',
'Country']

[27]: feat_imp, srt_feat_names = get_importance(class_idx,
                      all_coef, perm_feat_names,
                      )

[28]: srt_feat_names

[28]: ['Marital Status',
          'Education',
          'Capital Gain',
          'Occupation',
          'Workclass',
          'Race',
          'Country',
          'Sex',
          'Relationship',
          'Hours per week',
          'Age',
          'Capital Loss']

[29]: _, class_0_fig = plot_importance(feat_imp,
                       srt_feat_names,
                       class_idx,
                       left_x=-2.5,
                       right_x=3.7,
                       eps_factor=12  # controls text distance from end of bar
                       )

<table>
<thead>
<tr>
<th>Feature</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marital Status</td>
<td>-1.306</td>
</tr>
<tr>
<td>Education</td>
<td>-1.174</td>
</tr>
<tr>
<td>Capital Gain</td>
<td>-0.875</td>
</tr>
<tr>
<td>Occupation</td>
<td>-0.764</td>
</tr>
<tr>
<td>Workclass</td>
<td>-0.731</td>
</tr>
<tr>
<td>Race</td>
<td>-0.66</td>
</tr>
<tr>
<td>Country</td>
<td>-0.43</td>
</tr>
<tr>
<td>Sex</td>
<td>-0.276</td>
</tr>
<tr>
<td>Relationship</td>
<td>-0.173</td>
</tr>
<tr>
<td>Hours per week</td>
<td>-0.16</td>
</tr>
<tr>
<td>Age</td>
<td>-0.132</td>
</tr>
<tr>
<td>Capital Loss</td>
<td>3.214</td>
</tr>
</tbody>
</table>

Feature effects for class 0
Note that in the above, the feature effects are with respect to the model bias, which has a value of 1.31.

```python
# Sanity check to ensure graph is correct.
print(beta[class_idx, 0:4]) # Age, Capital Gains, Capital Loss, Hours per week
print(np.sum(beta[class_idx, 18:21])) # Marital status
[-0.15993364 -1.1740675 -0.13217167 -0.17288934]
3.2136221376837284
```

### 32.5 Apply KernelSHAP to explain the model

Note that the local accuracy property of SHAP (eq. (5) in [1]) requires

\[ f(x) = g(x') = \phi_0 + \sum_{i=1}^{M} \phi_i x'_i. \]  

Hence, sum of the feature importances should be equal to the model output, \( f(x) \). By passing `link='logit'` to the explainer, we ensure that \( \phi_0 \), the base value (see *Local explanation* section here) will be calculated in the margin space (i.e., a logit transformation is applied to the probabilities) where the logistic regression model is additive.

Further considerations when applying the KernelSHAP method to this dataset are:

- **the background dataset size**: by setting a larger value for the `stop_example_idx` in the set below, you can observe how the runtime of the algorithm increases. At the same time, it is important to have a diverse but sufficiently large set of samples as background so that the missing feature values are correctly integrated. A way to reduce the number of samples is to pass the `summarise_background=True` flag to the explainer fit option along with the desired number of samples (n_background_samples). If there are no categorical variables in the data and there is no data grouping, then a k-means clustering algorithm is used to summarise the data. Otherwise, the data is sampled uniformly at random. Below, we used the `train_test_split` function of sklearn instead so that the label proportions are approximately the same as in the original split.

- **the number of instances to be explained**: the test set contains a number of 2560 records, which are 49-dimensional after pre-processing, as opposed to 13-dimensional as in the Wine dataset example. For this reason, only a fraction of `fraction_explained` (default 5%) are explained by way of getting a more general view of the model behaviour compared to simply analysing local explanations.

- **treating the encoded categorical features as a group**: of features that are *jointly* perturbed as opposed to being perturbed individually

```python
def split_set(X, y, fraction, random_state=0):
    """
    Given a set X, associated labels y, splits a fraction y from X.
    """
    _, X_split, _, y_split = train_test_split(X, y,
                                            test_size=fraction,
                                            random_state=random_state,
                                            )
    print("Number of records: {0:.0f}".format(X_split.shape[0]))
    print("Number of class {0}: {1:.0f}".format(0, len(y_split) - y_split.sum()))
    print("Number of class {0}: {1:.0f}".format(1, y_split.sum()))
    return X_split, y_split
```

```python
fraction_explained = 0.05
X_explain, y_explain = split_set(X_test,
```

(continues on next page)
```python
y_test, fraction_explained,
X_explain_proc = preprocessor.transform(X_explain)
```

Number of records: 128  
Number of class 0: 96  
Number of class 1: 32

#### 32.5.1 Exploiting explanation model additivity to estimate the effects of categorical features

Inspired by equation (1), a way to estimate the overall effect of a categorical variable is to treat its encoded levels as individual binary variables and sum the estimated effects for the encoded dimensions.

```python
pred_fcn = classifier.predict_proba
lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_names)
lr_explainer.fit(X_train_proc[background_data, :])
```

```python
KernelShap(meta={
    'name': 'KernelShap',
    'type': ['blackbox'],
    'explanations': ['local', 'global'],
    'params': {
        'groups': None,
        'group_names': None,
        'weights': None,
        'summarise_background': False
    }
})
```

```python
mean_scores_train = logit(pred_fcn(X_train_proc[background_data, :]).mean(axis=0))
# print(mean_scores_train - lr_explainer.expected_value)
```

```python
lr_explainer.expected_value
```

```python
array([ 1.08786649, -1.08786649])
```

```python
explanation = lr_explainer.explain(X_explain_proc,
    summarise_result=True,
    cat_vars_start_idx=cat_feat_start,
    cat_vars_enc_dim=feat_enc_dim,
)
```

We now sum the estimate shap values for each dimension to obtain one shap value for each categorical variable!
def rank_features(shap_values, feat_names):
    
    Given an NxF array of shap values where N is the number of
    instances and F number of features, the function ranks the
    shap values according to their average magnitude.
    
    avg_mag = np.mean(np.abs(shap_values), axis=0)
srt = np.argsort(avg_mag)[::-1]
rank_values = avg_mag[srt]
rank_names = [feat_names[idx] for idx in srt]
    
    return rank_values, rank_names

def get_ranked_values(explanation):
    
    Retrives a tuple of (feature_effects, feature_names) for
    each class explained. A feature's effect is its average
    shap value magnitude across an array of instances.
    
    ranked_shap_vals = []
    for cls_idx in range(len(explanation.shap_values)):
        this_ranking = {
            explanation.raw['importances'][str(cls_idx)]['ranked_effect'],
            explanation.raw['importances'][str(cls_idx)]['names']
        }
        ranked_shap_vals.append(this_ranking)
    
    return ranked_shap_vals

ranked_combined_shap_vals = get_ranked_values(explanation)

Because the columns have been permuted by the preprocessor, the columns of the instances to be explained have
to be permuted before creating the summary plot.

perm_feat_names

['Age',
'Capital Gain',
'Capital Loss',
'Hours per week',
'Workclass',
'Education',
'Marital Status',
'Occupation',
'Relationship',
'Race',
'Sex',
'Country']

def permute_columns(X, feat_names, perm_feat_names):
    
    Permutates the original dataset so that its columns
    (ordered according to feat_names) have the order

(continues on next page)
of the variables after transformation with the 
sklearn preprocessing pipeline (perm_feat_names).

```python
perm_X = np.zeros_like(X)
perm = []
for i, feat_name in enumerate(perm_feat_names):
    feat_idx = feat_names.index(feat_name)
    perm_X[:, i] = X[:, feat_idx]
    perm.append(feat_idx)
return perm_X, perm
```

[43]: perm_X_explain, _ = permute_columns(X_explain, feature_names, perm_feat_names)

[44]: shap.summary_plot(explanation.shap_values[0], perm_X_explain, perm_feat_names)

Note that the aggregated local explanations of this limited set are in partial agreement with the global explanation provided by the model coefficients. The top 3 most important features are determined to be the same. We can see that, high values of the Capital Gains decrease the odds of a sample being classified as class_0 (income <$50k).
32.5.2 Grouping features with KernelShap

An alternative way to deal with one-hot encoded categorical variables is to group the levels of a categorical variable and treat them as a single variable during the sampling process that generates the training data for the explanation model. Dealing with the categorical variables in this way can help reduce the variance of the shap values estimate (1). Note that this does not necessarily result in a runtime saving; by default the algorithm estimates the shap values by creating a training dataset for the weighed regression, which consists of tiling nsamples (2) copies of the background dataset. By default, this parameter is set to auto, which is given by \(2^1 + 2^{**}11\) where \(M\) is the number of features which can be perturbed. Therefore, because \(2^1 \times 2^{**} 11\), one should not expect to see significant time savings when reducing the number of columns. The runtime can be improved by reducing nsamples at the cost of a loss in estimation accuracy. (3)

The following arguments should be passed to the fit step in order to perform grouping:

- **background_data**: in this case, X_train_proc
- **group_names**: a list containing the feature names
- **groups**: for each feature name in group_name, groups contains a list of column indices in X_train_proc which represent that feature.

```python
def make_groups(num_feats_names, cat_feats_names, feat_enc_dim):
    """
    Given a list with numerical feat. names, categorical feat. names and a list specifying the lengths of the encoding for each cat. variable, the function outputs a list of group names, and a list of the same len where each entry represents the column indices that the corresponding categorical feature
    """
    group_names = num_feats_names + cat_feats_names
    groups = []
    cat_var_idx = 0
    for name in group_names:
        if name in num_feats_names:
            groups.append(list(range(len(groups), len(groups) + 1)))
        else:
            start_idx = groups[-1][-1] + 1 if groups else 0
            groups.append(list(range(start_idx, start_idx + feat_enc_dim[cat_var_idx])))
        cat_var_idx += 1
    return group_names, groups

def sparse2ndarray(mat, examples=None):
    """
    Converts a scipy.sparse.csr.csr_matrix to a numpy.ndarray.
    If specified, examples is slice object specifying which selects a number of rows from mat and converts only the respective slice.
    """
    if examples:
        return mat[examples, :].toarray()
    return mat.toarray()
```

32.5. Apply KernelSHAP to explain the model
Having created the groups, we are now ready to instantiate the explainer and explain our set.

```python
X_train_proc_d = sparse2ndarray(X_train_proc, examples=background_data)
group_names, groups = make_groups(num_feats_names, cat_feats_names, feat_enc_dim)
```

```python
Having created the groups, we are now ready to instantiate the explainer and explain our set.

```python
X_explain_proc_d = sparse2ndarray(X_explain_proc)
grp_lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_names)
grp_lr_explainer.fit(X_train_proc_d, group_names=group_names, groups=groups)
```

```python
[ ]: grouped_explanation = grp_lr_explainer.explain(X_explain_proc_d)
```

```python
[50]: shap.summary_plot(grouped_explanation.shap_values[0], perm_X_explain, perm_feat_names)
```

Having ranked the features by the average magnitude of their shap value, we can now see if they provide the same ranking. Yet another way to deal with the categorical variables is to fit the explainer to the unprocessed dataset and combine the preprocessor with the predictor. We show this approach yields the same results in this example.

```python
def compare_ranking(ranking_1, ranking_2, methods=None):
    for i, (combined, grouped) in enumerate(zip(ranking_1, ranking_2)):
        print(f"Class: {i}")
        c_names, g_names = combined[1], grouped[1]
        c_mag, g_mag = combined[0], grouped[0]
        different = []
        for i, (c_n, g_n) in enumerate(zip(c_names, g_names)):
            if c_n != g_n:
                different.append((i, c_n, g_n))
```

(continues on next page)
if different:
    method_1 = methods[0] if methods else "Method_1"
    method_2 = methods[1] if methods else "Method_2"
    i, c_ns, g_ns = list(zip(*different))
    data = {"Rank": i, method_1: c_ns, method_2: g_ns}
    df = pd.DataFrame(data=data)
    print("Found the following rank differences:")
    print(df)
else:
    print("The methods provided the same ranking for the feature effects.")
    print(f"The ranking is: {c_names}")
    print('')

compare_ranking(ranked_combined_shap_vals, ranked_grouped_shap_vals)

Class: 0
The methods provided the same ranking for the feature effects.
The ranking is: ['Marital Status', 'Education', 'Capital Gain', 'Occupation', 'Sex',
    'Relationship', 'Age', 'Hours per week', 'Workclass', 'Capital Loss', 'Country',
    'Race']

Class: 1
The methods provided the same ranking for the feature effects.
The ranking is: ['Marital Status', 'Education', 'Capital Gain', 'Occupation', 'Sex',
    'Relationship', 'Age', 'Hours per week', 'Workclass', 'Capital Loss', 'Country',
    'Race']

As shown in this example, for a logistic regression model, the exact shap values can be computed as shown below. Note that, like KernelShap, this computation makes the assumption that the features are independent.

```python
[67]:
    exact_shap = [(beta[:, None, :]*X_explain_proc_d)[i, ...] for i in range(beta.shape[0])]
    combined_exact_shap = [sum_categories(shap_values, cat_feat_start, feat_enc_dim) for shap_values in exact_shap]
    ranked_combined_exact_shap = [rank_features(vals, perm_feat_names) for vals in combined_exact_shap]

[58]:
    shap.summary_plot(combined_exact_shap[0], perm_X_explain, perm_feat_names )
```
Comparing the two summary plots above, we notice that albeit the estimation and the exact method rank the features Marital Status, Education and Capital Gain as the features that are most important for the classification decision, the ranking of the remainder of the features differs. In particular, while Race is estimated to be the sixth more important feature using the exact shap value computation, it is deemed as the least important in the approximate computation. However, note that the exact shap value calculation takes into account the weight estimated by the logistic regression model. All the weights in the model are estimated jointly so that the model predictive distribution matches the predictive distribution of the training data. Thus, the values of the coefficients are a function of the entire dataset. On the other hand, to limit the computation time, the shap values are estimated using a small background dataset. This error is compounded by the fact that the estimation is approximate, since computing the exact values using the weighted regression has exponential computational complexity. Below, we show that the Race feature distribution is heavily skewed towards white individuals. Investigating correcting this imbalance would lead to more accurate estimation is left to future work.

```python
[69]: from functools import partial
    from collections import Counter

[70]: def get_feature_distribution(dataset, feature, category_map, feature_names):
      
      """Given a map of categorical variable indices to human-readable values and an array of feature integer values, the function outputs the distribution the feature in human readable format.""

      feat_mapping = category_map[feature_names.index(feature)]
      distrib_raw = Counter(dataset)
      distrib = {feat_mapping[key]: val for key, val in distrib_raw.items()}

      return distrib

[71]: get_distribution = partial(get_feature_distribution, feature_names=feature_names, _
   └→ category_map=category_map)
```

(continues on next page)
race_idx = feature_names.index("Race")
bkg_race_distrib = get_distribution(X_train[background_data, race_idx], 'Race')
train_race_distrib = get_distribution(X_train[:, race_idx], 'Race')
expl_race_distrib = get_distribution(X_explain[:, race_idx], 'Race')

[72]: print("Background data race distribution:")
print(bkg_race_distrib)
print("Train data race distribution:")
print(train_race_distrib)
print("Explain race distribution:")
print(expl_race_distrib)

Background data race distribution:
{'White': 89, 'Amer-Indian-Eskimo': 2, 'Black': 8, 'Asian-Pac-Islander': 1}
Train data race distribution:
{'White': 25634, 'Amer-Indian-Eskimo': 285, 'Black': 2868, 'Asian-Pac-Islander': 963, 'Other': 250}
Explain race distribution:
{'White': 105, 'Black': 20, 'Asian-Pac-Islander': 2, 'Amer-Indian-Eskimo': 1}

We now look to compare the approximate and the exact shap values as well as the relation between the shap computation and the logistic regression coefficients.

[372]: def reorder_feats(vals_and_names, src_vals_and_names):
    """Given a two tuples, each containing a list of ranked feature
    shap values and the corresponding feature names, the function
    reorders the values in vals according to the order specified in
    the list of names contained in src_vals_and_names.
    """
    _, src_names = src_vals_and_names
    vals, names = vals_and_names
    reordered = np.zeros_like(vals)
    for i, name in enumerate(src_names):
        alt_idx = names.index(name)
        reordered[i] = vals[alt_idx]
    return reordered, src_names

def compare_avg_mag_shap(class_idx, comparisons, baseline, **kwargs):
    """Given a list of tuples, baseline, containing the feature values and a list with
    -feature names
    for each class and, comparisons, a list of lists with tuples with the same
    -structure, the
    function reorders the values of the features in comparisons entries according to
    -the order
    of the feature names provided in the baseline entries and displays the feature
    values for comparison.
    """
    methods = kwargs.get("methods", [f"method_{i}" for i in range(len(comparisons) + 1)])
    n_features = len(baseline[class_idx][0])

(continues on next page)
# bar settings
bar_width = kwargs.get("bar_width", 0.05)
bar_space = kwargs.get("bar_space", 2)

# x axis
x_low = kwargs.get("x_low", 0.0)
x_high = kwargs.get("x_high", 1.0)
x_step = kwargs.get("x_step", 0.05)
x_ticks = np.round(np.arange(x_low, x_high + x_step, x_step), 3)

# y axis (these are the y coordinate of start and end of each group of bars)
start_y_pos = np.array(np.arange(0, n_features))*bar_space
end_y_pos = start_y_pos + bar_width*len(methods)
y_ticks = 0.5*(start_y_pos + end_y_pos)

# figure
fig_x = kwargs.get("fig_x", 10)
fig_y = kwargs.get("fig_y", 7)

# fontsizes
title_font = kwargs.get("title_fontsize", 20)
legend_font = kwargs.get("legend_fontsize", 20)
tick_labels_font = kwargs.get("tick_labels_fontsize", 20)
axes_label_fontsize = kwargs.get("axes_label_fontsize", 10)

# labels
title = kwargs.get("title", None)
ylabel = kwargs.get("ylabel", None)
xlabel = kwargs.get("xlabel", None)

# process input data
methods = list(reversed(methods))
base_vals = baseline[class_idx][0]
ordering = baseline[class_idx][1]
comp_vals = []

# reorder the features so that they match the order of the baseline (ordering)
for comparison in comparisons:
    vals, ord_ = reorder_feats(comparison[class_idx], baseline[class_idx])
    comp_vals.append(vals)
    assert ord_ is ordering

all_vals = [base_vals] + comp_vals
data = dict(zip(methods, all_vals))
df = pd.DataFrame(data=data, index=ordering)

# plotting logic
fig, ax = plt.subplots(figsize=(fig_x, fig_y))
for i, col in enumerate(df.columns):
    values = list(df[col])
    y_pos = [y + bar_width*i for y in start_y_pos]
    ax.barh(y_pos, list(values), bar_width, label=col)

# add ticks, legend and labels
ax.set_xticks(x_ticks)
ax.set_xticklabels([str(x) for x in x_ticks], rotation=45, fontsize=tick_labels_font)
ax.set_xlabel(xlabel, fontsize=axes_label_fontsize)
ax.set_yticks(y_ticks)
ax.set_yticklabels(ordering, fontsize=tick_labels_font)
ax.set_ylabel(ylabel, fontsize=axes_label_fontsize)
ax.invert_yaxis()  # labels read top-to-bottom
ax.legend(fontsize=legend_font)
plt.grid(True)
plt.title(title, fontsize=title_font)
return ax, fig, df

[356]:
class_idx = 0
ax, fig, _ = compare_avg_mag_shap(class_idx,
    ranked_combined_shap_vals,    
    ranked_combined_exact_shap,    
    methods=('approximate', 'exact'),
    bar_width=0.5,
    tick_labels_fontsize=12,      
    legend_fontsize=12,            
    title="Comparison between exact and approximate feature effects",
    xlabel="Features effects (class {})".format(0),
    ylabel="Feature",
    axes_label_fontsize=15,
)
As before, we see that features such as Occupation, Workclass or Race have similar effects according to the ranking of the logistic regression coefficients and that the exact shap value estimation recovers this effect since it is computed using the underlying coefficients. Unlike in our previous example, these relationships are not recovered by the approximate estimation procedure. Therefore, whenever possible, exact shap value computation should be preferred to approximations. As shown in this example it is possible to calculate exact shap values for linear models.
and exact algorithms exist for tree models. The approximate procedure still gives insights into the model, but, as shown above, it can be quite sensitive when the effects of the variables are similar. The notable differences between the two explanations are the importance of the Race and Country are underestimated by a significant margin and their rank significantly differs from the exact computation.

Finally, as noted in [4] as the model bias (7) increases, more weight can be assigned to irrelevant features. This is perhaps expected since a linear model will suffer from bias when applied to data generated from a nonlinear process, so we don’t expect the feature effects to be accurately estimated. This also affects the exact shap values, which depend on these coefficients.

### 32.5.3 Investigating the feature effects given a range of feature values

Given an individual record, one could ask questions of the type *What would have been the effect of feature x had its value been y?*. To answer this question one can create hypothetical instances starting from a base record, where the hypothetical instances have a different value for a chosen feature than the original record. Below, we study the effect of the Capital Gain feature as a function of its value. We choose the 0th record in the X_explain set, which represents an individual with no capital gain.

```python
[76]:
    idx = 0
    base_record = X_explain[idx, ]
    cap_gain = X_explain[idx, feature_names.index('Capital Gain')]
    print(f"The capital gain of individual {idx} is {cap_gain}!")

The capital gain of individual 0 is 0!
```

We now create a dataset of records that differ from a base record only by the Capital Gain feature.

```python
[77]:
    cap_increment = 100
    cap_range = range(0, 10100, cap_increment)
    hyp_record = np.repeat(base_record[None, :], len(cap_range), axis=0)
    hyp_record[:, feature_names.index('Capital Gain')] = cap_range
    assert (hyp_record[1, :] - hyp_record[0, :]).sum() == cap_increment
    X_hyp_proc = preprocessor.transform(hyp_record)
    X_hyp_proc_d = X_hyp_proc.toarray()
```

We can explain the hypothetical instances in order to understand the change in the Capital Gain effect as a function of its value.

```python
[ ]: hyp_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_names)
    hyp_explainer.fit(X_train_proc_d, group_names=group_names, groups=groups)
    hyp_explanation = hyp_explainer.explain(X_hyp_proc_d)

[79]: hyp_record_perm, _ = permute_columns(hyp_record, feature_names, perm_feat_names)

[80]: shap.dependence_plot('Capital Gain',
                        hyp_explanation.shap_values[1],
                        hyp_record_perm,
                        feature_names=perm_feat_names,
                        interaction_index=None,
                        )
```

32.5. Apply KernelSHAP to explain the model 237
In a logistic regression model, the predictors are linearly related to the logits. Estimating the shap values using the KernelShap clearly recovers this aspect, as shown by the plot above. The dependence of the feature effect on the feature value has important implications on the shap value estimation; since the model relies on using the background dataset to simulate the effect of missing inputs in order to estimate any feature effect, it is important to select an appropriate background dataset in order to avoid biasing the estimate of the feature effect of interest. Below, we will experiment with the size of the background dataset, split from the training set of the classifier while keeping the class represensation proportions of the training set roughly the same.

An alternative way to display the effect of a value as a function of the feature value is to group the similar prediction paths, which can be done by specifying the `hclust` feature ordering option.

```python
# obtain the human readable version of the base record (for display purposes)
base_perm, perm = permute_columns(base_record[None, :], feature_names, perm_feat_names)
br = []
for i, x in enumerate(np.nditer(base_record.squeeze())):
    if i in categorical_features:
        br.append(category_map[i][x])
    else:
        br.append(x.item())
br = [br[i] for i in perm]
df = pd.DataFrame(data=np.array(br).reshape(1, -1), columns=perm_feat_names)
```

```python
# r = shap.decision_plot(hyp_explainer.expected_value[1],
# hyp_explanation.shap_values[1][0:-1:5],
# hyp_record_perm,
```

(continues on next page)
The decision plot above informs us of the path to the decision $\text{Income} < 50,000$ for the original record (depicted in blue, and, for clarity, on its own below). Additionally, decision paths for fictitious records where only the Capital Gain feature was altered are displayed. For clarity, only a handful of these instances have been plotted. Note that the base value of the plot has been altered to be the classification threshold \(^6\) as opposed to the expected prediction probability for individuals earning more than $50,000.

We see that the second highlighted instance (in purple) would have been predicted as making an income over $50,000 with approximately 0.6 probability, and that this change in prediction is largely driven by the Capital Gain feature. We can see below that the income predictor would have predicted the income of this individual to be more than $50,000 had the Capital Gain been over $3,500.

```python
# the 7th record from the filtered ones would be predicted to make an income > 50k
income_pred_probas = pred_fcn(preprocessor.transform(hyp_record[0:-1:5][7, :][None, :]))
print(f"Prediction probabilities: {income_pred_probas}")
# we can see that the minimum capital gain for the prediction to change is: $3,500
cap_gain_min = hyp_record[0:-1:5][7, feature_names.index('Capital Gain')]
```
print(f"Minimum capital gain is: ${cap_gain_min}"

Prediction probabilities: [[0.49346669 0.50653331]]
Minimum capital gain is: $3500

[88]: shap.decision_plot(hyp_explainer.expected_value[1],
                       hyp_explanation.shap_values[1][1],
                       df,
                       link='logit',
                       feature_order=r.feature_idx,
                       highlight=0
)

Note that passing `return_objects=True` and using the `r.feature_idx` as an input to the decision plot above we were able to plot the original record along with the feature values in the same feature order. Additionally, by passing `logit` to the plotting function, the scale of the axis is mapped from the margin to probability space\(^5\).

Combined, the two decision plots show that:

- the largest decrease in the probability of earning more than $50,000 is significantly affected if the individual is **Never-Married**
- the largest increase in the probability of earning more than $50,000 is determined by the education level
- the probability of making an income greater than $50,000 increases with the capital gain; notice how this implies that features such as **Education** or **Occupation** also contribute more to the increase in probability of earning more than $50,000
### 32.5.4 Checking if prediction paths significantly differ for extreme probability predictions

One can employ the decision plot to check if the prediction paths for low (or high) probability examples differ significantly; conceptually, examples which exhibit prediction paths which are significantly different are potential outliers.

Below, we seek to explain only those examples which are predicted to have an income above $50,000 with small probability.

```python
[89]: predictions = classifier.predict_proba(X_explain_proc)
    low_prob_idx = np.logical_and(predictions[:, 1] <= 0.1, predictions[:, 1] >= 0.03)
    X_low_prob = X_explain_proc[low_prob_idx, :]

[ ]: low_prob_explanation = hyp_explainer.explain(X_low_prob.toarray())

[92]: X_low_prob_perm, _ = permute_columns(X_explain[low_prob_idx, :], feature_names, perm_feat_names)

shap.decision_plot(hyp_explainer.expected_value[1],
    low_prob_explanation.shap_values[1],
    X_low_prob_perm,
    feature_names=perm_feat_names,
    feature_order='hclust')
```

From the above plot, we see that the prediction paths for the samples with low probability of being class 1 are similar - no potential outliers are identified.
32.5.5 Investigating the effect of the background dataset size on shap value estimates

The shap values estimation relies on querying the model with samples where certain inputs are toggled off in order to infer the contribution of a particular feature. Since most models cannot accept arbitrary patterns of missing values, the background dataset is used to replace the values of the missing features, that is, as a background model. In more detail, the algorithm creates first a number of copies of this dataset, and then subsamples sets of

Since the model predicts on these perturbed samples and regresses on the predictions to infer the shap values, the quality of the background model is key for the explanation model. Here we will not be concerned with modelling the background, but instead investigate whether simply increasing the background set size can give rise to wildly different shap values. This part of the example is long running so the graph showing our original results can be loaded instead.

```python
import pickle

def get_dataset(X_train, y_train, split_fraction):
    """
    Splits and transforms a dataset
    """
    split_X, _ = split_set(X_train, y_train, split_fraction)
    split_X_proc = preprocessor.transform(split_X)
    split_X_proc_d = sparse2ndarray(split_X_proc)
    return split_X_proc_d

Below cell is long running, skip and display the graph instead.

```
We notice that with the exception of the Capital Gain and Capital Loss, the differences between the shap values estimates are not significant as the fraction of the training set used as a background dataset increases from 0.005 to 0.16. Notably, the Capital Gain feature would be ranked as the second most important by the all approximate models, whereas in the initial experiment which used the first 100 (0.003) examples from the training set the ranking of the two features was reversed. How to select an appropriate background dataset is an open ended question. In the future, we will explore whether clustering the training data can provide a more representative background model and increase the accuracy of the estimation.

A potential limitation of expensive explanation methods such as KernelShap when used to draw insights about the
global model behaviour is the fact that explaining large datasets can take a long time. Below, we explain a larger fraction of the testing set (0.4) in order to see if different conclusions about the feature importances would be made.

```
[366]: fraction_explained = 0.4
X_explain_large, y_explain_large = split_set(X_test,
y_test,
    fraction_explained,
)
X_explain_large_proc = preprocessor.transform(X_explain_large)
X_explain_large_proc_d = sparse2ndarray(X_explain_large_proc)

Number of records: 1024
Number of class 0: 763
Number of class 1: 261
```

```
[ ]: data = get_dataset(X_train, y_train, 0.08)
explainer = KernelShap(pred_fcn, link='logit')
explainer.fit(data, group_names=group_names, groups=groups)
explanation_large_dataset = explainer.explain(X_explain_large_proc_d)
ranked_avg_shap_l = get_ranked_values(explanation_large_dataset)
```

```
[370]: class_idx = 0  # income below $50,000
exact_shap_large = [(beta[ :, None, :]*X_explain_large_proc_d[i, ...] for i in ...
    range(beta.shape[0]))]
combined_exact_shap_large = [sum_categories(shap_values, cat_feat_start, feat_enc_ ...
    —dim) for shap_values in exact_shap_large]
ranked_combined_exact_shap_large = [rank_features(shap_values, perm_feat_names) for_ ...
    —shap_values in combined_exact_shap_large]
```

```
[383]: comparisons = [ranked_combined_exact_shap]
methods = ['exact_large', 'exact_small']
_, fg, df = compare_avg_mag_shap(class_idx,
    comparisons,
    ranked_combined_exact_shap_large,
    methods=methods,
    bar_width=0.5,
    legend_fontsize=15,
    axes_label_fontsize=15,
    tick_labels_fontsize=15,
    title="Comparison of exact shap values inferred from_ ...
    —a small (128) and a large (1024) explanation dataset",
    title_fontsize=15,
    xlabel='Feature effects (class {class_idx})',
    ylabel='Features'
)
```
As expected, the exact shap values have the same ranking when a larger set is explained, since they are derived from the same model coefficients.

```python
[387]: comparisons = [ranked_avg_shap]
methods = ['approx_large', 'approx_small']
_, fg, df = compare_avg_mag_shap(class_idx,
    comparisons,
    ranked_avg_shap_l,
    methods=methods,
    bar_width=0.5,
    legend_fontsize=15,
    axes_label_fontsize=15,
    tick_labels_fontsize=15,
    title="Comparison of approximate shap values inferred from a small (128) and a large (1024) explanation dataset",
    title_fontsize=15,
    xlabel=f'Feature effects (class {class_idx})',
    ylabel='Features'
)
```

32.5. Apply KernelSHAP to explain the model
The ranking of the features also remains unchanged for the approximate method even when significantly more instances are explained.

```python
with open('large_explain_set.pkl', 'wb') as f:
    pickle.dump(
        {'data': data,
         'explainer': explainer,
         'raw_shap': explanation_large_dataset,
         'ranked_shap_vals': ranked_avg_shap_l
        },
        f
    )
```

### Footnotes

1: As detailed in *Theorem 1* in [3], the estimation process for a shap value of feature $i$ from instance $x$ involves taking a weighted average of the contribution of feature $i$ to the model output, where the weighting takes into account all the possible orderings in which the previous and successor features can be added to the set. This computation is thus performed by choosing subsets of features from the full feature set and setting the values of these features to a background value; the prediction on these perturbed samples is used in a least squares objective (*Theorem 2*), weighted by the Shapley kernel. Note that the optimisation objective involves a summation over all possible subsets. Enumerating all the feature subsets has exponential computational cost, so the smaller the feature set, the more samples can be drawn and more accurate shap values can be estimated. Thus, grouping the features can serve to reduce the variance of the shap values estimation by providing a smaller set of features to choose from.

2: This is a kwarg to `shap_values` method.

3: Note the progress bars below show, however, different runtimes between the two methods. No accurate timing analysis was carried out to study this aspect.

4: Note that the shap library currently does not support grouping when the data is represented as a sparse matrix,
so it should be converted to a `numpy.ndarray` object, both during explainer initialisation and when calling the `shap_values` method.

(5): When `link='logit'` is passed to the plotting function, the model outputs are scaled to the probability space, so the inverse logit transformation is applied to the data and axis ticks. This is in contrast to passing `link='logit'` to the KernelExplainer, which maps the model output through the forward logit transformation, \( \log \left( \frac{p}{1-p} \right) \).

(6): We could alter the base value by specifying the `new_base_value` argument to `shap.decision_plot`. Note that this argument has to be specified in the same units as the explanation - if we explained the instances in margin space then to switch the base value of the plot to, say, \( p=0.4 \) then we would pass `new_base_value = \log(0.4/(1 - 0.4))` to the plotting function.

(7): In this context, bias refers to the bias-variance tradeoff; a simpler model will likely incur a larger error during training but will have a smaller generalisation gap compared to a more complex model which will have smaller training error but will generalise poorly.

32.5.7 References


33.1 Introduction

In *this* example, we showed that the categorical variables can be handled by fitting the explainer on preprocessed data and passing preprocessed data to the `explain` call. To handle the categorical variables, we either group them explicitly or sum the estimated shap values for each encoded shap dimension. An alternative way is to define our black-box model to include the preprocessor, as shown in *this* example. We now show that these methods give the same results.

```python
import shap
shap.initjs()
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from alibi.explainers import KernelShap
from alibi.datasets import fetch_adult
from scipy.special import logit
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, plot_confusion_matrix
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

33.2 Data preparation

33.2.1 Load and split

The `fetch_adult` function returns a Bunch object containing the features, the targets, the feature names and a mapping of categorical variables to numbers.

```python
adult = fetch_adult()
adult.keys()
dict_keys(['data', 'target', 'feature_names', 'target_names', 'category_map'])
```
Note that for your own datasets you can use our utility function `gen_category_map` to create the category map.

```python
from alibi.utils.data import gen_category_map
```

```python
np.random.seed(0)
data_perm = np.random.permutation(np.c_[data, target])data = data_perm[:, :-1]target = data_perm[:, -1]
```

```python
idx = 30000X_train, y_train = data[:idx, :], target[:idx]X_test, y_test = data[idx+1:, :], target[idx+1:]```

### 33.2.2 Create feature transformation pipeline

Create feature pre-processor. Needs to have ‘fit’ and ‘transform’ methods. Different types of pre-processing can be applied to all or part of the features. In the example below we will standardize ordinal features and apply one-hot-encoding to categorical features.

**Ordinal features:**

```python
ordinal_features = [x for x in range(len(feature_names)) if x not in list(category_map.keys())]ordinal_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),'scaler', StandardScaler())])
```

**Categorical features:**

```python
categorical_features = list(category_map.keys())
categorical_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),'onehot', OneHotEncoder(drop='first', handle_unknown='error')])
```

Note that in order to be able to interpret the coefficients corresponding to the categorical features, the option `drop='first'` has been passed to the OneHotEncoder. This means that for a categorical variable with n levels, the length of the code will be n−1. This is necessary in order to avoid introducing feature multicolinearity, which would skew the interpretation of the results. For more information about the issue about multicolinearity in the context of linear modelling see [1].

Combine and fit:

```python
preprocessor = ColumnTransformer(transformers=[('num', ordinal_transformer, ordinal_features),('cat', categorical_transformer, categorical_features)])preprocessor.fit(X_train)
```
33.3 Fit a binary logistic regression classifier to the preprocessed Adult dataset

33.3.1 Preprocess the data

```python
[10]: X_train_proc = preprocessor.transform(X_train)
X_test_proc = preprocessor.transform(X_test)
```
33.3.2 Training

```python
[11]: classifier = LogisticRegression(multi_class='multinomial',
                          random_state=0,
                          max_iter=500,
                          verbose=0,
                      )
classifier.fit(X_train_proc, y_train)

[11]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                          intercept_scaling=1, l1_ratio=None, max_iter=500,
                          multi_class='multinomial', n_jobs=None, penalty='l2',
                          random_state=0, solver='lbfgs', tol=0.0001, verbose=0,
                          warm_start=False)
```

33.3.3 Model assessment

```python
[12]: y_pred = classifier.predict(X_test_proc)
[13]: cm = confusion_matrix(y_test, y_pred)
[14]: title = 'Confusion matrix for the logistic regression classifier'
disp = plot_confusion_matrix(classifier,
                              X_test_proc,
                              y_test,
                              display_labels=target_names,
                              cmap=plt.cm.Blues,
                              normalize=None,
                          )
disp.ax_.set_title(title)
[14]: Text(0.5, 1.0, 'Confusion matrix for the logistic regression classifier')
```

![Confusion matrix for the logistic regression classifier](image)

```python
[15]: print('Test accuracy: ', accuracy_score(y_test, classifier.predict(X_test_proc)))
```
33.4 Explaining the model with an explainer fitted on the preprocessed data

To speed up computation, we will use a background dataset with only 100 samples.

```python
start_example_idx = 0
stop_example_idx = 100
background_data = slice(start_example_idx, stop_example_idx)
```

First, we group the categorical variables.

```python
def make_groups(num_feats_names, cat_feats_names, feat_enc_dim):
    """
    Given a list with numerical feat. names, categorical feat. names
    and a list specifying the lengths of the encoding for each cat.
    variable, the function outputs a list of group names, and a list
    of the same len where each entry represents the column indices that
    the corresponding categorical feature
    """

    group_names = num_feats_names + cat_feats_names
    groups = []
    cat_var_idx = 0
    
    for name in group_names:
        if name in num_feats_names:
            groups.append(list(range(len(groups), len(groups) + 1)))
        else:
            start_idx = groups[-1][-1] + 1 if groups else 0
            groups.append(list(range(start_idx, start_idx + feat_enc_dim[cat_var_idx])))
            cat_var_idx += 1
    
    return group_names, groups

def sparse2ndarray(mat, examples=None):
    """
    Converts a scipy.sparse.csr.csr_matrix to a numpy.ndarray.
    If specified, examples is slice object specifying which selects a
    number of rows from mat and converts only the respective slice.
    """

    if examples:
        return mat[examples, :].toarray()
    return mat.toarray()
```

```python
# obtain the indices of the categorical and the numerical features from the pipeline.
numerical_feats_idx = preprocessor.transformers_[0][2]
categorical_feats_idx = preprocessor.transformers_[1][2]
num_feats_names = [feature_names[i] for i in numerical_feats_idx]
cat_feats_names = [feature_names[i] for i in categorical_feats_idx]
```

(continues on next page)
perm_feat_names = num_feats_names + cat_feats_names
ohe = preprocessor.transformers_[1][1].named_steps['onehot']
feat_enc_dim = [len(cat_enc) - 1 for cat_enc in ohe.categories_]

[19]: # create the groups
X_train_proc_d = sparse2ndarray(X_train_proc, examples=background_data)
group_names, groups = make_groups(num_feats_names, cat_feats_names, feat_enc_dim)

Having created the groups, we are now ready to instantiate the explainer and explain our set.

[20]: pred_fcn = classifier.predict_proba
grp_lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_names,
                                 seed=0)
grp_lr_explainer.fit(X_train_proc_d, group_names=group_names, groups=groups)

[20]: KernelShap(meta={'name': 'KernelShap', 'type': 'blackbox', 'explanations': ['local',
                                                                            'global']}, 'params': {'groups': [[0], [1], [2], [3], [4, 5, 6, 7, 8, 9, 10, 11],
                                                                            [12, 13, 14, 15, 16, 17], [18, 19, 20], [21, 22, 23, 24, 25, 26, 27, 28], [29, 30,
                                                                            [31, 32, 33], [34, 35, 36, 37], [38], [39, 40, 41, 42, 43, 44, 45, 46, 47, 48]],
                                                                            'group_names': ['Age', 'Capital Gain', 'Capital Loss', 'Hours per week', 'Workclass',
                                                                            'Education', 'Marital Status', 'Occupation', 'Relationship', 'Race', 'Sex',
                                                                            'Country'], 'weights': None, 'summarise_background': False})

We select only a small fraction of the testing set to explain for the purposes of this example.

[21]: def split_set(X, y, fraction, random_state=0):
    
    """Given a set X, associated labels y, split\s a fraction y from X.  """
    _, X_split, _, y_split = train_test_split(X,
                                            y,
                                            test_size=fraction,
                                            random_state=random_state,
                                            )
    print("Number of records: ").format(X_split.shape[0]))
    print("Number of class 0: ").format(0, len(y_split) - y_split.sum())
    print("Number of class 1: ").format(1, y_split.sum())
    return X_split, y_split

[22]: fraction_explained = 0.01
X_explain, y_explain = split_set(X_test,
                                 y_test,
                                 fraction_explained,
                                 )
X_explain_proc = preprocessor.transform(X_explain)
X_explain_proc_d = sparse2ndarray(X_explain_proc)

Number of records: 26
Number of class 0: 20
Number of class 1: 6

[ ]: grouped_explanation = grp_lr_explainer.explain(X_explain_proc_d)
33.4.1 Explaining with an explainer fitted on the raw data

To explain with an explainer fitted on the raw data, we make the preprocessor part of the predictor, as shown below.

```python
[24]: pred_fcn = lambda x: classifier.predict_proba(preprocessor.transform(x))
lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=feature_names, seed=0)
```

We use the same background dataset to fit the explainer.

```python
[25]: lr_explainer.fit(X_train[background_data])
```

```python
[25]: KernelShap(meta={'name': 'KernelShap', 'type': 'blackbox', 'explanations': ['local', 'global'], 'params': {'groups': None, 'group_names': None, 'weights': None, 'summarise_background': False}})
```

We explain the same dataset as before.

```python
[ ]: explanation = lr_explainer.explain(X_explain)
```

33.4.2 Results comparison

To show that fitting the explainer on the raw data and combining the preprocessor with the classifier gives the same results as grouping the variables and fitting the explainer on the preprocessed data, we check to see that the same features are considered as most important when combining the two approaches.

```python
[27]: def get_ranked_values(explanation):
    """
    Retrives a tuple of (feature_effects, feature_names) for each class explained. A feature's effect is its average shap value magnitude across an array of instances.
    ""
    ranked_shap_vals = []
    for cls_idx in range(len(explanation.shap_values)):
        this_ranking = {
            explanation.raw['importances'][str(cls_idx)]['ranked_effect'],
            explanation.raw['importances'][str(cls_idx)]['names']
        )
        ranked_shap_vals.append(this_ranking)
    return ranked_shap_vals

def compare_ranking(ranking_1, ranking_2, methods=None):
    for i, (combined, grouped) in enumerate(zip(ranking_1, ranking_2)):
        print(f"Class: {i}")
        c_names, g_names = combined[1], grouped[1]
        c_mag, g_mag = combined[0], grouped[0]
        different = []
        for i, (c_n, g_n) in enumerate(zip(c_names, g_names)):
            if c_n != g_n:
                different.append((i, c_n, g_n))
        if different:
            method_1 = methods[0] if methods else "Method_1"
            method_2 = methods[1] if methods else "Method_2"
            i, c_ns, g_ns = list(zip(*different))
            data = ("Rank": i, method_1: c_ns, method_2: g_ns)
```

(continues on next page)
df = pd.DataFrame(data=data)
print("Found the following rank differences:")
print(df)
else:
    print("The methods provided the same ranking for the feature effects.")
    print(f"The ranking is: {c_names}")
    print(

```python
def reorder_feats(vals_and_names, src_vals_and_names):
    """Given a two tuples, each containing a list of ranked feature
    shap values and the corresponding feature names, the function
    reorders the values in vals according to the order specified in
    the list of names contained in src_vals_and_names.
    ""

    _, src_names = src_vals_and_names
    vals, names = vals_and_names
    reordered = np.zeros_like(vals)

    for i, name in enumerate(src_names):
        alt_idx = names.index(name)
        reordered[i] = vals[alt_idx]

    return reordered, src_names
```

```python
def compare_avg_mag_shap(class_idx, comparisons, baseline, **kwargs):
    """Given a list of tuples, baseline, containing the feature values and a list with
    feature names for each class and, comparisons, a list of lists with tuples with the same
    structure, the function reorders the values of the features in comparisons entries according to
    the order of the feature names provided in the baseline entries and displays the feature
    values for comparison.
    ""

    methods = kwargs.get("methods", [f"method_{i}" for i in range(len(comparisons) + 1))])

    n_features = len(baseline[class_idx][0])

    # bar settings
    bar_width = kwargs.get("bar_width", 0.05)
    bar_space = kwargs.get("bar_space", 2)

    # x axis
    x_low = kwargs.get("x_low", 0.0)
    x_high = kwargs.get("x_high", 1.0)
    x_step = kwargs.get("x_step", 0.05)
    x_ticks = np.round(np.arange(x_low, x_high + x_step, x_step), 3)

    # y axis (these are the y coordinate of start and end of each group
    # of bars)
    start_y_pos = np.array(np.arange(0, n_features)) * bar_space
    end_y_pos = start_y_pos + bar_width * len(methods)
    y_ticks = 0.5 * (start_y_pos + end_y_pos)
```

(continues on next page)
# figure
fig_x = kwargs.get("fig_x", 10)
fig_y = kwargs.get("fig_y", 7)

# fontsizes
title_font = kwargs.get("title_fontsize", 20)
legend_font = kwargs.get("legend_fontsize", 20)
tick_labels_font = kwargs.get("tick_labels_fontsize", 20)
axes_label_fontsize = kwargs.get("axes_label_fontsize", 10)

# labels
title = kwargs.get("title", None)
ylabel = kwargs.get("ylabel", None)
xlabel = kwargs.get("xlabel", None)

# process input data
methods = list(reversed(methods))
base_vals = baseline[class_idx][0]
ordering = baseline[class_idx][1]
comp_vals = []

# reorder the features so that they match the order of the baseline (ordering)
for comparison in comparisons:
    vals, ord_ = reorder_feats(comparison[class_idx], baseline[class_idx])
    comp_vals.append(vals)
    assert ord_ is ordering

all_vals = [base_vals] + comp_vals
data = dict(zip(methods, all_vals))
df = pd.DataFrame(data=data, index=ordering)

# plotting logic
fig, ax = plt.subplots(figsize=(fig_x, fig_y))

for i, col in enumerate(df.columns):
    values = list(df[col])
    y_pos = [y + bar_width*i for y in start_y_pos]
    ax.barh(y_pos, list(values), bar_width, label=col)

# add ticks, legend and labels
ax.set_xticks(x_ticks)
ax.set_xticklabels([str(x) for x in x_ticks], rotation=45, fontsize=tick_labels_font)
ax.set_xlabel(xlabel, fontsize=axes_label_fontsize)
ax.set_ylabel(ylabel, fontsize=axes_label_fontsize)
ax.invert_yaxis()  # labels read top-to-bottom
ax.legend(fontsize=legend_font)

plt.grid(True)
plt.title(title, fontsize=title_font)

return ax, fig, df
The methods provided the same ranking for the feature effects. The ranking is: ['Marital Status', 'Capital Gain', 'Education', 'Occupation', 'Sex', 'Relationship', 'Age', 'Hours per week', 'Workclass', 'Capital Loss', 'Country', 'Race']

The methods provided the same ranking for the feature effects. The ranking is: ['Marital Status', 'Capital Gain', 'Education', 'Occupation', 'Sex', 'Relationship', 'Age', 'Hours per week', 'Workclass', 'Capital Loss', 'Country', 'Race']

Above we can see that both methods returned the same feature importances.
We can see that the shap values are very similar. The differences appear because the regression dataset generated in order to compute the shap values differs slightly between the two runs due to the difference in the order of the features in the background dataset.

33.4.3 References

34.1 Introduction

In this example, KernelSHAP is used to explain a batch of instances on multiple cores. To run this example, please run `pip install alibi[ray]` first.

```python
import pprint
import shap
import ray
shap.initjs()

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from alibi.explainers import KernelShap
from alibi.datasets import fetch_adult
from collections import defaultdict
from scipy.special import logit
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, plot_confusion_matrix
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.pipeline import Pipeline
from typing import Dict, List, Tuple

<IPython.core.display.HTML object>
```

34.2 Data preparation

34.2.1 Load and split

The `fetch_adult` function returns a `Bunch` object containing the features, the targets, the feature names and a mapping of categorical variables to numbers.

```python
adult = fetch_adult()
adult.keys()
```
Note that for your own datasets you can use our utility function `gen_category_map` to create the category map.

```python
from alibi.utils.data import gen_category_map
```

```python
np.random.seed(0)
data_perm = np.random.permutation(np.c_[data, target])
data = data_perm[:,:-1]
target = data_perm[:,-1]
```

### 34.2.2 Create feature transformation pipeline

Create feature pre-processor. Needs to have ‘fit’ and ‘transform’ methods. Different types of pre-processing can be applied to all or part of the features. In the example below we will standardize ordinal features and apply one-hot-encoding to categorical features.

**Ordinal features:**

```python
ordinal_features = [x for x in range(len(feature_names)) if x not in list(category_map.keys())]
ordinal_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
                                        ('scaler', StandardScaler())])
```

**Categorical features:**

```python
categorical_features = list(category_map.keys())
categorical_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
                                          ('onehot', OneHotEncoder(drop='first',
                                          handle_unknown='error'))])
```

Note that in order to be able to interpret the coefficients corresponding to the categorical features, the option drop='first' has been passed to the OneHotEncoder. This means that for a categorical variable with n levels, the length of the code will be n-1. This is necessary in order to avoid introducing feature multicolinearity, which would skew the interpretation of the results. For more information about the issue about multicolinearity in the context of linear modelling see [1].

Combine and fit:

```python
preprocessor = ColumnTransformer(transformers=[('num', ordinal_transformer, ordinal_features),
                                               ('cat', categorical_transformer, categorical_features)])
preprocessor.fit(X_train)
```
34.2.3 Preprocess the data

```
X_train_proc = preprocessor.transform(X_train)
X_test_proc = preprocessor.transform(X_test)
```

Applying the `sklearn` processing pipeline modifies the column order of the original dataset. The new feature ordering is necessary in order to correctly plot visualisations, and is inferred from the `preprocessor` object below:

```
numerical_feats_idx = preprocessor.transformers_[0][2]
categorical_feats_idx = preprocessor.transformers_[1][2]
scaler = preprocessor.transformers_[0][1].named_steps['scaler']
um_feats_names = [feature_names[i] for i in numerical_feats_idx]
cat_feats_names = [feature_names[i] for i in categorical_feats_idx]
perm_feat_names = num_feats_names + cat_feats_names
```

```
Original order:
['Age',
 'Workclass',
 'Education',
 'Marital Status',
 'Occupation',
 'Relationship',
 'Race',
 'Sex',
 'Capital Gain',
 'Capital Loss',
 'Hours per week',
 'Country']

New features order:
['Age',
 'Capital Gain',
 'Capital Loss',
 'Hours per week',
 'Workclass',
 'Education',

(continues on next page)
Create a utility to reorder the columns of an input array so that the features have the same ordering as that induced by the preprocessor.

```python
[13]:
def permute_columns(X: np.ndarray, feat_names: List[str], perm_feat_names: List[str]) -> np.ndarray:
    ""
    Permutes the original dataset so that its columns (ordered according to feat_names) have the order of the variables after transformation with the sklearn preprocessing pipeline (perm_feat_names).
    ""
    perm_X = np.zeros_like(X)
    perm = []
    for i, feat_name in enumerate(perm_feat_names):
        feat_idx = feat_names.index(feat_name)
        perm_X[:, i] = X[:, feat_idx]
        perm.append(feat_idx)
    return perm_X, perm
```

The categorical variables will be grouped to reduce shap values variance, as shown in this example. To do so, the dimensionality of each categorical variable is extracted from the preprocessor:

```python
[14]:
# get feature names for the encoded categorical features
ohe = preprocessor.transformers_[1][1].named_steps['onehot']
fts = [feature_names[x] for x in categorical_features]
cat_enc_names = ohe.get_feature_names(fts)
    # compute encoded dimension; -1 as ohe is setup with drop='first'
feat_enc_dim = [len(cat_enc) - 1 for cat_enc in ohe.categories_]
d = {'feature_names': fts, 'encoded_dim': feat_enc_dim}
df = pd.DataFrame(data=d)
print(df)
total_dim = df['encoded_dim'].sum()
print("The dimensionality of the encoded categorical features is ".format(total_dim))
assert total_dim == len(cat_enc_names)
```

```
<table>
<thead>
<tr>
<th>feature_names</th>
<th>encoded_dim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workclass</td>
<td>8</td>
</tr>
<tr>
<td>Education</td>
<td>6</td>
</tr>
<tr>
<td>Marital Status</td>
<td>3</td>
</tr>
<tr>
<td>Occupation</td>
<td>8</td>
</tr>
<tr>
<td>Relationship</td>
<td>5</td>
</tr>
<tr>
<td>Race</td>
<td>4</td>
</tr>
<tr>
<td>Sex</td>
<td>1</td>
</tr>
<tr>
<td>Country</td>
<td>10</td>
</tr>
</tbody>
</table>
```

The dimensionality of the encoded categorical features is 45.
### 34.2.4 Select a subset of test instances to explain

```python
[15]:
def split_set(X, y, fraction, random_state=0):
    
    """Given a set X, associated labels y, splits a fraction y from X."
    """
    _, X_split, _, y_split = train_test_split(X, y,
                                            test_size=fraction,
                                            random_state=random_state,
                                           
    print("Number of records: {}").format(X_split.shape[0]))
    print("Number of class 0: ").format(0, len(y_split) - y_split.sum())
    print("Number of class 1: ").format(1, y_split.sum())
    
    return X_split, y_split

[16]:
fraction_explained = 0.05
X_explain, y_explain = split_set(X_test,
                                  y_test,
                                  fraction_explained,)
X_explain_proc = preprocessor.transform(X_explain)

Number of records: 128
Number of class 0: 96
Number of class 1: 32
```

Create a version of the dataset to be explained that has the same feature ordering as that of the feature matrix after applying the preprocessing (for plotting purposes).

```python
[17]:
perm_X_explain, _ = permute_columns(X_explain, feature_names, perm_feat_names)
```

### 34.3 Fit a binary logistic regression classifier to the Adult dataset

#### 34.3.1 Training

```python
[18]:
classifier = LogisticRegression(multi_class='multinomial',
                                random_state=0,
                                max_iter=500,
                                verbose=0,
                              
classifier.fit(X_train_proc, y_train)
```

```python
[18]:
LogisticRegression(max_iter=500, multi_class='multinomial', random_state=0)
```
34.3.2 Model assessment

```python
[19]: y_pred = classifier.predict(X_test_proc)
[20]: cm = confusion_matrix(y_test, y_pred)
[21]: title = 'Confusion matrix for the logistic regression classifier'
disp = plot_confusion_matrix(classifier,
   X_test_proc,
   y_test,
   display_labels=target_names,
   cmap=plt.cm.Blues,
   normalize=None,
)
disp.ax_.set_title(title)
```

```
Text(0.5, 1.0, 'Confusion matrix for the logistic regression classifier')
```

```
[22]: print('Test accuracy: ', accuracy_score(y_test, classifier.predict(X_test_proc)))
    Test accuracy:  0.8546875
```

34.4 Running KernelSHAP in sequential mode

A background dataset is selected.

```python
[23]: start_example_idx = 0
    stop_example_idx = 100
    background_data = slice(start_example_idx, stop_example_idx)
```

Groups are specified by creating a list where each sublist contains the column indices that a given variable occupies in the preprocessed feature matrix.
[24]:
```python
def make_groups(num_feats_names: List[str], cat_feats_names: List[str], feat_enc_dim: List[int]) -> Tuple[List[str], List[List[int]]]:
    ""
    Given a list with numerical feat. names, categorical feat. names
    and a list specifying the lengths of the encoding for each cat.
    variable, the function outputs a list of group names, and a list
    of the same len where each entry represents the column indices that
    the corresponding categorical feature
    ""
    group_names = num_feats_names + cat_feats_names
    groups = []
    cat_var_idx = 0
    for name in group_names:
        if name in num_feats_names:
            groups.append(list(range(len(groups), len(groups) + 1)))
        else:
            start_idx = groups[-1][-1] + 1 if groups else 0
            groups.append(list(range(start_idx, start_idx + feat_enc_dim[cat_var_idx])))
        cat_var_idx += 1
    return group_names, groups
```

def sparse2ndarray(mat, examples=None):
    ""
    Converts a scipy.sparse.csr.csr_matrix to a numpy.ndarray.
    If specified, examples is slice object specifying which selects a
    number of rows from mat and converts only the respective slice.
    ""
    if examples:
        return mat[examples, :].toarray()
    return mat.toarray()

X_train_proc_d = sparse2ndarray(X_train_proc, examples=background_data)
group_names, groups = make_groups(num_feats_names, cat_feats_names, feat_enc_dim)

Initialise and run the explainer sequentially.

```python
[ ]:
pred_fcn = classifier.predict_proba
seq_lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_names)
seq_lr_explainer.fit(X_train_proc_d[background_data, :], group_names=group_names, groups=groups)
```

[26]:
n_runs = 3

[27]:
s_explanations, s_times = [], []

[ ]:
```python
for run in range(n_runs):
    t_start = timer()
    explanation = seq_lr_explainer.explain(sparse2ndarray(X_explain_proc))
    t_elapsed = timer() - t_start
    s_times.append(t_elapsed)
    s_explanations.append(explanation.shap_values)
```

34.4. Running KernelSHAP in sequential mode
34.5 Running KernelSHAP in distributed mode

The only change needed to distribute the computation is to pass a dictionary containing the number of (physical) CPUs available to distribute the computation to the `KernelShap` constructor:

```
[29]: def distrib_opts_factory(n_cpus: int) -> Dict[str, int]:
    return {'n_cpus': n_cpus}

[30]: cpu_range = range(2, 5)
distrib_avg_times = dict(zip(cpu_range, [0.0]*len(cpu_range)))
distrib_min_times = dict(zip(cpu_range, [0.0]*len(cpu_range)))
distrib_max_times = dict(zip(cpu_range, [0.0]*len(cpu_range)))
d_explanations = defaultdict(list)

[ ]: for n_cpu in cpu_range:
    opts = distrib_opts_factory(n_cpu)
    distrib_lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_˓
→names, distributed_opts=opts)
    distrib_lr_explainer.fit(X_train_proc_d[background_data, :], group_names=group_˓
→names, groups=groups)
    raw_times = []
    for _ in range(n_runs):
        t_start = timer()
        d_explanations[n_cpu].append(distrib_lr_explainer.explain(sparse2ndarray(X_˓
→explain_proc), silent=True).shap_values)
        t_elapsed = timer() - t_start
        raw_times.append(t_elapsed)
    distrib_avg_times[n_cpu] = np.round(np.mean(raw_times), 3)
    distrib_min_times[n_cpu] = np.round(np.min(raw_times), 3)
    distrib_max_times[n_cpu] = np.round(np.max(raw_times), 3)
    ray.shutdown()  
```

34.6 Results analysis

34.6.1 Timing

```
[41]: print(f"Distributed average times for {n_runs} runs (n_cpus: avg_time):")
print(distrib_avg_times)
print(""
print(f"Sequential average time for {n_runs} runs:")
print(np.round(np.mean(s_times), 3), "s")

Distributed average times for 3 runs (n_cpus: avg_time):
{2: 57.197, 3: 41.728, 4: 36.751)
Sequential average time for 3 runs: 119.656 s
```

Running KernelSHAP in a distributed fashion improves the runtime as the results above show. However, the results above should not be interpreted as performance measurements since they were not run in a controlled environment. See our blog post for a more thorough analysis.
34.6.2 Explanations comparison

[33]:
```python
cls = 0  # class of prediction explained
run = 1  # which run to compare the result for
```

[34]:
```python
# sequential
shap.summary_plot(s_explanations[run][cls], perm_X_explain, perm_feat_names)
```

[36]:
```python
# distributed
n_cpu = 3
shap.summary_plot(d_explanations[n_cpu][run][cls], perm_X_explain, perm_feat_names)
```
Comparing the results above one sees that the running the algorithm across multiple cores gave identical results, indicating its correctness.

### 34.7 Conclusion

This example showed that batches of explanations can be explained much faster by simply passing `distributed_opts={'n_cpus': k}` to the `KernelShap` constructor (here `k` is the number of physical cores available). The significant runtime reduction makes it possible to explain larger datasets faster and combine shap values estimated with KernelSHAP into global explanations or use larger background datasets.
35.1 General definition

The model linearity module in alibi provides metric to measure how linear an ML model is. Linearity is defined based on how much the linear superposition of the model’s outputs differs from the output of the same linear superposition of the inputs.

Given $N$ input vectors $v_i$, $N$ real coefficients $\alpha_i$ and a predict function $M(v_i)$, the linearity of the predict function is defined as

$$L = \left\| \sum_i \alpha_i M(v_i) - M\left(\sum_i \alpha_i v_i\right) \right\|$$

If $M$ is a regressor

$$L = \left\| \sum_i \alpha_i \log M(v_i) - \log M\left(\sum_i \alpha_i v_i\right) \right\|$$

If $M$ is a classifier

Note that a lower value of $L$ means that the model $M$ is more linear.

35.2 Alibi implementation

- Based on the general definition above, alibi calculates the linearity of a model in the neighborhood of a given instance $v_0$.

35.3 Iris Data set

- As an example, we will visualize the decision boundaries and the values of the linearity measure for various classifier on the iris dataset. Only 2 features are included for visualization purposes.

```python
import pandas as pd
import numpy as np
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
```

(continues on next page)
from xgboost import XGBClassifier
from itertools import product
from alibi.confidence import linearity_measure, LinearityMeasure

### 35.4 Dataset

```python
[3]:
    ds = load_iris()
    X_train, y_train = ds.data[:, :2], ds.target

[4]:
    lins_dict = {}
```

### 35.5 Models

We will experiment with 5 different classifiers:
* A logistic regression model, which is expected to be highly linear.
* A random forest classifier, which is expected to be highly non-linear.
* An xgboost classifier.
* A support vector machine classifier.
* A feed forward neural network

```python
[5]:
    lr = LogisticRegression(fit_intercept=False, multi_class='multinomial', solver=’newton-cg’)
    rf = RandomForestClassifier(n_estimators=100)
    xgb = XGBClassifier(n_estimators=100)
    svm = SVC(gamma=.1, kernel='rbf', probability=True)
    nn = MLPClassifier(hidden_layer_sizes=(100, 50), activation='relu', max_iter=1000)

[6]:
    lr.fit(X_train, y_train)
    rf.fit(X_train, y_train)
    xgb.fit(X_train, y_train)
    svm.fit(X_train, y_train)
    nn.fit(X_train, y_train);
```

### 35.6 Decision boundaries and linearity

```python
[7]:
    # Creating a grid
    x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
    y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))

[8]:
    # Flattening points in the grid
    X = np.empty((len(xx.flatten()), 2))
    for i in range(xx.shape[0]):
        for j in range(xx.shape[1]):
            k = i * xx.shape[1] + j
            X[k] = np.array([xx[i, j], yy[i, j]])
```
35.6.1 Logistic regression

[9]: # Defining predict function for logistic regression
    clf = lr
    predict_fn = lambda x: clf.predict_proba(x)

[10]: # Calculating linearity for all points in the grid
    lm = LinearityMeasure(agg='pairwise')
    lm.fit(X_train)
    L = lm.score(predict_fn, X)
    L = L.reshape(xx.shape)
    lins_dict['LR'] = L.mean()

[11]: # Visualising decision boundaries and linearity values
    f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(16, 8))
    idx = (0,0)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    axarr[0].contourf(xx, yy, Z, alpha=0.4)
    axarr[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20, edgecolor='k',
    axarr[0].set_title('Decision boundaries', fontsize=20)
    axarr[0].set_xlabel('sepal length (cm)', fontsize=18)
    axarr[0].set_ylabel('sepal width (cm)', fontsize=18)
    LPL = axarr[1].contourf(xx, yy, L, alpha=0.8, cmap='Greys')
    axarr[1].set_title('Model linearity', fontsize=20)
    axarr[1].set_xlabel('sepal length (cm)', fontsize=18)
    axarr[1].set_ylabel('sepal width (cm)', fontsize=18)
    cbar = f.colorbar(LPL)
    plt.show()

print('Decision boundaries (left panel) and linearity measure (right panel) for a logistic regression (LR) classifier in feature space. The x and y axis in the plots represent the sepal length and the sepal width, respectively. Different colours correspond to different predicted classes. The markers represents the data points in the training set.')
print('Maximum value model linearity: {:.5f}'.format(np.round(L.max(), 5)))
print('Minimum value model linearity: {:.5f}'.format(np.round(L.min(), 5)))
Decision boundaries (left panel) and linearity measure (right panel) for a logistic regression (LG) classifier in feature space. The x and y axis in the plots represent the sepal length and the sepal width, respectively. Different colours correspond to different predicted classes. The markers represent the data points in the training set.

Maximum value model linearity: 0.01841
Minimum value model linearity: 0.0

### 35.6.2 Random forest

```
[12]: # Defining predict function for random forest
clf = rf
predict_fn = lambda x: clf.predict_proba(x)

[13]: # Calculating linearity for all points in the grid
lm = LinearityMeasure(agg='pairwise')
lm.fit(X_train)
L = lm.score(predict_fn, X)
L = L.reshape(xx.shape)
lins_dict['RF'] = L.mean()

[14]: # Visualising decision boundaries and linearity values
f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(16, 8))
idx = (0,0)
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
axarr[0].contourf(xx, yy, Z, alpha=0.4)
axarr[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20, edgecolor='k',
alpha=1)
axarr[0].set_title('Decision boundaries', fontsize=20)
(continues on next page)```
Decision boundaries (left panel) and linearity measure (right panel) for a random forest (RF) classifier in feature space. The x and y axis in the plots represent the sepal length and the sepal width, respectively. Different colours correspond to different predicted classes. The markers represents the data points in the training set.

Maximum value model linearity: 12.07288
Minimum value model linearity: 0.0
35.6.3 Xgboost

```python
[15]: # Defining predict function for xgboost
clf = xgb
predict_fn = lambda x: clf.predict_proba(x)

[16]: # Calculating linearity for all points in the grid
lm = LinearityMeasure(agg='pairwise')
lm.fit(X_train)
L = lm.score(predict_fn, X)
L = L.reshape(xx.shape)
lins_dict['XB'] = L.mean()

[17]: # Visualising decision boundaries and linearity values
f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(16, 8))
idx = (0,0)
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
axarr[0].contourf(xx, yy, Z, alpha=0.4)
axarr[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20, edgecolor='k',
                 alpha=1)
axarr[0].set_title('Decision boundaries', fontsize=20)
axarr[0].set_xlabel('sepal length (cm)', fontsize=20)
axarr[0].set_ylabel('sepal width (cm)', fontsize=20)

LPL = axarr[1].contourf(xx, yy, L, alpha=0.8, cmap='Greys')
axarr[1].set_title('L measure', fontsize=20)
axarr[1].set_xlabel('sepal length (cm)', fontsize=20)
axarr[1].set_ylabel('sepal width (cm)', fontsize=20)

cbar = f.colorbar(LPL)
#cbar.ax.set_ylabel('Linearity')
plt.show()
print('Decision boundaries (left panel) and linearity measure (right panel) for an xgboost (XB) classifier in feature space. The x and y axis in the plots represent the sepal length and the sepal width, respectively. Different colours correspond to different predicted classes. The markers represents the data points in the training set.')
print('Maximum value model linearity: {:.5f}'.format(np.round(L.max(), 5)))
print('Minimum value model linearity: {:.5f}'.format(np.round(L.min(), 5)))
```

Chapter 35. Linearity measure applied to Iris
Decision boundaries (left panel) and linearity measure (right panel) for an XGBoost classifier in feature space. The x and y axis in the plots represent the sepal length and the sepal width, respectively. Different colours correspond to different predicted classes. The markers represent the data points in the training set.

Maximum value model linearity: 1.42648
Minimum value model linearity: 0.0

35.6.4 SVM

[18]: # Defining predict function for svm
clf = svm
predict_fn = lambda x: clf.predict_proba(x)

[19]: # Calculating linearity for all points in the grid
lm = LinearityMeasure(agg='pairwise')
lm.fit(X_train)
L = lm.score(predict_fn, X)
L = L.reshape(xx.shape)
lins_dict['SM'] = L.mean()

[20]: # Visualising decision boundaries and linearity values
f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(16, 8))
idx = (0,0)
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
axarr[0].contourf(xx, yy, Z, alpha=0.4)
axarr[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20, edgecolor='k', alpha=1)
axarr[0].set_title('Decision boundaries', fontsize=20)
Decision boundaries (left panel) and linearity measure (right panel) for a support vector machine (SM) classifier in feature space. The x and y axis in the plots represent the sepal length and the sepal width, respectively. Different colours correspond to different predicted classes. The markers represent the data points in the training set.

Maximum value model linearity: 0.45113
Minimum value model linearity: 0.00083
35.6.5 NN

[21]: # Defining predict function for svm
clf = nn
predict_fn = lambda x: clf.predict_proba(x)

[22]: # Calculating linearity for all points in the grid
lm = LinearityMeasure(agg='pairwise')
lm.fit(X_train)
L = lm.score(predict_fn, X)
L = L.reshape(xx.shape)
lins_dict['NN'] = L.mean()

[23]: # Visualising decision boundaries and linearity values
f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(16, 8))
idx = (0,0)
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
axarr[0].contourf(xx, yy, Z, alpha=0.4)
axarr[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20, edgecolor='k', alpha=1)
axarr[0].set_title('Decision boundaries', fontsize=20)
axarr[0].set_xlabel('sepal length (cm)', fontsize=18)
axarr[0].set_ylabel('sepal width (cm)', fontsize=18)

LPL = axarr[1].contourf(xx, yy, L, alpha=0.8, cmap='Greys')
axarr[1].set_title('Model linearity', fontsize=20)
axarr[1].set_xlabel('sepal length (cm)', fontsize=18)
axarr[1].set_ylabel('sepal width (cm)', fontsize=18)

cbar = f.colorbar(LPL)
cbar.ax.set_ylabel('Linearity')
plt.show()
print('Decision boundaries (left panel) and linearity measure (right panel) for a feed forward neural network classifier (NN) classifier in feature space. The x and y axis in the plots represent the sepal length and the sepal width, respectively. Different colours correspond to different predicted classes. The markers represents the data points in the training set.')
print('Maximum value model linearity: {:.5f}'.format(np.round(L.max(), 5)))
print('Minimum value model linearity: {:.5f}'.format(np.round(L.min(), 5)))

35.6. Decision boundaries and linearity
35.7 Average linearity over the whole feature space

Decision boundaries (left panel) and linearity measure (right panel) for a feedforward neural network classifier (NN) classifier in feature space. The x and y axis in the plots represent the sepal length and the sepal width, respectively. Different colours correspond to different predicted classes. The markers represent the data points in the training set.

Maximum value model linearity: 0.11615
Minimum value model linearity: 3e-05

Comparison of the linearity measure $L$ averaged over the whole feature space for various models trained on the iris dataset: random forest (RF), xgboost (XB), support vector machine (SM), neural network (NN) and logistic regression (LR). Note that the scale of the X axis is logarithmic.

```python
ax = pd.Series(data=lins_dict).sort_values().plot(kind='barh', figsize=(10,5),
          fontsize=20, color='dimgray',
          width=0.8, logx=True)
ax.set_xlabel('L measure (log scale)', fontsize=20)
print('Comparison of the linearity measure L averaged over the whole feature space
for various models trained on the iris dataset: random forest (RF), xgboost (XB),
support vector machine (SM), neural network (NN) and logistic regression (LR). Note
that the scale of the X axis is logarithmic.')
```
35.7. Average linearity over the whole feature space
LINEARITY MEASURE APPLIED TO FASHION MNIST

36.1 General definition

The model linearity module in alibi provides metric to measure how linear an ML model is. Linearity is designed based on how much the linear superposition of the model’s outputs differs from the output of the same linear superposition of the inputs.

Given $N$ input vectors $v_i$, $N$ real coefficients $\alpha_i$ and a predict function $M(v_i)$, the linearity of the predict function is defined as

$$L = \left\| \sum_i \alpha_i M(v_i) - M \left( \sum_i \alpha_i v_i \right) \right\|$$

If $M$ is a regressor

$$L = \left\| \sum_i \alpha_i \log M(v_i) - \log M \left( \sum_i \alpha_i v_i \right) \right\|$$

If $M$ is a classifier

Note that a lower value of $L$ means that the model $M$ is more linear.

36.2 Alibi implementation

- Based on the general definition above, alibi calculates the linearity of a model in the neighborhood of a given instance $v_0$.

36.3 Fashion MNIST data set

- We train a convolutional neural network to classify the images in the fashion MNIST dataset.
- We investigate the correlation between the model’s linearity associated to a certain instance and the class the instance belong to.
- We also calculate the linearity measure for each internal layer of the CNN and show how linearity propagates through the model.

[1]:

```python
import pandas as pd
import numpy as np
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
from time import time
```

(continues on next page)
import tensorflow as tf

from alibi.confidence.model_linearity import linearity_measure, LinearityMeasure
from alibi.confidence.model_linearity import _infer_feature_range

from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D,
   Input, Activation
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import backend as K

### 36.4 Load data fashion mnist

The fashion MNIST data set consists of 60000 images of shape $28 \times 28$ divided in 10 categories. Each category corresponds to a different type of clothing piece, such as “boots”, “t-shirts”, etc.

```python
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
print('x_train shape:', x_train.shape, 'y_train shape:', y_train.shape)
x_train shape: (60000, 28, 28) y_train shape: (60000,)
```

```python
idx = 0
plt.imshow(x_train[idx])
print('Sample instance from the MNIST data set.')
Sample instance from the MNIST data set.
```

```python
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = np.reshape(x_train, x_train.shape + (1,))
x_test = np.reshape(x_test, x_test.shape + (1,))
print('x_train shape:', x_train.shape, 'x_test shape:', x_test.shape)
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
print('y_train shape:', y_train.shape, 'y_test shape:', y_test.shape)
```
36.5 Convolutional neural network

Here we define and train a 2 layer convolutional neural network on the fashion MNIST data set.

36.5.1 Define model

```python
[6]: def model():
    x_in = Input(shape=(28, 28, 1), name='input')
    x = Conv2D(filters=64, kernel_size=2, padding='same', name='conv_1')(x_in)
    x = Activation('relu', name='relu_1')(x)
    x = MaxPooling2D(pool_size=2, name='maxp_1')(x)
    x = Dropout(0.3, name='drop_1')(x)
    x = Conv2D(filters=64, kernel_size=2, padding='same', name='conv_2')(x)
    x = Activation('relu', name='relu_2')(x)
    x = MaxPooling2D(pool_size=2, name='maxp_2')(x)
    x = Dropout(0.3, name='drop_2')(x)
    x = Flatten(name='flat')(x)
    x = Dense(256, name='dense_1')(x)
    x = Activation('relu', name='relu_3')(x)
    x = Dropout(0.5, name='drop_3')(x)
    x_out = Dense(10, name='dense_2')(x)
    x_out = Activation('softmax', name='softmax')(x_out)
    cnn = Model(inputs=x_in, outputs=x_out)
    cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return cnn

[7]:
cnn = model()
cnn.summary()
```

Model: "model"

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>input (InputLayer)</td>
<td>(None, 28, 28, 1)</td>
<td>0</td>
</tr>
<tr>
<td>conv_1 (Conv2D)</td>
<td>(None, 28, 28, 64)</td>
<td>320</td>
</tr>
<tr>
<td>relu_1 (Activation)</td>
<td>(None, 28, 28, 64)</td>
<td>0</td>
</tr>
<tr>
<td>maxp_1 (MaxPooling2D)</td>
<td>(None, 14, 14, 64)</td>
<td>0</td>
</tr>
<tr>
<td>drop_1 (Dropout)</td>
<td>(None, 14, 14, 64)</td>
<td>0</td>
</tr>
<tr>
<td>conv_2 (Conv2D)</td>
<td>(None, 14, 14, 64)</td>
<td>16448</td>
</tr>
</tbody>
</table>

(continues on next page)
36.5.2 Training

```python
[cnn.fit(x_train, y_train, batch_size=64, epochs=5);
Epoch 1/5
60000/60000 [==============================] - 40s 674us/sample - loss: 0.5552 - acc: 0.7955
Epoch 2/5
60000/60000 [==============================] - 43s 717us/sample - loss: 0.3865 - acc: 0.8596
Epoch 3/5
60000/60000 [==============================] - 51s 852us/sample - loss: 0.3421 - acc: 0.8765
Epoch 4/5
60000/60000 [==============================] - 47s 782us/sample - loss: 0.3123 - acc: 0.8851
Epoch 5/5
60000/60000 [==============================] - 48s 802us/sample - loss: 0.2938 - acc: 0.8936
```

36.6 Linearity of each Layer

Here we calculate the linearity of the model considering each layer as the output in turn. The values are averaged over 100 random instances sampled from the training set.
36.6.1 Extract layers

```python
inp = cnn.input
outs = {l.name: l.output for l in cnn.layers}
predict_fns = {name: K.function([inp], [out]) for name, out in outs.items()}
```

36.6.2 Calculate linearity

```python
# Infering feature ranges.
features_range = _infer_feature_range(x_test)

# Selecting random instances from training set.
rnd = np.random.randint(len(x_test) - 101, size=100)

lins_layers = {}
for name, l in predict_fns.items():
    if name != 'input':
        def predict_fn(x):
            layer = l([x])
            return layer[0]
        if name == 'softmax':
            lins_layers[name] = linearity_measure(predict_fn, x_test[rnd], feature_range=features_range,
                                                  agg='global', model_type='classifier', nb_samples=20)
        else:
            lins_layers[name] = linearity_measure(predict_fn, x_test[rnd], feature_range=features_range,
                                                  agg='global', model_type='regressor', nb_samples=20)
    lins_layers_mean = {k: v.mean() for k, v in lins_layers.items()}
S = pd.Series(data=lins_layers_mean)

colors = ['gray' for l in S[:-1]]
colors.append('r')
ax = S.plot(kind='bar', linewidth=3, figsize=(15,10), color=colors, width=0.7,
            fontsize=18)
ax.set_ylabel('L measure', fontsize=20)
ax.set_xlabel('Layer', fontsize=20)
print('Linearity measure calculated taking as output each layer of a convolutional network.')
```

Linearity measure calculated taking as output each layer of a convolutional neural network.
Linearity measure in the locality of a given instance calculated taking as output each layer of a convolutional neural network trained on the fashion MNIST data set. * The linearity measure of the first convolutional layer conv_1 is 0, as expected since convolutions are linear operations. * The relu activation introduces non-linearity, which is increased by maxpooling. Dropout layers and flatten layers do no change the output at inference time so the linearity doesn’t change. * The second convolutional layer conv_2 and the dense layers change the linearity even though they are linear operations. * The softmax layer in red is obtained by inverting the softmax function. * For more details see arxiv reference.

36.7 Linearity and categories

Here we calculate the linearity averaged over all instances belonging to the same class, for each class.

```python
[13]: class_groups = []
    for i in range(10):
        y = y_test.argmax(axis=1)
        idxs_i = np.where(y == i)[0]
        class_groups.append(x_test[idxs_i])
```

```python
[14]: def predict_fn(x):
    return cnn.predict(x)
lins_classes = []
t_0 = time()
```
```python
for j in range(len(class_groups)):
    print(f'Calculating linearity for instances belonging to class {j}')
    class_group = class_groups[j]
    class_group = np.random.permutation(class_group)[:2000]
    t_i = time()
    lin = linearity_measure(predict_fn, class_group, feature_range=features_range,
                            agg='global', model_type='classifier', nb_samples=20)
    t_i_1 = time() - t_i
    print(f'Run time for class {j}: {t_i_1}')
    lins_classes.append(lin)
    t_fin = time() - t_0
    print(f'Total run time: {t_fin}')

Calculating linearity for instances belonging to class 0
Run time for class 0: 2.941605806350708
Calculating linearity for instances belonging to class 1
Run time for class 1: 3.3313376903533936
Calculating linearity for instances belonging to class 2
Run time for class 2: 3.178601026535034
Calculating linearity for instances belonging to class 3
Run time for class 3: 3.324582815170288
Calculating linearity for instances belonging to class 4
Run time for class 4: 3.08533830947876
Calculating linearity for instances belonging to class 5
Run time for class 5: 3.159513473510742
Calculating linearity for instances belonging to class 6
Run time for class 6: 3.4014275074005127
Calculating linearity for instances belonging to class 7
Run time for class 7: 3.3238165378570557
Calculating linearity for instances belonging to class 8
Run time for class 8: 3.985218143463135
Calculating linearity for instances belonging to class 9
Run time for class 9: 3.4760279655456543
Total run time: 32.22387504577637

[15]: df = pd.DataFrame(data=lins_classes).T
[16]: ax = df.mean().plot(kind='bar', linewidth=3, figsize=(15,10), color='gray', width=0.7,
                           fontsize=10)
    ax.set_ylabel('L measure', fontsize=20)
    ax.set_xlabel('Class', fontsize=20)
    print("Linearity measure distribution means for each class in the fashion MNIST data set.")
```

Linearity measure distribution means for each class in the fashion MNIST data set.
Linearity measure distributions for each class in the fashion MNIST data set.
36.7. Linearity and categories
It is important to know when a machine learning classifier’s predictions can be trusted. Relying on the classifier’s (uncalibrated) prediction probabilities is not optimal and can be improved upon. Trust scores measure the agreement between the classifier and a modified nearest neighbor classifier on the test set. The trust score is the ratio between the distance of the test instance to the nearest class different from the predicted class and the distance to the predicted class. Higher scores correspond to more trustworthy predictions. A score of 1 would mean that the distance to the predicted class is the same as to another class.

The original paper on which the algorithm is based is called To Trust Or Not To Trust A Classifier. Our implementation borrows heavily from https://github.com/google/TrustScore, as does the example notebook.

37.1 Load and prepare Iris dataset

```python
import matplotlib
%matplotlib inline
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedShuffleSplit
from alibi.confidence import TrustScore
```

```python
dataset = load_iris()
```

Scale data

```python
dataset.data = (dataset.data - dataset.data.mean(axis=0)) / dataset.data.std(axis=0)
```

Define training and test set

```python
idx = 140
X_train, y_train = dataset.data[:idx, :], dataset.target[:idx]
X_test, y_test = dataset.data[idx+1:, :], dataset.target[idx+1:]
```
### 37.2 Fit model and make predictions

```python
[5]: np.random.seed(0)
clf = LogisticRegression(solver='liblinear', multi_class='auto')
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(f'Predicted class: {y_pred}')
Predicted class: [2 2 2 2 2 2 2 2 2]
```

### 37.3 Basic Trust Score Usage

#### 37.3.1 Initialise Trust Scores and fit on training data

The trust score algorithm builds k-d trees for each class. The distance of the test instance to the kth nearest neighbor of each tree (or the average distance to the kth neighbor) can then be used to calculate the trust score. We can optionally filter out outliers in the training data before building the trees. The example below uses the `distance_knn` (filter_type) method to filter out the 5% (alpha) instances of each class with the highest distance to its 10th nearest neighbor (k_filter) in that class.

```python
[6]: ts = TrustScore(k_filter=10,  # nb of neighbors used for kNN distance or probability
       alpha=.05,  # target fraction of instances to filter out
       filter_type='distance_knn',  # filter method: None, 'distance_knn' or 'probability_knn'
       leaf_size=40,  # affects speed and memory to build KDTrees, memory scales with n_samples / leaf_size
       metric='euclidean',  # distance metric used for the KDTrees
       dist_filter_type='point')  # 'point' uses distance to k-nearest point
       # 'mean' uses average distance from the 1st to the kth nearest point

[7]: ts.fit(X_train, y_train, classes=3)  # classes = nb of prediction classes
```

#### 37.3.2 Calculate Trust Scores on test data

Since the trust score is the ratio between the distance of the test instance to the nearest class different from the predicted class and the distance to the predicted class, higher scores correspond to more trustworthy predictions. A score of 1 would mean that the distance to the predicted class is the same as to another class. The `score` method returns arrays with both the trust scores and the class labels of the closest not predicted class.

```python
[8]: score, closest_class = ts.score(X_test,  # kth nearest neighbor used
       y_pred, k=2,  # kth nearest neighbor used
       dist_type='point')  # 'point' or 'mean' distance

print(f'Trust scores: (score)')
print(f'\nClosest not predicted class: (closest_class)')
Trust scores: [2.574271277538439 2.163033495787011 3.162940536774222 2.725849454415792 2.541748027539072 1.402878283257114 1.941073062524019 2.060172542435929 2.178112149457351]
```

(continues on next page)
Closest not predicted class: [1 1 1 1 1 1 1 1 1]

37.4 Comparison of Trust Scores with model prediction probabilities

Let’s compare the prediction probabilities from the classifier with the trust scores for each prediction. The first use case checks whether trust scores are better than the model’s prediction probabilities at identifying correctly classified examples, while the second use case does the same for incorrectly classified instances.

First we need to set up a couple of helper functions.

- Define a function that handles model training and predictions for a simple logistic regression:

```python
[9]: def run_lr(X_train, y_train, X_test):
    clf = LogisticRegression(solver='liblinear', multi_class='auto')
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    y_pred_proba = clf.predict_proba(X_test)
    probas = y_pred_proba[range(len(y_pred)), y_pred]  # probabilities of predicted class
    return y_pred, probas
```

- Define the function that generates the precision plots:

```python
[10]: def plot_precision_curve(plot_title, percentiles, labels, final_tp, final_stderr, final_misclassification, colors = ['blue', 'darkorange', 'brown', 'red', 'purple']):
    plt.title(plot_title, fontsize=18)
    colors = colors + list(cm.rainbow(np.linspace(0, 1, len(final_tp))))
    plt.xlabel("Percentile", fontsize=14)
    plt.ylabel("Precision", fontsize=14)
    for i, label in enumerate(labels):
        ls = "--" if ("Model" in label) else "-
        plt.plot(percentiles, final_tp[i], ls, c=colors[i], label=label)
        plt.fill_between(percentiles, final_tp[i] - final_stderr[i],
                         final_tp[i] + final_stderr[i],
                         color=colors[i],
                         alpha=.1)
    if 0. in percentiles:
        plt.legend(loc="lower right", fontsize=14)
    else:
        plt.legend(loc="upper left", fontsize=14)
    model_acc = 100 * (1 - final_misclassification)
    plt.axvline(x=model_acc, linestyle="dotted", color="black")
    plt.show()
```
The function below trains the model on a number of folds, makes predictions, calculates the trust scores, and generates the precision curves to compare the trust scores with the model prediction probabilities:

```python
[11]: def run_precision_plt(X, y, nfolds, percentiles, run_model, test_size=.5,
   plt_title='', plt_names=[], predict_correct=True, classes=3):

    def stderr(L):
        return np.std(L) / np.sqrt(len(L))

    all_tp = [[[] for p in percentiles] for _ in plt_names]
    misclassifications = []
    mult = 1 if predict_correct else -1

    folds = StratifiedShuffleSplit(n_splits=nfolds, test_size=test_size, random_=0)
    for train_idx, test_idx in folds.split(X, y):
        # create train and test folds, train model and make predictions
        X_train, y_train = X[train_idx, :], y[train_idx]
        X_test, y_test = X[test_idx, :], y[test_idx]
        y_pred, probas = run_model(X_train, y_train, X_test)
        # target points are the correctly classified points
        target_points = np.where(y_pred == y_test)[0] if predict_correct else np.where(y_pred != y_test)[0]
        final_curves = [probas]
        # calculate trust scores
        ts = TrustScore()
        ts.fit(X_train, y_train, classes=classes)
        scores, _ = ts.score(X_test, y_pred)
        final_curves.append(scores)  # contains prediction probabilities and trust scores
        # check where prediction probabilities and trust scores are above a certain percentage level
        for p, perc in enumerate(percentiles):
            high_proba = [np.where(mult * curve >= np.percentile(mult * curve, perc))[0] for curve in final_curves]
            if 0 in map(len, high_proba):
                continue
            # calculate fraction of values above percentage level that are correctly (or incorrectly) classified
            tp = [len(np.intersect1d(hp, target_points)) / (1. * len(hp)) for hp in high_proba]
            for i in range(len(plt_names)):
                all_tp[i][p].append(tp[i])  # for each percentile, store fraction of values above cutoff value
            misclassifications.append(len(target_points) / (1. * len(X_test)))

        # average over folds for each percentile
        final_tp = [[[] for _ in plt_names] for p, perc in enumerate(percentiles):
            for i in range(len(plt_names)):
                final_tp[i].append(np.mean(all_tp[i][p]))
            final_stdderr[i].append(stderr(all_tp[i][p]))

        for i in range(len(all_tp)):
            final_tp[i] = np.array(final_tp[i])
            final_stdderr[i] = np.array(final_stdderr[i])

    # (continues on next page)
```
37.4.1 Detect correctly classified examples

The x-axis on the plot below shows the percentiles for the model prediction probabilities of the predicted class for each instance and for the trust scores. The y-axis represents the precision for each percentile. For each percentile level, we take the test examples whose trust score is above that percentile level and plot the percentage of those points that were correctly classified by the classifier. We do the same with the classifier’s own model confidence (i.e. softmax probabilities). For example, at percentile level 80, we take the top 20% scoring test examples based on the trust score and plot the percentage of those points that were correctly classified. We also plot the top 20% scoring test examples based on model probabilities and plot the percentage of those that were correctly classified. The vertical dotted line is the error of the logistic regression classifier. The plots are an average over 10 folds of the dataset with 50% of the data kept for the test set.

The Trust Score and Model Confidence curves then show that the model precision is typically higher when using the trust scores to rank the predictions compared to the model prediction probabilities.

```python
X = dataset.data
y = dataset.target
percentiles = [0 + 0.5 * i for i in range(200)]
nfolds = 10
plt_names = ['Model Confidence', 'Trust Score']
plt_title = 'Iris -- Logistic Regression -- Predict Correct'

run_precision_plt(X, y, nfolds, percentiles, run_lr, plt_title=plt_title,
                  plt_names=plt_names, predict_correct=True)
```

Iris -- Logistic Regression -- Predict Correct

![Plot of Precision vs Percentile](image)
37.4.2 Detect incorrectly classified examples

By taking the negative of the prediction probabilities and trust scores, we can also see on the plot below how the trust scores compare to the model predictions for incorrectly classified instances. The vertical dotted line is the accuracy of the logistic regression classifier. The plot shows the precision of identifying incorrectly classified instances. Higher is obviously better.

```
percentiles = [50 + 0.5 * i for i in range(100)]
plt_title = 'Iris -- Logistic Regression -- Predict Incorrect'
run_precision_plt(X, y, nfolds, percentiles, run_lr, plt_title=plt_title,
                  plt_names=plt_names, predict_correct=False)
```

TRUST SCORES APPLIED TO MNIST

It is important to know when a machine learning classifier’s predictions can be trusted. Relying on the classifier’s (uncalibrated) prediction probabilities is not optimal and can be improved upon. Trust scores measure the agreement between the classifier and a modified nearest neighbor classifier on the test set. The trust score is the ratio between the distance of the test instance to the nearest class different from the predicted class and the distance to the predicted class. Higher scores correspond to more trustworthy predictions. A score of 1 would mean that the distance to the predicted class is the same as to another class.

The original paper on which the algorithm is based is called To Trust Or Not To Trust A Classifier. Our implementation borrows heavily from https://github.com/google/TrustScore, as does the example notebook.

Trust scores work best for low to medium dimensional feature spaces. This notebook illustrates how you can apply trust scores to high dimensional data like images by adding an additional pre-processing step in the form of an auto-encoder to reduce the dimensionality. Other dimension reduction techniques like PCA can be used as well.

```python
[1]: import tensorflow as tf
    from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D,
    Input, UpSampling2D
    from tensorflow.keras.models import Model, load_model
    from tensorflow.keras.utils import to_categorical
    import matplotlib
    %matplotlib inline
    import matplotlib.cm as cm
    import matplotlib.pyplot as plt
    import numpy as np
    from sklearn.model_selection import StratifiedShuffleSplit
    from alibi.confidence import TrustScore

[2]: (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
    print('x_train shape:', x_train.shape, 'y_train shape:', y_train.shape)
    plt.gray()
    plt.imshow(x_test[0]);
    x_train shape: (60000, 28, 28) y_train shape: (60000,)
```
Prepare data: scale, reshape and categorize

```python
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = np.reshape(x_train, x_train.shape + (1,))
x_test = np.reshape(x_test, x_test.shape + (1,))
print('x_train shape:', x_train.shape, 'x_test shape:', x_test.shape)
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
print('y_train shape:', y_train.shape, 'y_test shape:', y_test.shape)
x_train shape: (60000, 28, 28, 1) x_test shape: (10000, 28, 28, 1)
y_train shape: (60000, 10) y_test shape: (10000, 10)
```

```python
xmin, xmax = -.5, .5
x_train = ((x_train - x_train.min()) / (x_train.max() - x_train.min())) * (xmax - xmin) + xmin
x_test = ((x_test - x_test.min()) / (x_test.max() - x_test.min())) * (xmax - xmin) + xmin
```

38.1 Define and train model

For this example we are not interested in optimizing model performance so a simple softmax classifier will do:

```python
def sc_model():
    x_in = Input(shape=(28, 28, 1))
x = Flatten()(x_in)
x_out = Dense(10, activation='softmax')(x)
sc = Model(inputs=x_in, outputs=x_out)
sc.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
return sc
```

```python
sc = sc_model()
s.summary()
s.fit(x_train, y_train, batch_size=128, epochs=5);
```
38.2 Define and train auto-encoder

```python
[8]: def ae_model():
    # encoder
    x_in = Input(shape=(28, 28, 1))
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x_in)
    x = MaxPooling2D((2, 2), padding='same')(x)
    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)
    x = Conv2D(4, (3, 3), activation=None, padding='same')(x)
    encoded = MaxPooling2D((2, 2), padding='same')(x)
    encoder = Model(x_in, encoded)

    # decoder
    dec_in = Input(shape=(4, 4, 4))
    x = Conv2D(4, (3, 3), activation='relu', padding='same')(dec_in)
    x = UpSampling2D((2, 2))(x)
    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
```

(continues on next page)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation=None, padding='same')(x)
decoder = Model(dec_in, decoded)

# autoencoder = encoder + decoder
x_out = decoder(encoder(x_in))
autoencoder = Model(x_in, x_out)
autoencoder.compile(optimizer='adam', loss='mse')
return autoencoder, encoder, decoder

[9]: ae, enc, dec = ae_model()
ae.summary()
ae.fit(x_train, x_train, batch_size=128, epochs=8, validation_data=(x_test, x_test))
ae.save('mnist_ae.h5')
enc.save('mnist_enc.h5')

Model: "model_3"

Layer (type)                  Output Shape  Param #
=================================================================
input_2 (InputLayer)         [(None, 28, 28, 1)] 0
model_1 (Model)              (None, 4, 4, 4)  1612
model_2 (Model)              (None, 28, 28, 1) 1757
=================================================================
Total params: 3,369
Trainable params: 3,369
Non-trainable params: 0

Train on 60000 samples, validate on 10000 samples
Epoch 1/8
60000/60000 [==============================] - 29s 477us/sample - loss: 0.0606 - val_
→loss: 0.0399
Epoch 2/8
60000/60000 [==============================] - 34s 572us/sample - loss: 0.0341 - val_
→loss: 0.0301
Epoch 3/8
60000/60000 [==============================] - 43s 715us/sample - loss: 0.0288 - val_
→loss: 0.0272
Epoch 4/8
60000/60000 [==============================] - 48s 806us/sample - loss: 0.0265 - val_
→loss: 0.0253
Epoch 5/8
60000/60000 [==============================] - 41s 680us/sample - loss: 0.0249 - val_
→loss: 0.0239
Epoch 6/8
60000/60000 [==============================] - 39s 649us/sample - loss: 0.0237 - val_
→loss: 0.0230
Epoch 7/8
60000/60000 [==============================] - 33s 545us/sample - loss: 0.0229 - val_
→loss: 0.0222
Epoch 8/8
60000/60000 [==============================] - 29s 484us/sample - loss: 0.0224 - val_
→loss: 0.0217

(continues on next page)
38.3 Calculate Trust Scores

Initialize trust scores:

```python
[11]: ts = TrustScore()
```

The key is to fit and calculate the trust scores on the encoded instances. The encoded data still needs to be reshaped from (60000, 4, 4, 4) to (60000, 64) to comply with the k-d tree format. This is handled internally:

```python
[12]: x_train_enc = enc.predict(x_train)
ts.fit(x_train_enc, y_train, classes=10)  # 10 classes present in MNIST
```

Reshaping data from (60000, 4, 4, 4) to (60000, 64) so k-d trees can be built.

We can now calculate the trust scores and closest not predicted classes of the predictions on the test set, using the distance to the 5th nearest neighbor in each class:

```python
[13]: n_samples = 1000  # calculate the trust scores for the first 1000 predictions on the test set
    x_test_enc = enc.predict(x_test[:n_samples])
    y_pred = sc.predict(x_test[:n_samples])
    score, closest_class = ts.score(x_test_enc[:n_samples], y_pred, k=5)
```

Reshaping data from (1000, 4, 4, 4) to (1000, 64) so k-d trees can be queried.

Let's inspect which predictions have low and high trust scores:

```python
[14]: n = 5
    # lowest and highest trust scores
    idx_min, idx_max = np.argsort(score)[:n], np.argsort(score)[-n:]
    score_min, score_max = score[idx_min], score[idx_max]
    closest_min, closest_max = closest_class[idx_min], closest_class[idx_max]
    pred_min, pred_max = y_pred[idx_min], y_pred[idx_max]
    imgs_min, imgs_max = x_test[idx_min], x_test[idx_max]
    label_min, label_max = np.argmax(y_test[idx_min], axis=1), np.argmax(y_test[idx_max], axis=1)
    # model confidence percentiles
    max_proba = y_pred.max(axis=1)
    # low score high confidence examples
    idx_low = np.where((max_proba>0.80) & (max_proba<0.9) & (score<1))[:n]
    score_low = score[idx_low]
    closest_low = closest_class[idx_low]
    pred_low = y_pred[idx_low]
    imgs_low = x_test[idx_low]
    label_low = np.argmax(y_test[idx_low], axis=1)
```

38.3. Calculate Trust Scores
38.3.1 Low Trust Scores

The image below makes clear that the low trust scores correspond to misclassified images. Because the trust scores are significantly below 1, they correctly identified that the images belong to another class than the predicted class, and identified that class.

![Images showing low trust scores](image)

38.3.2 High Trust Scores

The high trust scores on the other hand all are very clear 1’s:

![Images showing high trust scores](image)
38.3.3 High model confidence, low trust score

Where trust scores really matter is when the predicted model confidence is relatively high (e.g. $p \in [0.8, 0.9]$) but the corresponding trust score is low, this can indicate samples for which the model is overconfident. The trust score provides a diagnostic for finding these examples:

```python
[18]: plt.figure(figsize=(20, 4))
for i in range(n):
    ax = plt.subplot(1, n, i+1)
    plt.imshow(imgs_low[i].reshape(28, 28))
    plt.title('Model prediction: {} (p={:.2f})
              Label: {}
              Trust score: {:.3f}
              Closest other class: {}
'.format(pred_low[i].argmax(), pred_low[i].max(), label_low[i], score_low[i], closest_low[i]), fontsize=14)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

We can see several examples of an over-confident model predicting the wrong class, the low trust score, however, reveals that this is happening and the predictions should not be trusted despite the high model confidence.

In the following section we will see that on average trust scores outperform the model confidence for identifying correctly classified samples.

38.4 Comparison of Trust Scores with model prediction probabilities

Let’s compare the prediction probabilities from the classifier with the trust scores for each prediction by checking whether trust scores are better than the model’s prediction probabilities at identifying correctly classified examples.

First we need to set up a couple of helper functions.

- Define a function that handles model training and predictions:

```python
[19]:
def run_sc(X_train, y_train, X_test):
    clf = sc_model()
    clf.fit(X_train, y_train, batch_size=128, epochs=5, verbose=0)
    y_pred_proba = clf.predict(X_test)
    y_pred = np.argmax(y_pred_proba, axis=1)
    probas = y_pred_proba[range(len(y_pred)), y_pred]  # probabilities of predicted class
    return y_pred, probas
```

- Define the function that generates the precision plots:
```
[20]: `def` plot_precision_curve(plot_title,
   percentiles,
   labels,
   final_tp,
   final_stderr,
   final_misclassification,
   colors = ['blue', 'darkorange', 'brown', 'red', 'purple']):

   plt.title(plot_title, fontsize=18)
   colors = colors + list(cm.rainbow(np.linspace(0, 1, len(final_tp))))
   plt.xlabel("Percentile", fontsize=14)
   plt.ylabel("Precision", fontsize=14)

   for i, label in enumerate(labels):
       ls = "--" if "Model" in label else "-"
       plt.plot(percentiles, final_tp[i], ls, c=colors[i], label=label)
       plt.fill_between(percentiles,
                        final_tp[i] - final_stderr[i],
                        final_tp[i] + final_stderr[i],
                        color=colors[i],
                        alpha=.1)

   if 0. in percentiles:
       plt.legend(loc="lower right", fontsize=14)
   else:
       plt.legend(loc="upper left", fontsize=14)
   model_acc = 100 * (1 - final_misclassification)
   plt.axvline(x=model_acc, linestyle="dotted", color="black")
   plt.show()

• The function below trains the model on a number of folds, makes predictions, calculates the trust scores, and
  generates the precision curves to compare the trust scores with the model prediction probabilities:

[21]: `def` run_precision_plt(X, y, nfolds, percentiles, run_model, test_size=.2,
    plot_title="", plt_names=[], predict_correct=True, classes=10):

   def stderr(L):
       return np.std(L) / np.sqrt(len(L))

   all_tp = [[[] for p in percentiles] for _ in plt_names]
   misclassifications = []
   mult = 1 if predict_correct else -1

   folds = StratifiedShuffleSplit(n_splits=nfolds, test_size=test_size, random_state=0)
   for train_idx, test_idx in folds.split(X, y):
       # create train and test folds, train model and make predictions
       X_train, y_train = X[train_idx, :], y[train_idx, :]
       X_test, y_test = X[test_idx, :], y[test_idx, :]
       y_pred, probas = run_model(X_train, y_train, X_test)
       # target points are the correctly classified points
       y_test_class = np.argmax(y_test, axis=1)
       target_points = (np.where(y_pred == y_test_class)[0] if predict_correct else
                        np.where(y_pred != y_test_class)[0])
       final_curves = [probas]
       # calculate trust scores
       ts = TrustScore()

   (continues on next page)"


```python
ts.fit(enc.predict(X_train), y_train, classes=classes)
scores, _ = ts.score(enc.predict(X_test), y_pred, k=5)
final_curves.append(scores)  # contains prediction probabilities and trust scores

# check where prediction probabilities and trust scores are above a certain percentage level
for p, perc in enumerate(percentiles):
    high_proba = [np.where(mult * curve >= np.percentile(mult * curve, perc))[0] for curve in final_curves]
    if 0 in map(len, high_proba):
        continue
    # calculate fraction of values above percentage level that are correctly (or incorrectly) classified
    tp = [len(np.intersect1d(hp, target_points)) / (1. * len(hp)) for hp in high_proba]
    for i in range(len(plt_names)):
        all_tp[i][p].append(tp[i])  # for each percentile, store fraction of values above cutoff value

        # average over folds for each percentile
final_tp = [[] for _ in plt_names]
final_stderr = [[] for _ in plt_names]
for p, perc in enumerate(percentiles):
    for i in range(len(plt_names)):
        final_tp[i].append(np.mean(all_tp[i][p]))
        final_stderr[i].append(stderr(all_tp[i][p]))

    for i in range(len(all_tp)):
        final_tp[i] = np.array(final_tp[i])
        final_stderr[i] = np.array(final_stderr[i])

final_misclassification = np.mean(misclassifications)

# create plot
plot_precision_curve(plt_title, percentiles, plt_names, final_tp, final_stderr, final_misclassification)
```

### 38.5 Detect correctly classified examples

The x-axis on the plot below shows the percentiles for the model prediction probabilities of the predicted class for each instance and for the trust scores. The y-axis represents the precision for each percentile. For each percentile level, we take the test examples whose trust score is above that percentile level and plot the percentage of those points that were correctly classified by the classifier. We do the same with the classifier’s own model confidence (i.e. softmax probabilities). For example, at percentile level 80, we take the top 20% scoring test examples based on the trust score and plot the percentage of those points that were correctly classified. We also plot the top 20% scoring test examples based on model probabilities and plot the percentage of those that were correctly classified. The vertical dotted line is the error of the classifier. The plots are an average over 2 folds of the dataset with 20% of the data kept for the test set.

The **Trust Score** and **Model Confidence** curves then show that the model precision is typically higher when using the trust scores to rank the predictions compared to the model prediction probabilities.

```python
X = x_train
y = y_train
```

```
[22]:
```

(continues on next page)
percentiles = [0 + 0.5 * i for i in range(200)]
nfolds = 2
plt_names = ['Model Confidence', 'Trust Score']
plt_title = 'MNIST -- Softmax Classifier -- Predict Correct'

[23]: run_precision_plt(X, y, nfolds, percentiles, run_sc, plt_title=plt_title, plt_names=plt_names, predict_correct=True)

Reshaping data from (48000, 4, 4, 4) to (48000, 64) so k-d trees can be built.
Reshaping data from (12000, 4, 4, 4) to (12000, 64) so k-d trees can be queried.
Reshaping data from (48000, 4, 4, 4) to (48000, 64) so k-d trees can be built.
Reshaping data from (12000, 4, 4, 4) to (12000, 64) so k-d trees can be queried.
39.1 Introduction

This example shows how to apply interventional Tree SHAP to compute shap values exactly for an \texttt{xgboost} model fitted to the \texttt{Adult} dataset (binary classification task). Furthermore, the shap values computed by Kernel SHAP, an approximate feature attribution method, are shown to converge to the interventional Tree SHAP contributions given a sufficiently large number of model evaluations.

```python
import json
import pickle
import shap
shap.initjs()
import numpy as np
import matplotlib.pyplot as plt
import xgboost as xgb
from alibi.datasets import fetch_adult
from alibi.explainers import KernelShap, TreeShap
from collections import defaultdict, Counter
from functools import partial
from itertools import product, zip_longest
from scipy.special import expit
invlogit=expit
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.utils import resample
from timeit import default_timer as timer
<IPython.core.display.HTML object>
```
39.2 Data preparation

39.2.1 Load and split

The `fetch_adult` function returns a `Bunch` object containing features, targets, feature names and a mapping of categorical variables to numbers.

```python
[3]: adult = fetch_adult()
    adult.keys()
[3]: dict_keys(['data', 'target', 'feature_names', 'target_names', 'category_map'])
```

```python
[4]: data = adult.data
target = adult.target
target_names = adult.target_names
feature_names = adult.feature_names
category_map = adult.category_map
```

Note that for your own datasets you can use the utility function `gen_category_map` imported from `alibi.utils.data` to create the category map.

```python
[5]: np.random.seed(0)
data_perm = np.random.permutation(np.c_[data, target])
data = data_perm[:, :-1]
target = data_perm[:, -1]
```

```python
[6]: idx = 30000
    X_train, y_train = data[:idx, :], target[:idx]
    X_test, y_test = data[idx+1:, :], target[idx+1:]
```

`xgboost` wraps arrays using `DMatrix` objects, optimised for both memory efficiency and training speed.

```python
[7]: def wrap(arr):
    return np.ascontiguousarray(arr)

dtrain = xgb.DMatrix(
    wrap(X_train),
    label=wrap(y_train),
    feature_names=feature_names,
)
dtest = xgb.DMatrix(wrap(X_test), label=wrap(y_test), feature_names=feature_names)
```

Finally, a matrix that contains the raw string values for categorical variables (used for display) is created:

```python
[8]: def _decode_data(X, feature_names, category_map):
    ""
    Given an encoded data matrix `X` returns a matrix where the
    categorical levels have been replaced by human readable categories.
    ""
    X_new = np.zeros(X.shape, dtype=object)
    for idx, name in enumerate(feature_names):
        categories = category_map.get(idx, None)
        if categories:
            for j, category in enumerate(categories):
```

(continues on next page)
encoded_vals = X[:, idx] == j
    X_new[encoded_vals, idx] = category
    else:
        X_new[:, idx] = X[:, idx]

return X_new

decode_data = partial(_decode_data,
    feature_names=feature_names,
    category_map=category_map)

[9]: X_display = decode_data(X_test)

[10]: X_display

array([[52, 'Private', 'Associates', ..., 0, 60, 'United-States'],
    [21, 'Private', 'High School grad', ..., 0, 20, 'United-States'],
    [43, 'Private', 'Dropout', ..., 0, 50, 'United-States'],
    ...,
    [23, 'Private', 'High School grad', ..., 0, 40, 'United-States'],
    [45, 'Local-gov', 'Doctorate', ..., 0, 45, 'United-States'],
    [25, 'Private', 'High School grad', ..., 0, 48, 'United-States']],
   dtype=object)

39.3 Model definition

The model fitted in the xgboost fitting example will be explained. The confusion matrix of this model is shown below.

[11]: def plot_conf_matrix(y_test, y_pred, class_names):
    
    Plots confusion matrix. Taken from:
    http://queirozf.com/entries/visualizing-machine-learning-models-examples-with-
    scikit-learn-and-matplotlib
    
    matrix = confusion_matrix(y_test, y_pred)

    # place labels at the top
    plt.gca().xaxis.tick_top()
    plt.gca().xaxis.set_label_position('top')

    # plot the matrix per se
    plt.imshow(matrix, interpolation='nearest', cmap=plt.cm.Blues)

    # plot colorbar to the right
    plt.colorbar()

    fmt = 'd'

    # write the number of predictions in each bucket
thresh = matrix.max() / 2.
for i, j in product(range(matrix.shape[0]), range(matrix.shape[1])):
    # if background is dark, use a white number, and vice-versa
    plt.text(j, i, format(matrix[i, j], fmt),
             horizontalalignment="center",
             color="white" if matrix[i, j] > thresh else "black")

tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names, rotation=45)
plt.yticks(tick_marks, class_names)
plt.tight_layout()
plt.ylabel('True label', size=14)
plt.xlabel('Predicted label', size=14)
plt.show()

def predict(xgb_model, dataset, proba=False, threshold=0.5):
    ""
    Predicts labels given a xgboost model that outputs raw logits.
    ""

    y_pred = model.predict(dataset)  # raw logits are predicted
    y_pred_proba = invlogit(y_pred)
    if proba:
        return y_pred_proba
    y_pred_class = np.zeros_like(y_pred)
    y_pred_class[y_pred_proba >= threshold] = 1  # assign a label
    return y_pred_class

[12]: model = xgb.Booster()
    model.load_model('adult_xgb.mdl')

[13]: y_pred_train = predict(model, dtrain)
    y_pred_test = predict(model, dtest)

[14]: plot_conf_matrix(y_test, y_pred_test, target_names)
39.4 Explaining xgboost with interventional Tree SHAP: global knowledge from local explanations

Recall that the goal of shap values computation for an instance $x$ is to attribute the difference $f(x) - \mathbb{E}_D[f(x)]$ to $M$ input features. Here $D$ represents the background data. Unlike the path-dependent perturbation algorithm which exploits the tree structure and cover information (derived from the training data) to obviate the need for a background dataset, the interventional perturbation algorithm follows a similar idea to Kernel SHAP and uses a background dataset to compute the expected value as the average of the leaves where the background samples fall plus the baseline model offset (1). As explained in the algorithm overview, this allows explaining nonlinear transformations of the model output, so this method can be used to explain loss function fluctuations.

As discussed in (1) and detailed in the overview, this perturbation method enforces the conditional independence $x_S \perp x_{\bar{S}}$ where $\bar{S}$ is a subset of missing features. This section shows that this method is consistent with the path-dependent perturbation method, in the sense that it leads to very similar analysis conclusions assuming an appropriate background dataset is used.

Because the background dataset contains 30,000 examples, the next part of the example is long running. In practice, sufficient accuracy can be achieved using a couple of hundred samples (the library authors recommend anywhere between 200 and 1000 examples), provided that the samples chosen represent the underlying distribution accurately (i.e., they cover the entire support of the distribution). Instead you can load the results by calling the load_shap_values function.

```python
[15]:
print(f'Train accuracy: {round(100*accuracy_score(y_train, y_pred_train), 4)} %.
print(f'Test accuracy: {round(100*accuracy_score(y_test, y_pred_test), 4)}%.

Train accuracy: 87.75 %.
Test accuracy: 86.6797%.
```
### 314 Chapter 39. Explaining Tree Models with Interventional Feature Perturbation Tree SHAP

Unfortunately, one notices that the local accuracy property does not hold for all examples. A number of issues are currently opened about this issue in the [shap library github repository](https://github.com/slundberg/shap) (see here).

NB: The 0.5 offset is due to the fact that the shap library adds the model initial output value (set to 0.5) to the expected value.

```python
[17]: def load_shap_values():
    with open('data/shap_interv.pkl', 'rb') as f:
        shap_interventional = pickle.load(f)
    return shap_interventional

[18]: interventional_shap_values = load_shap_values()

[19]: # Shows how the above values were computed
# interventional_shap_explanation = tree_explainer_interventional.explain(X_test)
# interventional_shap_values = interventional_shap_explanation.shap_values[0]

100%|===================| 2559/2560 [107:15<00:02]

The error observed seems to be the same for all but 3 examples, and for the majority of the examples could be due to the way the expected value is calculated, as explained in the *White-box vs black-box model explanations: a comparison with Kernel SHAP* section.

```python
[19]: errs = np.abs(model.predict(dtest) - tree_explainer_interventional.expected_value - interventional_shap_values.sum(1))

[20]: print(Counter(np.round(errs, 2)))

Counter({0.5: 2557, 0.84: 1, 0.58: 1, 0.36: 1})

[21]: shap.summary_plot(interventional_shap_values, X_test, feature_names)
Figure 1: Summary plot of the interventional perturbation Tree SHAP explanations for the test set

```
[22]: shap.summary_plot(interventional_shap_values, X_test, feature_names, plot_type='bar')
```
One might be tempted to proceed to compare the feature rankings displayed above with the ranking provided by the path-dependent Tree SHAP example. However, these algorithms have different ways of estimating the effect of missing features and:

1. The length of the bar represents the average magnitude of the points in the summary plot above; each point is the average of the shap values computed for a given instance $x$ with respect to $R$ different background samples. Hence, one can consider that for each instance to be explained the shap value of the $j$th feature is a random variable, denoted by $\Phi_{i,j}$. One way to define the importance of the $j$th feature, $I_j$, is

$$I_j = \frac{1}{N} \sum_{i=1}^{N} |E[\Phi_{i,j}]|,$$

where the expectation is taken over the background distribution and $N$ is the number of instances explained. This corresponds to the notion of feature importance according to which a feature is important for explaining the model behaviour over a given dataset if:

- either the instances to explained are consistently affected by the feature, or the feature has a particularly large impact for certain subgroups and a small or moderate impact for the remainder. Traditional global explanation feature importances hide this information whereas the summary plot reveals why a particular feature was deemed important
- locally, one also requires that cancellation effects are not significant. In other words, for a particular instance, a feature would be considered as not important if, across different backgrounds, cancellation effects result in a small average for the effect.

It should be noted that the error $I_j$ is inversely proportional to the square root of the size of the background dataset for a given dataset to be explained, so it is important to select a sufficient number of background samples in order to reduce the error of this estimate.
2. The two methods explain the dataset with respect to different expected values, so the contributions will be different. This also arises because of the different set of conditional assumptions are made when estimating the individual contributions, as explained in the algorithm overview.

Instead of analysing feature importance rankings, it is perhaps more instructive to look at the dependence plots and see if the conclusions from the previous model interpretation hold. Although the decision plots in Figure 3 show the same patterns as their counterparts in the path-dependent example, different variables are found to have the strongest interaction with the variables of interest so the colouring of the plot is different. This is expected since the different conditional independence assumptions give rise to different magnitudes for the shap values, and therefore the estimations for the Pearson coefficients will be affected.

```python
[31]: def _dependence_plot(features, shap_values, dataset, feature_names, category_map,
˓display_features=None, **kwargs):
    
    Plots dependence plots of specified features in a grid.

    features: List[str], List[Tuple[str, str]]
        Names of features to be plotted. If List[str], then shap values are plotted as a function of feature value, coloured by the value of the feature determined to have the strongest interaction (empirically). If List[Tuple[str, str]], shap interaction values are plotted.

display_features: np.ndarray, N x F
        Same as dataset, but contains human readable values for categorical levels as opposed to numerical values

    
    def _set_fonts(fig, ax, fonts=None, set_cbar=False):
        
        Sets fonts for axis labels and colobar.
        
        ax.xaxis.label.set_size(xlabelfontsize)
        ax.yaxis.label.set_size(ylabelfontsize)
        ax.tick_params(axis='x', labelsize=xtickfontsize)
        ax.tick_params(axis='y', labelsize=ytickfontsize)
        if set_cbar:
            fig.axes[-1].tick_params(labelsize=cbartickfontsize)
            fig.axes[-1].tick_params(labelrotation=cbartickrotation)
            fig.axes[-1].yaxis.label.set_size(cbarlabelfontsize)

        # parse plotting args
        figsize = kwargs.get("figsize", (15, 10))
        nrows = kwargs.get('nrows', len(features))
        ncols = kwargs.get('ncols', 1)
        xlabelfontsize = kwargs.get('xlabelfontsize', 14)
        xtickfontsize = kwargs.get('xtickfontsize', 11)
        ylabelfontsize = kwargs.get('ylabelfontsize', 14)
        ytickfontsize = kwargs.get('ytickfontsize', 11)
        cbartickfontsize = kwargs.get('cbartickfontsize', 14)
        cbartickrotation = kwargs.get('cbartickrotation', 10)
        cbarlabelfontsize = kwargs.get('cbarlabelfontsize', 14)
        rotation_orig = kwargs.get('xticklabelrotation', 25)
        alpha = kwargs.get("alpha", 1)
        x_jitter_orig = kwargs.get("x_jitter", 0.8)
        grouped_features = list(zip_longest(*[iter(features)] * ncols))
```

(continues on next page)
```python
fig, axes = plt.subplots(nrows, ncols, figsize=figsize)
if nrows == len(features):
    axes = list(zip_longest(*[iter(axes)] * 1))

for i, (row, group) in enumerate(zip(axes, grouped_features), start=1):
    # plot each feature or interaction in a subplot
    for ax, feature in zip(row, group):
        # set x-axis ticks and labels and x-jitter for categorical variables
        if not feature:
            continue
        if isinstance(feature, list) or isinstance(feature, tuple):
            feature_index = feature_names.index(feature[0])
        else:
            feature_index = feature_names.index(feature)
        if feature_index in category_map:
            ax.set_xticks(np.arange(len(category_map[feature_index])))
            if i == nrows:
                rotation = 90
            else:
                rotation = rotation_orig
            ax.set_xticklabels(category_map[feature_index], rotation=rotation, fontsize=22)
        else:
            x_jitter = x_jitter_orig

        shap.dependence_plot(feature,
                              shap_values,
                              dataset,
                              feature_names=feature_names,
                              display_features=display_features,
                              interaction_index='auto',
                              ax=ax,
                              show=False,
                              x_jitter=x_jitter,
                              alpha=alpha)

        if i!= nrows:
            ax.tick_params('x', labelrotation=rotation_orig)

_set_fonts(fig, ax, set_cbar=True)

plot_dependence = partial(
    _dependence_plot,
    feature_names=feature_names,
    category_map=category_map,
)

[32]: plot_dependence(['Marital Status', 'Age', 'Hours per week', 'Occupation'],
                      interventional_shap_values,
                      X_test,
                      display_features=X_display,
                      nrows=2,
                      ncols=2,
                      figsize=(22, 10),
                      alpha=0.5)
```

318 Chapter 39. Explaining Tree Models with Interventional Feature Perturbation Tree SHAP
Figure 3: Decision plots of the variables Marital Status, Age, Sex, Race, Occupation, Education using the interventional perturbation Tree SHAP algorithm for the test set

By changing the value of feature below, one can recolour the decision plots according to the interactions estimate from the path-dependent perturbation example. Generally, the same interaction patterns are observed, with the exception of Age, where the interaction with the Capital Gain feature is not conclusive.

```python
[33]: path_dep_interactions = {
    'Marital Status': 'Hours per week',
    'Age': 'Capital Gain',
    'Hours per week': 'Age',
    'Occupation': 'Sex',
}
```

```python
[34]: feature = 'Occupation'
x_jitter = 0.5 if feature in ['Occupation', 'Marital Status'] else 0
shap.dependence_plot(feature,
    interventional_shap_values,
    X_test,
    feature_names=feature_names,
    display_features=X_display,
    interaction_index=path_dep_interactions[feature],
    alpha=0.5,
    x_jitter=x_jitter
)```
If interaction effects are of interest, these can be computed exactly using the path-dependent perturbation algorithm as opposed to approximated.

### 39.4.1 White-box vs black-box model explanations: a comparison with Kernel SHAP

The main drawback of model-agnostic methods such as Kernel SHAP is their sample complexity, which leads to variability in the results obtained. Given enough samples, the feature attributions estimated Kernel SHAP algorithm approach their exact values and give rise to the same feature importance rankings, as shown below.

Below, both the Tree SHAP and Kernel SHAP algorithms are used to explain 100 instances from the test set using a background dataset of 200 samples. For the Kernel SHAP algorithm, each explanation is computed 10 times to account for the variability in the estimation.

```python
n_background_samples = 200
n_explained = 100
background_dataset, y_background = resample(X_train, y_train, n_samples=n_background_samples, replace=False, random_state=0)

X_display_background = decode_data(background_dataset)
X_explain = X_test[:n_explained, :]

tree_explainer = TreeShap(model, model_output='raw', task='classification')
tree_explainer.fit(background_dataset)
explanation = tree_explainer.explain(X_explain)
tree_shap_values = explanation.shap_values[0]
```

XGBoost requires the model inputs to be a DMatrix instance, so `predict_fcn` needs to account for this transformation to avoid errors.
To assess convergence, Kernel SHAP is run with the numbers of samples specified in `n_samples` for `n_runs`.

```python
n_runs = 5
n_samples = [50, 100, 500, 1000, 5000, 10000]
```

```python
results = defaultdict(list)
times = defaultdict(list)
for n_samp in n_samples:
    print(f"Number of samples {n_samp}"
    for run in range(n_runs):
        t_start = timer()
        exp = kernel_explainer.explain(X_explain, nsamples=n_samp)
        t_end = timer()
        times[str(n_samp)].append(t_end - t_start)
        results[str(n_samp)].append(exp.shap_values[0])
results['time'] = times
with open('data/kernel_convergence.pkl', 'wb') as f:
pickle.dump(results, f)
```

To compare the two algorithms, the mean absolute deviation from the ground truth provided by the Tree SHAP algorithm with interventional feature perturbation is computed. For each number of samples, either the maximum mean absolute deviation across the feature, or the mean of this quantity across the features is computed. This calculation can be performed for one instance, or averaged across an entire distribution. The plots below show that all these quantities approach the ground truth values. A threshold of 1% from the effect of the most important feature (Marital Status) is depicted.

```python
def get_errors(tree_shap_values, convergence_data, instance_idx=None):
    """
    Compute the mean and max maximum absolute deviation of Kernel SHAP values
    from Tree SHAP values for a specific instance or as an average over instances.
    If instance_idx is set, then the errors are computed at instance level.
    """
    mad = []
    for key in convergence_data:
        if key != 'time':
            mad.append(np.abs(tree_shap_values - np.mean(convergence_data[key], axis=0)))
    if instance_idx is not None:
        err_max = [max(x[instance_idx, :]) for x in mad]
        err_mean = [np.mean(x[instance_idx, :]).item() for x in mad]
    else:
        err_max = [max(x.mean(axis=0)) for x in mad]
        err_mean = [np.mean(x.mean(axis=0)).item() for x in mad]
```

(continues on next page)
def plot_convergence(err_mean, err_max, n_samples, threshold, instance_idx=None):
    ""
    Plots the average error across the features and the maximum error across
    features as a function of the number of samples Kernel SHAP uses to estimate
    the contributions.
    ""
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
    ax1.loglog(n_samples, err_max, '--*')
    ax1.plot([0] + n_samples, [threshold]*(len(n_samples)+1), '--', color='gray', linewidth='3')
    ax1.grid(True)
    ax1.set_ylabel('Estimation error (max over all features)')
    ax1.set_xlabel('Number of samples')
    ax2.loglog(n_samples, err_mean, '--*')
    ax2.plot([0] + n_samples, [threshold]*(len(n_samples)+1), '--', color='gray', linewidth='3')
    ax2.grid(True)
    ax2.set_ylabel('Estimation error (mean over all features)')
    ax2.set_xlabel('Number of samples')
    if instance_idx is not None:
        plt.suptitle(f'Convergence of the Kernel SHAP algorithm to exact shap values (instance {instance_idx})')
    else:
        plt.suptitle('Convergence of the Kernel SHAP algorithm to exact shap values (mean)')

[35]: threshold = 0.01*np.max(np.mean(np.abs(tree_shap_values), axis=0))

[36]: err_max, err_mean = get_errors(tree_shap_values, convergence_data, instance_idx=0)
    plot_convergence(err_max, err_mean, n_samples, threshold, instance_idx=0)

Figure 4: Converge of Kernel SHAP to true values according to the maximum error (left) and mean error (right) for
Figure 5: Converge of Kernel SHAP according to the maximum error (left) and mean error (right) averaged across 100 instances.

If a high enough number of samples is selected, the algorithms yield the same global patterns, as shown below.

Again, the local accuracy check reveals a large (compare value below vs shap values scale) difference between the expected sum of shap values and the actual sum.
While the Tree SHAP values take a few seconds to compute, the Kernel SHAP takes a few minutes to provide estimates for the shap values. Note that this is also a consequence of the fact that the implementation of Tree SHAP is distributed.

```python
[44]: shap.summary_plot(shap_values_500_tree, X_explained, feature_names, plot_type='bar')
```
Figure 6: Feature importances estimated using the interventional feature perturbation Tree SHAP algorithm

```python
[45]: shap.summary_plot(shap_values_500_kernel, X_explained, feature_names, plot_type='bar')
```

39.4. Explaining xgboost with interventional Tree SHAP: global knowledge from local explanations
Since the errors incurred in estimating the shap values are relatively small, the feature importance rankings shown in Figures 6 and 7 are identical.

The expected values of the two explainers appear to be different between the two explainers despite their use of the background dataset.

This differences arises because Tree Shap takes into account base_score the initial model bias, used by xgboost to make predictions at the start of training. This parameter value does not matter so long the training process succeeds so it can be set to 0 to eliminate the differences.
39.5 Footnotes

(1) The base offset represents the leaf value before training. It can be specified as part of params as 'base_score' and defaults to 0.5.

39.6 References

EXPLAINING TREE MODELS WITH PATH-DEPENDENT FEATURE PERTURBATION TREE SHAP

40.1 Introduction

This example shows how to apply path-dependent feature perturbation Tree SHAP to compute shap values exactly for an xgboost model fitted to the Adult dataset (binary classification task). An example of how to decompose the contribution of any given feature into a main effect and interactions with other features is also presented.

40.2 Data preparation

40.2.1 Load and split

The fetch_adult function returns a Bunch object containing the features, targets, feature names and a mapping of categorical variables to numbers.
Note that for your own datasets you can use the utility function `gen_category_map` imported from alibi.utils.data to create the category map.

```python
[72x694] data_perm = np.random.permutation(np.c_[data, target])
data = data_perm[:, :-1]
target = data_perm[:, -1]
```

Finally, a matrix that contains the raw string values for categorical variables (used for display) is created:

```python
[72x380] def _decode_data(X, feature_names, category_map):
    ""
    Given an encoded data matrix `X` returns a matrix where the
categorical levels have been replaced by human readable categories.
    ""
    X_new = np.zeros(X.shape, dtype=object)
    for idx, name in enumerate(feature_names):
        categories = category_map.get(idx, None)
        if categories:
            for j, category in enumerate(categories):
                encoded_vals = X[:, idx] == j
                X_new[encoded_vals, idx] = category
        else:
            X_new[:, idx] = X[:, idx]
    return X_new
```

```python
[72x355] dtrain = xgb.DMatrix(wrap(X_train), label=wrap(y_train), feature_names=feature_names)
dtest = xgb.DMatrix(wrap(X_test), label=wrap(y_test), feature_names=feature_names)
```

```python
[72x138] decode_data = partial(_decode_data, feature_names=feature_names, category_map)
```

```python
[72x42] X_display = decode_data(X_test)
```
40.3 Model definition

The model fitted in the xgboost fitting example will be explained. The confusion matrix of this model is shown below:

```python
def plot_conf_matrix(y_test, y_pred, class_names):
    
    # place labels at the top
    plt.gca().xaxis.tick_top()
    plt.gca().xaxis.set_label_position('top')

    # plot the matrix per se
    plt.imshow(matrix, interpolation='nearest', cmap=plt.cm.Blues)

    # plot colorbar to the right
    plt.colorbar()

    fmt = 'd'

    # write the number of predictions in each bucket
    thresh = matrix.max() / 2.
    for i, j in product(range(matrix.shape[0]), range(matrix.shape[1])):
        # if background is dark, use a white number, and vice-versa
        plt.text(j, i, format(matrix[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if matrix[i, j] > thresh else "black")

    tick_marks = np.arange(len(class_names))
    plt.xticks(tick_marks, class_names, rotation=45)
    plt.yticks(tick_marks, class_names)
    plt.tight_layout()
    plt.ylabel('True label', size=14)
    plt.xlabel('Predicted label', size=14)
    plt.show()
```

(continues on next page)
```python
def predict(xgb_model, dataset, proba=False, threshold=0.5):
    """
    Predicts labels given a xgboost model that outputs raw logits.
    """

    y_pred = model.predict(dataset)  # raw logits are predicted
    y_pred_proba = invlogit(y_pred)
    if proba:
        return y_pred_proba
    y_pred_class = np.zeros_like(y_pred)
    y_pred_class[y_pred_proba >= threshold] = 1  # assign a label
    return y_pred_class
```

```
[11]: model = xgb.Booster()
model.load_model('adult_xgb.mdl')

[12]: y_pred_train = predict(model, dtrain)
y_pred_test = predict(model, dtest)

[13]: plot_conf_matrix(y_test, y_pred_test, target_names)
```

![Figure 1: Model confusion matrix](image)

```
[14]: print(f'\nTrain accuracy: {round(100*accuracy_score(y_train, y_pred_train), 4)} \%.')
print(f'Test accuracy: {round(100*accuracy_score(y_test, y_pred_test), 4)} \%.')
```

Train accuracy: 87.75 \%.
Test accuracy: 86.6797\%.
40.4 Explaining xgboost via global feature importance

Locally, one could interpret an outcome predicted by a decision tree by analysing the path followed by the sample through the tree (known as the decision path). However, for xgboost the final decision depends on the number of boosting rounds so this technique is not practical. Moreover, this approach only informs one about which features factored in the decision of the algorithm but nothing about the relative importance of the features. Such a view can only be obtained at a global level, for example, by combining information from decision paths of all ensemble members. The xgboost library offers the following measures of feature importance for a feature:

- **weight** - the number of times a feature is used to split the data across all trees
- **gain** - the average gain (that is, contribution to the model output) across all splits the feature is used in
- **cover(l)** - the average coverage across all splits the feature is used in
- **total_gain** - the total gain across all splits the feature is used in
- **total_cover** - the total coverage across all splits the feature is used in.

Therefore, one is first faced with the task of choosing a notion of feature importance before interpreting their model. As shown below, different notions of feature importance lead to different explanations for the same model.

```python
[15]: def _get_importance(model, measure='weight'):
    """
    Retrieves the feature importances from an xgboost models, measured according to the criterion 'measure'.
    """
    imps = model.get_score(importance_type=measure)
    names, vals = list(imps.keys()), list(imps.values())
    sorter = np.argsort(vals)
    s_names, s_vals = tuple(zip(*[(names[i], vals[i]) for i in sorter]))
    return s_vals[::-1], s_names[::-1]

def plot_importance(feat_imp, feat_names, ax=None, **kwargs):
    """
    Create a horizontal barchart of feature effects, sorted by their magnitude.
    """
    left_x, step, right_x = kwargs.get("left_x", 0), kwargs.get("step", 50), kwargs.get("right_x")
    xticks = np.arange(left_x, right_x, step)
    xlabel = kwargs.get("xlabel", 'Feature effects')
    xposfactor = kwargs.get("xposfactor", 1)
    textfont = kwargs.get("text_fontsize", 25) # 16
    yticks_fontsize = kwargs.get("yticks_fontsize", 25)
xlabel_fontsize = kwargs.get("xlabel_fontsize", 30)
textxpos = kwargs.get("textxpos", 60)
textcolor = kwargs.get("textcolor", 'white')

    if ax:
        fig = None
    else:
        fig, ax = plt.subplots(figsize=(10, 5))

    y_pos = np.arange(len(feat_imp))
    ax.barh(y_pos, feat_imp)
    ax.set_yticks(y_pos)
    # (continues on next page)
To demonstrate this, the feature importances obtained when the measures of importance are set to `weight`, `total_gain` and `gain` are plotted below. The difference between the latter two is that the decrease in loss due to a feature is reported as a sum (`total_gain`) and as an average across the splits (`gain`).

```python
[16]: imp_by_weight_v, imp_by_weight_n = get_importance()
imp_by_gain_v, imp_by_gain_n = get_importance(measure='total_gain')
imp_by_a_gain_v, imp_by_a_gain_n = get_importance(measure='gain')
```

```python
[17]: fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(62, 13))
plot_importance(imp_by_weight_v, imp_by_weight_n, ax=ax1, xlabel='Feature effects (weights)', textxpos=45, right_x=1000, step=200)
plot_importance(imp_by_gain_v, imp_by_gain_n, ax=ax2, xlabel='Feature effects (total gain)', textxpos=5, right_x=65000, step=10000, textcolor='black')
plot_importance(imp_by_a_gain_v, imp_by_a_gain_n, ax=ax3, xlabel='Feature effects (gain)', textxpos=0, right_x=250, step=50, textcolor='black')
```

Figure 2: Feature importances as measured by the total number of splits (left), total loss decrease due to feature (middle) and average decrease in loss due to splitting on a particular feature (right)

When using the `weight` criterion for feature importance, all the continuous variables are ranked above categorical ones in terms of feature importance. This occurs because these continuous variables can be split multiple times at different levels in the tree, whereas binary variables such as `Sex` can only be used to partition the data once, so the expected number of splits is smaller for such a variable. To avoid such biases, the feature importance can be quantified by using the total and average gain in information (or, equivalently, decrease in objective). Although the `Marital Status` feature was used to partition the data only 151 times, it contributed the most to decreasing the loss, both across the entire ensemble and when averaged across the splits.

In general, the notion of importance should balance the information gain from making a split on a particular feature with how frequently this feature is used for splitting. Features such as `Age` may have a large cumulative gain courtesy
to them being split on multiple times, but on average they may contribute less to the outcome compared to other features such as Capital Gain which are also split on significant number of times.

However, despite mitigating some of the shortcomings of the split-frequency feature importance, the gain notion of feature-importance suffers from lack of consistency, a property that allows one to compare feature effects across models. The interested reader is referred to this example (page 22) published by Lundberg et al. for details. Such a problem can be mitigated by defining the notion of feature importance with respect to Shapley values, which are consistent as well as faithful to the model (locally).

### 40.5 Explaining xgboost with path-dependent Tree SHAP: global knowledge from local explanations

As described in the overview, the path-dependent feature perturbation Tree SHAP algorithm uses node-level statistics (cover) extracted from the training data in order to estimate the effect of missing features on the model output. Since tree structures also support efficient computation of the model outputs for all possible subsets of missing features, the use of tree paths makes exact shap value estimation possible without a background dataset. In contrast, algorithms such as Kernel SHAP use a background dataset to approximate shap values while interventional feature perturbation Tree SHAP uses a background dataset to compute the effect of missing features on function output and exactly computes the feature contributions given these values.

```python
[18]: path_dependent_explainer = TreeShap(model, model_output='raw', task='classification')
path_dependent_explainer.fit() # does not require background_data
Setting feature_perturbation = "tree_path_dependent" because no background data was given.
Predictor returned a scalar value. Ensure the output represents a probability or decision score as opposed to a classification label!

[18]: TreeShap(meta={
    'name': 'TreeShap',
    'type': ['whitebox'],
    'task': 'classification',
    'explanations': ['local', 'global'],
    'params': {'summarise_background': False, 'kwargs': {}}
})
```

Note that the model_output kwarg was set to raw, to indicate the fact that the model outputs log-odds ratios. This is the only option supported at this moment by this algorithm.

```python
[19]: path_dependent_explanation = path_dependent_explainer.explain(X_test)
path_dependent_shap_values = path_dependent_explanation.shap_values[0]

The shap values computed in this way have the local accuracy property, as expected. That is, they sum to the difference between the model output to be explained and the reference value.

```python
[20]: np.max(np.abs(model.predict(dtest) - path_dependent_explainer.expected_value - path_dependent_shap_values.sum(1)))
[20]: 0.5000074921901536
```

The features which are most important for the predicting whether an individual makes an income greater than $50,000 are shown in Figure 3, where the feature importance of feature $j$ is defined as:

$$ I_j = \frac{1}{N} \sum_{i=1}^{N} |\phi_{i,j}|. $$
Here $N$ is the size of the explained dataset. According to this criterion, the Marital Status feature seems to be the most important, followed by features such as Age or Capital Gain. This global view does not provide information about the direction of the effect at individual level (i.e., whether the prediction that an individual earns more than $50,000 is affected positively or negatively by a particular feature), the magnitude of the effect at individual level (i.e., whether the Marital Status feature, the most important globally, has a significant impact on the prediction about each individual) or the prevalence of a particular effect (how many members of the population are affected in similar ways by a particular feature).

To answer such questions, the same feature ranking can be displayed in a summary plot (Figure 4), which is an aggregation of local explanations. Note that at each feature, points with the same shap value pile up to show density.

Figure 3: Most important features as predicted by the path-dependent perturbation Tree SHAP algorithm

To answer such questions, the same feature ranking can be displayed in a summary plot (Figure 4), which is an aggregation of local explanations. Note that at each feature, points with the same shap value pile up to show density.
from collections import Counter

feat_name = 'Marital Status'
decode_dict = {}
for i, val in enumerate(category_map[feature_names.index(feat_name))):
    decode_dict[i] = val

print("Mapping of feature encoded values to readable values", decode_dict)

Mapping of feature encoded values to readable values {0: 'Married', 1: 'Never-Married', 2: 'Separated', 3: 'Widowed'}

The summary plot shows that being married increases the odds of making more than $50,000 and that, with few exceptions, being widowed decreases the odds of making an income above this threshold. Despite having a significant effect in aggregate, the Age feature does not affect all individuals as significantly: the impact on the prediction of this feature can be significantly negative for young individuals, making it unlikely that young individuals will be predicted to earn more than $50,000. However, while in general elderly tend to be more wealthy, the impact of this feature can be less significant compared to other “less important” features such as Capital Loss, Capital Gain or Education.

The tails in the summary plot of the Capital Loss feature indicate that while this feature is less important than Education or Sex as far as the global model behaviour is concerned, for specific individuals this feature can be a stronger predictor of the income class than the aforementioned features. This granularity in explanations is beyond the reach of traditional methods for tree interpretability.

The vertical spread in the summary plots is indicative of feature interactions, which can be identified approximately, as described in this example, through the shap dependence plot. The Model explanations with Shapley interaction values section shows that Tree SHAP supports exact computation of Shapley interaction values which allow attributing a change in an outcome not only to the features, but also to first order interactions between features.
def _dependence_plot(features, shap_values, dataset, feature_names, category_map, ...
   display_features=None, **kwargs):
   ""
   Plots dependence plots of specified features in a grid.

   features: List[str], List[Tuple[str, str]]
   Names of features to be plotted. If List[str], then shap values are plotted as a function of feature value, coloured by the value of the feature determined to have the strongest interaction (empirically). If List[Tuple[str, str]], shap interaction values are plotted.

display_features: np.ndarray, N x F
   Same as dataset, but contains human readable values for categorical levels as opposed to numerical values
   ""

def _set_fonts(fig, ax, fonts=None, set_cbar=False):
   ""
   Sets fonts for axis labels and colobar.
   ""
   ax.xaxis.label.set_size(xlabelfontsize)
   ax.yaxis.label.set_size(ylabelfontsize)
   ax.tick_params(axis='x', labelsize=xtickfontsize)
   ax.tick_params(axis='y', labelsize=ytickfontsize)
   if set_cbar:
      fig.axes[-1].tick_params(labelsize=cbartickfontsize)
      fig.axes[-1].tick_params(labelrotation=cbartickrotation)
      fig.axes[-1].yaxis.label.set_size(cbarlabelfontsize)

   # parse plotting args
   figsize = kwargs.get("figsize", (15, 10))
   nrows = kwargs.get('nrows', len(features))
   ncols = kwargs.get('ncols', 1)
   xlabelfontsize = kwargs.get('xlabelfontsize', 14)
   xtickfontsize = kwargs.get('xtickfontsize', 11)
   ylabelfontsize = kwargs.get('ylabelfontsize', 14)
   ytickfontsize = kwargs.get('ytickfontsize', 11)
   cbartickfontsize = kwargs.get('cbartickfontsize', 14)
   cbartickrotation = kwargs.get('cbartickrotation', 10)
   cbarlabelfontsize = kwargs.get('cbarlabelfontsize', 14)
   rotation_orig = kwargs.get('xticklabelrotation', 25)
   alpha = kwargs.get("alpha", 1)
   x_jitter_orig = kwargs.get("x_jitter", 0.8)
   grouped_features = list(zip_longest(*[iter(features)] * ncols))

   fig, axes = plt.subplots(nrows, ncols, figsize=figsize)
   if nrows == len(features):
      axes = list(zip_longest(*[iter(axes)] * 1))

   for i, (row, group) in enumerate(zip(axes, grouped_features), start=1):
      # plot each feature or interaction in a subplot
      for ax, feature in zip(row, group):
         # set x-axis ticks and labels and x-jitter for categorical variables
      (continues on next page)
if not feature:
    continue
if isinstance(feature, list) or isinstance(feature, tuple):
    feature_index = feature_names.index(feature[0])
else:
    feature_index = feature_names.index(feature)
if feature_index in category_map:
    ax.set_xticks(np.arange(len(category_map[feature_index])))
    if i == nrows:
        rotation = 90
    else:
        rotation = rotation_orig
    ax.set_xticklabels(category_map[feature_index], rotation=rotation, fontsize=22)
    x_jitter = x_jitter_orig
else:
    x_jitter = 0

shap.dependence_plot(feature, shap_values, dataset, feature_names=feature_names, display_features=display_features, interaction_index='auto', ax=ax, show=False, x_jitter=x_jitter, alpha=alpha)

if i != nrows:
    ax.tick_params('x', labelrotation=rotation_orig)

_set_fonts(fig, ax, set_cbar=True)

def plot_dependence = partial(
    _dependence_plot,
    feature_names=feature_names,
    category_map=category_map,
)

The dependence plots (Figure 5, below) reveal that the strongest interaction of the Marital Status shap values are due to the Hours per week variable. Although the odds for earning in excess of $50,000 are against people who are not married or have separated, they tend to be more favourable for individuals working long hours.

As far as Age is concerned, the odds of earning more increase as a person ages, and, in general, this variable is used by the model to assign individuals to a lower income class. People in their 30s-60s are thought to be more likely to make an income over $50,000 if their capital gains are high. Interestingly, for people over 60, high capital gains have a large negative contribution to the odds of making large incomes, a pattern that is perhaps not intuitive.

As far as the Hours per week is concerned, one sees that older people working no to few hours a week are predicted better odds for making a larger income, and that, up to a certain threshold (of approximately 60 hours), working more than 20 hours increases the odds of a $>\$50,000 prediction for all ages.

Finally, note that not knowing the occupation hurts the odds of predicting a high income. No significant interactions between the sex of the individual (males in red), their occupation and their predicted odds are observed with the exception of, perhaps, Admin and Blue Collar groups.
Performing local explanations across multiple instances efficiently can provide insight into how features contributed to misclassifications and the most common causes of misclassification. This can be achieved by performing a similar analysis for those individuals whose income was predicted below $50,000 but who are known to make an income in excess of this threshold.

```python
[26]: # identify false negatives
misclassified = (np.logical_and(y_test == 1, y_pred_test == 0)).nonzero()[0]
X_misclassified = X_test[misclassified]
# explain the predictions
shap_vals_misclassified = path_dependent_shap_values[misclassified, :]
```

The summary plot indicates that the feature with the most impact on misclassification is Marital Status and that
the model does not correctly capture the fact that individuals who were never married, widowed or separated can also make high incomes.

```
[27]: shap.summary_plot(shap_vals_misclassified, X_misclassified, feature_names )
```

Figure 6: Summary plot of path-dependent perturbation Tree SHAP explanations for individuals misclassified as earning less than $50,000.

```
[28]: X_misclassified_display = decode_data(X_misclassified)
plot_dependence(
    ['Marital Status', 'Age', 'Sex', 'Race', 'Occupation', 'Education'],
    shap_vals_misclassified,
    X_misclassified,
    display_features=X_misclassified_display,
    rotation=33,
    figsize=(47.5, 22),
    alpha=1,
    x_jitter=0.5,
    nrows=3,
    ncols=2,
    xlabelfontsize=24,
    xtickfontsize=20,
    xticklabelrotation=0,
    ylabelfontsize=24,
    ytickfontsize=21,
    cbarlabelfontsize=22,
    cbartickfontsize=20,
    cbartickrotation=0,
)
```
Figure 7: Decision plots of the variables Marital Status, Age, Sex, Race, Occupation, Education using the path-dependent Tree SHAP algorithm for individuals misclassified as earning less than $50,000.

Analysing the plots above reveals that some of the patterns that can lead to misclassification are:

- individuals are not married or are divorced/widowed
- individuals below 40 years old are expected to earn less, across all occupation categories
- individuals are female; being single further increases the odds against the high income class
- racial bias does not seem to be one of the drivers of misclassification, although we can see that for Black people the contribution is slightly negative whereas for white people the contribution is zero
- individuals being Blue-Collar workers, working in Admin jobs, the Service industry or individuals whose occupation is unknown
- individuals having dropped out of education or being high school graduates

40.5.1 Model explanations with Shapley interaction values

As described in the algorithm overview, path-dependent feature perturbation Tree Shap can attribute a change in outcome not only to the $M$ input features, but to the $M$ features and the first-order interactions between them. For each instance to be explained, a tensor of $M \times M$ numbers is returned. The diagonal of this tensor, indexed by $(i, i)$, represents the main effects (i.e., due to the feature itself) whereas the off-diagonal terms indexed by $(i, j)$ represent the interaction between the $i$’th and the $j$’th feature in the input. Summing along the rows of an entry in the Shapley interaction values tensor yields the $M$ shap values for that instance. Note that the interaction value is split equally between each feature so the returned matrix is symmetric: the total interaction effect between feature $i$ and $j$ is therefore obtained by adding the two symmetric entries $(i, j)$ and $(j, i)$.

```
[29]: shap_interactions_explanation = path_dependent_explainer.explain(X_test,
                   interactions=True)
```

```
[30]: shap_interactions_values = shap_interactions_explanation.shap_interaction_values[0]
```
Plots of the interactions between the features Age, Sex, Education and Occupation with Capital Gain are shown below.

```python
[31]: plot_dependence(
    [('Age', 'Capital Gain'),
     ('Sex', 'Capital Gain'),
     ('Education', 'Capital Gain'),
     ('Occupation', 'Capital Gain'),
    ],
    shap_interactions_values,
    X_test,
    figsize=(30,16.5),
    rotation=15,
    ncols=2,
    nrows=2,
    display_features=X_display,
    xtickfontsize=20,
    xlabelfontsize=20,
    ylabelfontsize=20,
    ytickfontsize=17,
    cbarlabelfontsize=20,
    cbartickfontsize=18,
)
```

Figure 11: Shap interaction values for the features Age, Sex, Education and Occupation with Capital Gain

The model has captured the following patterns:

- The interaction between Age and Capital gain increases the odds of predicting an income $>$50,000 for most individuals below 60 years old but significantly decreases the odds for individuals above 60 years old. This interaction has no effect when the individuals don’t have any capital gains.
• For females, capital gains generally increase the prediction odds while for males they decrease them, although these latter interactions are much smaller in magnitude.

• Having a capital gain and education level at Masters and Prof-School or High School grad decreases the prediction odds for higher income.

• For most individuals in occupation categories Professional and Sales, high capital gains slightly reduce the odds of predicting >$50,000. For White-Collar individuals, high capital gain can both increase or decrease the odds.

The `plot_decomposition` function can be used to decompose the shap values of a particular feature into a set of shap values that do not account for the interaction with a specific feature and the interaction values with that specific feature, as shown below. This is depicted in Figure 12.

```python
[32]:
```
def plot_decomposition(feature_pair, shap_interaction_vals, features, feat_names,
                      display_features=None, **kwargs):
    ""
    Given a list containing two feature names (`feature_pair`), an n_instances x n_features tensor of shap interaction values (`shap_interaction_vals`), an n_instances x n_features tensor of feature values and a list of feature names (which assigns a name to each column of `features`), this function plots:
    - left: shap values for feature_pair[0] coloured by the value of feature_pair[1]
    - middle: shap values for feature_pair[0] after subtracting the interaction with feature_pair[1]
    - right: the interaction values between feature_pair[0] and feature_pair[1], which are subtracted from the left plot to get the middle plot.
    NB: `display_features` is the same shape as `features` but should contain the raw categories for categorical variables so that the colorbar can be discretised and the category names displayed alongside the colorbar.
    ""
    def _set_fonts(fig, ax, fonts=None, set_cbar=False):
        ""
        Sets fonts for axis labels and colorbar.
        ""
        ax.xaxis.label.set_size(xlabelfontsize)
        ax.yaxis.label.set_size(ylabelfontsize)
        ax.tick_params(axis='x', labelsize=xtickfontsize)
        ax.tick_params(axis='y', labelsize=ytickfontsize)
        if set_cbar:
            fig.axes[-1].tick_params(labelsize=cbartickfontsize)
            fig.axes[-1].yaxis.label.set_size(cbarlabelfontsize)

    # parse plotting args
    xlabelfontsize = kwargs.get('xlabelfontsize', 21)
    ylabelfontsize = kwargs.get('ylabelfontsize', 21)
    cbartickfontsize = kwargs.get('cbartickfontsize', 16)
    cbarlabelfontsize = kwargs.get('cbarlabelfontsize', 21)
    xtickfontsize = kwargs.get('xtickfontsize', 20)
    ytickfontsize = kwargs.get('ytickfontsize', 20)
    alpha = kwargs.get('alpha', 0.7)
```
```python
figsize = kwargs.get('figsize', (44, 10))
ncols = kwargs.get('ncols', 3)
nrows = kwargs.get('nrows', 1)
# compute shap values and shap values without interaction
feat1_idx = feat_names.index(feature_pair[0])
feat2_idx = feat_names.index(feature_pair[1])
# shap values
shap_vals = shap_interaction_vals.sum(axis=2)
# shap values for feat1, all samples
shap_val_ind1 = shap_interaction_vals[..., feat1_idx].sum(axis=1)
# shap values for (feat1, feat2) interaction
shap_int_ind1_ind2 = shap_interaction_vals[:, feat2_idx, feat1_idx]
# subtract effect of feat2
shap_val_minus_ind2 = shap_val_ind1 - shap_int_ind1_ind2
shap_val_minus_ind2 = shap_val_minus_ind2[:, None]
# create plot
fig, (ax1, ax2, ax3) = plt.subplots(nrows, ncols, figsize=figsize)
# plot the shap values including the interaction
shap.dependence_plot(feature_pair[0],
                      shap_vals,
                      features,
                      display_features = display_features,
                      feature_names=feat_names,
                      interaction_index=feature_pair[1],
                      alpha=alpha,
                      ax=ax1,
                      show=False)
_set_fonts(fig, ax1, set_cbar=True)

# plot the shap values excluding the interaction
shap.dependence_plot(0,
                      shap_val_minus_ind2,
                      features[:, feat1_idx][:, None],
                      feature_names=[feature_pair[0]],
                      interaction_index=None,
                      alpha=alpha,
                      ax=ax2,
                      show=False)
ax2.set_ylabel(f' Shap value for {feature_pair[0]} \n wo {feature_pair[1]} interaction')
_set_fonts(fig, ax2)

# plot the interaction value
shap.dependence_plot(feature_pair,
                      shap_interaction_vals,
                      features,
                      feature_names=feat_names,
                      display_features=display_features,
                      interaction_index='auto',
                      alpha=alpha,
                      ax=ax3,
                      show=False)
```

(continues on next page)
Figure 12: A decomposition of the shap values for Age (left) into shap values for Age excluding the Capital Gain interaction (middle). The total interaction between Age and Capital Gain shown on right.

40.6 Model explanations using xgboost predict method

The xgboost library implements an optimised version of the path-dependent feature perturbation algorithm, which is also internally used by the shap library. xgboost also provides an optimised algorithm for computing the shap interaction values.

The predict method can output the shap values if called as follows:

```python
[34]: xgb_shap_vals = model.predict(dtest, pred_contribs=True)
```

```python
[35]: print(f"shap values shape: {xgb_shap_vals.shape}")
```

```
shap values shape: (2560, 13)
```

Note that there are only 12 features in the dataset. The last column is the expected value with respect to which the feature contributions are computed.

One can also estimate the shap interaction values as follows:

```python
[36]: xgb_shap_interaction_vals = model.predict(dtest, pred_interactions=True)
```

```python
[37]: print(f"shap values shape: {xgb_shap_interaction_vals.shape}")
```

```
shap values shape: (2560, 13, 13)
```

Note that the expected value is again returned in the last column.

The xgboost library also implements an approximate feature attribution method, first described by Sabaas here. This feature attribution method is similar in spirit to Shapley value, but does not account for the effect of variable order as explained here (pp. 10-11). This explanation method can be invoked as follows:
40.7 Footnotes

(1): See the algorithm overview for a brief explanation of coverage.

(2): `model_output=raw` should always be used with the path-dependent perturbation for classification problems in xgboost, irrespective of whether the model is trained with the `binary:logitraw` or `binary:logistic`. Even though a model trained with the latter outputs probabilities, internally xgboost explains the output in margin space due to the `model_output=raw` option. To explain the probability output of a model, one should use the interventional algorithm and pass `model_output=probability` to the constructor along with the objective `binary:logistic` to the training function.
INTEGRATED GRADIENTS FOR A RESNET MODEL TRAINED ON IMAGENET DATASET

In this notebook we apply the integrated gradients method to a pretrained ResNet model trained on the Imagenet dataset. Integrated gradients defines an attribution value for each feature (in this case for each pixel and channel in the image) by integrating the model’s gradients with respect to the input along a straight path from a baseline instance $x'$ to the input instance $x$.

A more detailed description of the method can be found here. Integrated gradients was originally proposed in Sundararajan et al., “Axiomatic Attribution for Deep Networks”

```python
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from alibi.explainers import IntegratedGradients
from tensorflow.keras.applications.resnet_v2 import ResNet50V2
from alibi.datasets import fetch_imagenet
from alibi.utils.visualization import visualize_image_attr

print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # True

TF version:  2.3.1
Eager execution enabled:  True
```

### 41.1 Load data

Load 3 images from the Imagenet dataset, all belonging to the category “Persian cat”.

```python
category = 'Persian cat'
image_shape = (224, 224, 3)
data, labels = fetch_imagenet(category, nb_images=3, target_size=image_shape[:2], seed=2, return_X_y=True)
print(f'Images shape: {data.shape}"
data = (data / 255).astype('float32')
print(data)

Images shape: (3, 224, 224, 3)
```

```python
i = 2
plt.imshow(data[i]);
```
41.2 Load model

Load a pretrained tensorflow model with a ResNet architecture trained on the Imagenet dataset.

```python
[4]: model = ResNet50V2(weights='imagenet')
```

41.3 Calculate integrated gradients

The IntegratedGradients class implements the integrated gradients features attributions method. A description of the method can be found here.

In the first example, the baselines (i.e. the starting points of the path integral) are black images (all pixel values are set to zero). This means that black areas of the image will always have zero attributions. In the second example we consider random uniform noise baselines. The path integral is defined as a straight line from the baseline to the input image. The path is approximated by choosing 50 discrete steps according to the Gauss-Legendre method.

```python
[5]: n_steps = 50
method = "gausslegendre"
internal_batch_size = 50
ig = IntegratedGradients(model,
    n_steps=n_steps,
    method=method,
    internal_batch_size=internal_batch_size)
```

```python
[6]: predictions = model(data).numpy().argmax(axis=1)
explanation = ig.explain(data,
    baselines=None,
    target=predictions)
```

```python
[7]: # Metadata from the explanation object
explanation.meta
```
41.4 Visualize attributions

41.4.1 Black image baseline

Sample image from the test set and its attributions. The attributions are shown by overlaying the attributions values for each pixel to the original image. The attribution value for a pixel is obtained by summing up the attributions values for the three color channels. The attributions are scaled in a $[-1, 1]$ range: red pixels represent negative attributions, while green pixels represent positive attributions. The original image is shown in gray scale for clarity.
41.4.2 Random baselines

Here we show the attributions obtained choosing random uniform noise as a baseline. It can be noticed that the attributions can be considerably different from the previous example, where the black image is taken as a baseline. An extensive discussion about the impact of the baselines on integrated gradients attributions can be found in P. Sturmfels at al., “Visualizing the Impact of Feature Attribution Baselines”.

```python
[11]: baselines = np.random.random_sample(data.shape)

[12]: explanation = ig.explain(data,
                  baselines=baselines,
                  target=predictions)

[13]: attrs = explanation.attributions[0]

Sample image from the test dataset and its attributions. The attributions are shown by overlaying the attributions values for each pixel to the original image. The attribution value for a pixel is obtained by summing up the attributions values for the three color channels. The attributions are scaled in a $[-1, 1]$ range: red pixel represents negative attributions, while green pixels represents positive attributions. The original image is shown in gray scale for clarity.

```python
[14]: fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))
    visualize_image_attr(attr=None, original_image=data[i], method='original_image',
                        title='Original Image', plt_fig_axis=(fig, ax[0]), use_pyplot=False);
    visualize_image_attr(attr=attrs[i], original_image=data[i], method='blended_heat_map',
                         sign='all', show_colorbar=True, title='Overlaid Attributions',
                         plt_fig_axis=(fig, ax[1]), use_pyplot=True);```
41.4. Visualize attributions
INTEGRATED GRADIENTS FOR MNIST

In this notebook we apply the integrated gradients method to a convolutional network trained on the MNIST dataset. Integrated gradients defines an attribution value for each feature of the input instance (in this case for each pixel in the image) by integrating the model’s gradients with respect to the input along a straight path from a baseline instance $x'$ to the input instance $x$.

A more detailed description of the method can be found here. Integrated gradients was originally proposed in Sundararajan et al., “Axiomatic Attribution for Deep Networks”.

```python
import numpy as np
import os
import tensorflow as tf
from tensorflow.keras.layers import Activation, Conv2D, Dense, Dropout
from tensorflow.keras.layers import Flatten, Input, Reshape, MaxPooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical
from alibi.explainers import IntegratedGradients
import matplotlib.pyplot as plt
print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # True
```

**42.1 Load data**

Loading and preparing the MNIST data set.

```python
train, test = tf.keras.datasets.mnist.load_data()
X_train, y_train = train
X_test, y_test = test
test_labels = y_test.copy()
train_labels = y_train.copy()

X_train = X_train.reshape(-1, 28, 28, 1).astype('float64') / 255
X_test = X_test.reshape(-1, 28, 28, 1).astype('float64') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

Output:

```
(60000, 28, 28, 1) (60000, 10) (10000, 28, 28, 1) (10000, 10)
```
42.2 Train model

Train a convolutional neural network on the MNIST dataset. The model includes 2 convolutional layers and it reaches a test accuracy of 0.98. If `save_model = True`, a local folder `./model_mnist` will be created and the trained model will be saved in that folder. If the model was previously saved, it can be loaded by setting `load_mnist_model = True`.

```python
load_mnist_model = False
save_model = True

filepath = './model_mnist/'  # change to directory where model is saved
if load_mnist_model:
    model = tf.keras.models.load_model(os.path.join(filepath, 'model.h5'))
else:
    # define model
    inputs = Input(shape=(X_train.shape[1:]), dtype=tf.float64)
    x = Conv2D(64, 2, padding='same', activation='relu')(inputs)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(.3)(x)
    x = Conv2D(32, 2, padding='same', activation='relu')(x)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(.3)(x)
    x = Flatten()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(.5)(x)
    logits = Dense(10, name='logits')(x)
    outputs = Activation('softmax', name='softmax')(logits)
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    # train model
    model.fit(X_train,
              y_train,
              epochs=6,
              batch_size=256,
              verbose=1,
              validation_data=(X_test, y_test))

if save_model:
    if not os.path.exists(filepath):
        os.makedirs(filepath)
    model.save(os.path.join(filepath, 'model.h5'))
```

356 Chapter 42. Integrated gradients for MNIST
42.3 Calculate integrated gradients

The `IntegratedGradients` class implements the integrated gradients attribution method. A description of the method can be found [here](#).

In the following example, the baselines (i.e. the starting points of the path integral) are black images (all pixel values are set to zero). This means that black areas of the image will always have zero attribution. The path integral is defined as a straight line from the baseline to the input image. The path is approximated by choosing 50 discrete steps according to the Gauss-Legendre method.

```python
# Initialize IntegratedGradients instance
n_steps = 50
method = "gausslegendre"
ig = IntegratedGradients(model,
                        n_steps=n_steps,
                        method=method)
```

```python
# Calculate attributions for the first 10 images in the test set
nb_samples = 10
X_test_sample = X_test[:nb_samples]
predictions = model(X_test_sample).numpy().argmax(axis=1)
explanation = ig.explain(X_test_sample,
                         baselines=None,
                         target=predictions)
```

```python
# Metadata from the explanation object
explanation.meta
```

```python
# Data fields from the explanation object
explanation.data.keys()
```

```python
# Get attributions values from the explanation object
attrs = explanation.attributions[0]
```

42.4 Visualize attributions

Sample images from the test dataset and their attributions.

- The first column shows the original image.
- The second column shows the values of the attributions.
- The third column shows the positive valued attributions.
- The fourth column shows the negative valued attributions.
The attributions are calculated using the black image as a baseline for all samples.

```python
fig, ax = plt.subplots(nrows=3, ncols=4, figsize=(10, 7))
image_ids = [0, 1, 9]
cmap_bound = np.abs(attrs[[0, 1, 9]]).max()

for row, image_id in enumerate(image_ids):
    # original images
    ax[row, 0].imshow(X_test[image_id].squeeze(), cmap='gray')
    ax[row, 0].set_title(f'Prediction: {predictions[image_id]}')

    # attributions
    attr = attrs[image_id]
    im = ax[row, 1].imshow(attr.squeeze(), vmin=-cmap_bound, vmax=cmap_bound, cmap='PiYG')

    # positive attributions
    attr_pos = attr.clip(0, 1)
    im_pos = ax[row, 2].imshow(attr_pos.squeeze(), vmin=-cmap_bound, vmax=cmap_bound, cmap='PiYG')

    # negative attributions
    attr_neg = attr.clip(-1, 0)
    im_neg = ax[row, 3].imshow(attr_neg.squeeze(), vmin=-cmap_bound, vmax=cmap_bound, cmap='PiYG')

ax[0, 1].set_title('Attributions');
ax[0, 2].set_title('Positive attributions');
ax[0, 3].set_title('Negative attributions');

for ax in fig.axes:
    ax.axis('off')

fig.colorbar(im, cax=fig.add_axes([0.95, 0.25, 0.03, 0.5]));
```
42.4. Visualize attributions
Integrated Gradients for Text Classification on the IMDB Dataset

In this example, we apply the integrated gradients method to a sentiment analysis model trained on the IMDB dataset. In text classification models, integrated gradients define an attribution value for each word in the input sentence. The attributions are calculated considering the integral of the model gradients with respect to the word embedding layer along a straight path from a baseline instance \( x' \) to the input instance \( x \). A description of the method can be found here. Integrated gradients was originally proposed in Sundararajan et al., “Axiomatic Attribution for Deep Networks”

The IMDB data set contains 50K movie reviews labelled as positive or negative. We train a convolutional neural network classifier with a single 1-d convolutional layer followed by a fully connected layer. The reviews in the dataset are truncated at 100 words and each word is represented by a 50-dimesional word embedding vector. We calculate attributions for the elements of the embedding layer.

```python
import tensorflow as tf
import numpy as np
import os
import pandas as pd
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Embedding, Conv1D, GlobalMaxPooling1D, Dropout
from tensorflow.keras.utils import to_categorical
from alibi.explainers import IntegratedGradients
import matplotlib.pyplot as plt

print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # True
```

43.1 Load data

Loading the imdb dataset.

```python
max_features = 10000
maxlen = 100

print('Loading data...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
test_labels = y_test.copy()
```

(continues on next page)
train_labels = y_train.copy()
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')
y_train, y_test = to_categorical(y_train), to_categorical(y_test)

print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)

index = imdb.get_word_index()
reverse_index = {value: key for key, value in index.items()}

Loading data...
25000 train sequences
25000 test sequences
Pad sequences (samples x time)
x_train shape: (25000, 100)
x_test shape: (25000, 100)

A sample review from the test set. Note that unknown words are replaced with ‘UNK’

```python
def decode_sentence(x, reverse_index):
    # the `-3` offset is due to the special tokens used by keras
    # see https://stackoverflow.com/questions/42821330/restore-original-text-from-keras-s-imdb-dataset
    return " ".join([reverse_index.get(i - 3, 'UNK') for i in x])
```

```python
print(decode_sentence(x_test[1], reverse_index))
a powerful study of loneliness sexual UNK and desperation be patient UNK up the...

```
```python
if load_model:
    model = tf.keras.models.load_model(os.path.join(filepath, 'model.h5'))
else:
    print('Build model...')
    inputs = Input(shape=(maxlen,), dtype='int32')
    embedded_sequences = Embedding(max_features,
                                    embedding_dims)(inputs)
    out = Conv1D(filters,
                  kernel_size,
                  padding='valid',
                  activation='relu',
                  strides=1)(embedded_sequences)
    out = Dropout(0.4)(out)
    out = GlobalMaxPooling1D()(out)
    out = Dropout(0.4)(out)
    outputs = Dense(2, activation='softmax')(out)
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])
    print('Train...')
    model.fit(x_train, y_train,
               batch_size=256,
               epochs=3,
               validation_data=(x_test, y_test))
    if save_model:
        if not os.path.exists(filepath):
            os.makedirs(filepath)
        model.save(os.path.join(filepath, 'model.h5'))

43.3 Calculate integrated gradients

The integrated gradients attributions are calculated with respect to the embedding layer for 10 samples from the test set. Since the model uses a word to vector embedding with vector dimensionality of 50 and sequence length of 100 words, the dimensionality of the attributions is (10, 100, 50). In order to obtain a single attribution value for each word, we sum all the attribution values for the 50 elements of each word’s vector representation.

The default baseline is used in this example which is internally defined as a sequence of zeros. In this case, this corresponds to a sequence of padding characters \( \text{NB: in general the numerical value corresponding to a “non-informative” baseline such as the PAD token will depend on the tokenizer used, make sure that the numerical value of the baseline used corresponds to your desired token value to avoid surprises}. \) The path integral is defined as a straight line from the baseline to the input image. The path is approximated by choosing 50 discrete steps according to the Gauss-Legendre method.

```
ig = IntegratedGradients(model,
    layer=model.layers[1],
    n_steps=n_steps,
    method=method,
    internal_batch_size=internal_batch_size)

x_test_sample = x_test[:nb_samples]
predictions = model(x_test_sample).numpy().argmax(axis=1)
explanation = ig.explain(x_test_sample,
    baselines=None,
    target=predictions)

# Metadata from the explanation object
print(explanation.meta)

# Data fields from the explanation object
print(explanation.data.keys())

# Get attributions values from the explanation object
print(explanation.attributions[0].shape)

43.4 Sum attributions

print(attribution.sum(axis=2).shape)

43.5 Visualize attributions

i = 1
print('Predicted label = ({}: {})'.format(pred, pred_dict[pred]))
We can visualize the attributions for the text instance by mapping the values of the attributions onto a matplotlib colormap. Below we define some utility functions for doing this.

```python
[18]: from IPython.display import HTML
def hlstr(string, color='white'):
    """
    Return HTML markup highlighting text with the desired color.
    """
    return f"<mark style=background-color:{color}>\{string\}</mark>"

[19]: def colorize(attrs, cmap='PiYG'):
    """
    Compute hex colors based on the attributions for a single instance.
    Uses a diverging colorscale by default and normalizes and scales
    the colormap so that colors are consistent with the attributions.
    """
    import matplotlib as mpl
    cmap_bound = np.abs(attrs).max()
    norm = mpl.colors.Normalize(vmin=-cmap_bound, vmax=cmap_bound)
    cmap = mpl.cm.get_cmap(cmap)
    # now compute hex values of colors
    colors = list(map(lambda x: mpl.colors.rgb2hex(cmap(norm(x))), attrs))
    return colors

Below we visualize the attribution values (highlighted in the text) having the highest positive attributions. Words with high positive attribution are highlighted in shades of green and words with negative attribution in shades of pink. Stronger shading corresponds to higher attribution values. Positive attributions can be interpreted as increase in probability of the predicted class (“Positive sentiment”) while negative attributions correspond to decrease in probability of the predicted class.

```python
[20]: words = decode_sentence(x_i, reverse_index).split()
colors = colorize(attrs_i)

```python
[21]: HTML(""".join(list(map(hlstr, words, colors))))

```python
[21]: <IPython.core.display.HTML object>
44.1 alibi package

44.1.1 Subpackages

alibi.api package

Submodules

alibi.api.defaults module

This module defines the default metadata and data dictionaries for each explanation method. Note that the “name” field is automatically populated upon initialization of the corresponding Explainer class.

```python
alibi.api.defaults.DEFAULT_DATA_ALE: dict = {'ale0': [], 'ale_values': [], 'constant_value': None, 'feature_deciles': None, 'feature_names': None, 'feature_values': [], 'target_names': None}
```
Default ALE data.

```python
alibi.api.defaults.DEFAULT_DATA_ANCHOR: dict = {'anchor': [], 'coverage': None, 'precision': None, 'raw': None}
```
Default anchor data.

```python
alibi.api.defaults.DEFAULT_DATA_ANCHOR_IMG: dict = {'anchor': [], 'coverage': None, 'precision': None, 'raw': None, 'segments': None}
```
Default anchor image data.

```python
alibi.api.defaults.DEFAULT_DATA_CEM = {'PN': None, 'PN_pred': None, 'PP': None, 'PP_pred': None, 'X': None, 'X_pred': None, 'grads_graph': None, 'grads_num': None}
```
Default CEM data.

```python
alibi.api.defaults.DEFAULT_DATA_CF: dict = {'all': [], 'cf': None, 'orig_class': None, 'orig_proba': None, 'success': None}
```
Default counterfactual data.

```python
alibi.api.defaults.DEFAULT_DATA_CFP: dict = {'all': [], 'cf': None, 'id_proto': None, 'orig_class': None, 'orig_proba': None}
```
Default counterfactual prototype metadata.

```python
alibi.api.defaults.DEFAULT_DATA_INTGRAD: dict = {'X': None, 'attributions': None, 'baselines': None, 'deltas': None, 'predictions': None}
```
Default IntegratedGradients data.

```python
alibi.api.defaults.DEFAULT_DATA_KERNEL_SHAP: dict = {'categorical_names': {}, 'expected_value': [], 'feature_names': [], 'raw': {'importances': {}, 'instances': None, 'prediction': None, 'raw_prediction': None}, 'shap_values': []}
```
Default KernelShap data.

```python
alibi.api.defaults.DEFAULT_DATA_TREE_SHAP: dict = {'categorical_names': {}, 'expected_value': [], 'feature_names': [], 'raw': {'importances': {}, 'instances': None, 'loss': None, 'prediction': None, 'raw_prediction': None}, 'shap_interaction_values': [], 'shap_values': []}
```
Default TreeShap data.

```python
alibi.api.defaults.DEFAULT_META_ALE: dict = {'explanations': ['global'], 'name': None, 'params': {}, 'type': ['blackbox']}
```
Default ALE metadata.
alibi.api.defaults.DEFAULT_META_ANCHOR = {'explanations': ['local'], 'name': None, 'params': {}}
Default anchor metadata.

alibi.api.defaults.DEFAULT_META_CEM = {'explanations': ['local'], 'name': None, 'params': {}}
Default CEM metadata.

alibi.api.defaults.DEFAULT_META_CF = {'explanations': ['local'], 'name': None, 'params': {}}
Default counterfactual metadata.

alibi.api.defaults.DEFAULT_META_CFP = {'explanations': ['local'], 'name': None, 'params': {}}
Default counterfactual prototype metadata.

alibi.api.defaults.DEFAULT_META_INTGRAD: dict = {'explanations': ['local'], 'name': None, 'params': {}}
Default IntegratedGradients metadata.

alibi.api.defaults.DEFAULT_META_KERNEL_SHAP: dict = {'explanations': ['local', 'global'], 'name': None, 'params': {'link': None, 'group_names': None, 'grouped': None, 'groups': None, 'weights': None, 'summarise_background': None, 'summarise_result': None, 'transpose': None, 'kwargs': None}, 'task': None, 'type': ['blackbox']}
Default KernelShap metadata.

alibi.api.defaults.DEFAULT_META_TREE_SHAP: dict = {'explanations': ['local', 'global'], 'name': None, 'params': {'algorithm': None, 'model_output': None, 'summarise_background': None, 'summarise_result': None}, 'task': None, 'type': ['whitebox']}
Default TreeShap metadata.

alibi.api.interfaces module

class alibi.api.interfaces.AlibiPrettyPrinter(*args, **kwargs)
   Bases: pprint.PrettyPrinter
   Overrides the built in dictionary pretty representation to look more similar to the external prettyprinter library.

class alibi.api.interfaces.Explainer(meta=NOTHING)
   Bases: abc.ABC
   Base class for explainer algorithms
   
   abstract explain(X)
      
      Return type Explanation

class alibi.api.interfaces.Explanation(meta, data)
   Bases: object
   Explanation class returned by explainers.
   
   __attrs_post_init__()  
      Expose keys stored in self.meta and self.data as attributes of the class.
   
   __getitem__(item)
      This method is purely for deprecating previous behaviour of accessing explanation data via items in the 
      returned dictionary.
   
  classmethod from_json(jsonrepr)
      Create an instance of an Explanation class using a json representation of the Explanation.
      
      Parameters jsonrepr -- json representation of an explanation
      
      Return type Explanation
      
      Returns An Explanation object
to_json()  
Serialize the explanation data and metadata into a json format.  

return_type str  
returns String containing json representation of the explanation

class alibi.api.interfaces.FitMixin  
Bases: abc.ABC  

def fit(self, X)  
return_type Explainer

class alibi.api.interfaces.NumpyEncoder(*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None)  
Bases: json.encoder.JSONEncoder  

default(self, o)  
Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a TypeError).  
For example, to support arbitrary iterators, you could implement default like this:

```python  
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
# Let the base class default method raise the TypeError  
return JSONEncoder.default(self, o)
```

alibi.api.interfaces.default_meta()  
return_type dict

alibi.confidence package

The 'alibi.confidence' module includes trust scores.

alibi.confidence.linearity_measure(preview_fn, x, feature_range=None, method='grid', X_train=None, epsilon=0.04, nb_samples=10, res=100, alphas=None, agg='global', model_type='classifier')  
Calculate the linearity measure of the model around an instance of interest x.

Parameters

- **predict_fn** (Callable) – Predict function.
- **x** (ndarray) – Instance of interest.
- **feature_range** (Union[List, ndarray, None]) – Array with min and max values for each feature.
- **method** (str) – Method for sampling. Supported values ‘knn’ or ‘grid’.
- **X_train** (Optional[ndarray]) – Training set.
- **epsilon** (float) – Size of the sampling region as a percentage of the feature range.
• `nb_samples` (int) – Number of samples to generate.
• `res` (int) – Resolution of the grid. Number of intervals in which the features range is
discretized.
• `alphas` (Optional[ndarray]) – Coefficients in the superposition.
• `agg` (str) – Aggregation method. Supported values ‘global’ or ‘pairwise’.
• `model_type` (str) – Type of task. Supported values ‘regressor’ or ‘classifier’.

Return type  ndarray

Returns  Linearity measure

class alibi.confidence.LinearityMeasure(method='grid', epsilon=0.04, nb_samples=10, res=100, alphas=None, model_type='classifier', agg='pairwise', verbose=False)

Bases: object

__init__(method='grid', epsilon=0.04, nb_samples=10, res=100, alphas=None, model_type='classifier', agg='pairwise', verbose=False)

Parameters

• `method` (str) – Method for sampling. Supported methods are ‘knn’ or ‘grid’.
• `epsilon` (float) – Size of the sampling region around the central instance as a per-
centage of the features range.
• `nb_samples` (int) – Number of samples to generate.
• `res` (int) – Resolution of the grid. Number of intervals in which the feature range is
discretized.
• `alphas` (Optional[ndarray]) – Coefficients in the superposition.
• `agg` (str) – Aggregation method. Supported values are ‘global’ or ‘pairwise’.
• `model_type` (str) – Type of task. Supported values are ‘regressor’ or ‘classifier’.

Return type  None

fit(X_train)

Parameters  X_train (ndarray) – Training set

Return type  None

Returns  None

score(predict_fn, x)

Parameters

• `predict_fn` (Callable) – Prediction function
• `x` (ndarray) – Instance of interest

Return type  ndarray

Returns  Linearity measure

class alibi.confidence.TrustScore(k_filter=10, alpha=0.0, filter_type=None, leaf_size=40, metric='euclidean', dist_filter_type='point')

Bases: object
Initialize trust scores.

**Parameters**

- `k_filter (int)` – Number of neighbors used during either kNN distance or probability filtering.
- `alpha (float)` – Fraction of instances to filter out to reduce impact of outliers.
- `filter_type (Optional[str])` – Filter method; either ‘distance_knn’ or ‘probability_knn’
- `leaf_size (int)` – Number of points at which to switch to brute-force. Affects speed and memory required to build trees. Memory to store the tree scales with n_samples / leaf_size.
- `metric (str)` – Distance metric used for the tree. See sklearn’s DistanceMetric class for a list of available metrics.
- `dist_filter_type (str)` – Use either the distance to the k-nearest point (dist_filter_type = ‘point’) or the average distance from the first to the k-nearest point in the data (dist_filter_type = ‘mean’).

**Return type** None

**filter_by_distance_knn** *(X)*
Filter out instances with low kNN density. Calculate distance to k-nearest point in the data for each instance and remove instances above a cutoff distance.

**Parameters**

- `X (ndarray)` – Data

**Return type** ndarray

**Returns** Filtered data.

**filter_by_probability_knn** *(X, Y)*
Filter out instances with high label disagreement amongst its k nearest neighbors.

**Parameters**

- `X (ndarray)` – Data
- `Y (ndarray)` – Predicted class labels

**Return type** Tuple[ndarray, ndarray]

**Returns** Filtered data and labels.

**fit** *(X, Y, classes=None)*
Build KDTrees for each prediction class.

**Parameters**

- `X (ndarray)` – Data
- `Y (ndarray)` – Target labels, either one-hot encoded or the actual class label.
- `classes (Optional[int])` – Number of prediction classes, needs to be provided if Y equals the predicted class.

**Return type** None

**score** *(X, Y, k=2, dist_type='point')*
Calculate trust scores = ratio of distance to closest class other than the predicted class to distance to predicted class.
Parameters

- **X** (ndarray) – Instances to calculate trust score for.
- **Y** (ndarray) – Either prediction probabilities for each class or the predicted class.
- **k** (int) – Number of nearest neighbors used for distance calculation.
- **dist_type** (str) – Use either the distance to the k-nearest point (dist_type = ‘point’) or the average distance from the first to the k-nearest point in the data (dist_type = ‘mean’).

Return type: Tuple[ndarray, ndarray]

Returns: Batch with trust scores and the closest not predicted class.

Submodules

**aliibi.confidence.model_linearity module**

class aliibi.confidence.model_linearity.LinearityMeasure (method='grid', epsilon=0.04, nb_samples=10, res=100, alphas=None, model_type='classifier', agg='pairwise', verbose=False)

Bases: object

__init__(method='grid', epsilon=0.04, nb_samples=10, res=100, alphas=None, model_type='classifier', agg='pairwise', verbose=False)

Parameters

- **method** (str) – Method for sampling. Supported methods are ‘knn’ or ‘grid’.
- **epsilon** (float) – Size of the sampling region around the central instance as a percentage of the features range.
- **nb_samples** (int) – Number of samples to generate.
- **res** (int) – Resolution of the grid. Number of intervals in which the feature range is discretized.
- **alphas** (Optional[ndarray]) – Coefficients in the superposition.
- **agg** (str) – Aggregation method. Supported values are ‘global’ or ‘pairwise’.
- **model_type** (str) – Type of task. Supported values are ‘regressor’ or ‘classifier’.

Return type: None

**fit** (X_train)

Parameters: X_train (ndarray) – Training set

Return type: None

Returns: None

**score** (predict_fn, x)

Parameters:

- **predict_fn** (Callable) – Prediction function
• \( \mathbf{x} \) (ndarray) – Instance of interest

**Return type** ndarray

**Returns** Linearity measure

```python
alibi.confidence.model_linearity.linearity_measure(predict_fn, x, feature_range=None, method='grid', X_train=None, epsilon=0.04, nb_samples=10, res=100, alphas=None, agg='global', model_type='classifier')
```

Calculate the linearity measure of the model around an instance of interest \( x \).

**Parameters**

• `predict_fn` (*Callable*) – Predict function.

• `x` (ndarray) – Instance of interest.

• `feature_range` (Union[List, ndarray, None]) – Array with min and max values for each feature.

• `method` (str) – Method for sampling. Supported values ‘knn’ or ‘grid’.

• `X_train` (Optional[ndarray]) – Training set.

• `epsilon` (float) – Size of the sampling region as a percentage of the feature range.

• `nb_samples` (int) – Number of samples to generate.

• `res` (int) – Resolution of the grid. Number of intervals in which the features range is discretized.

• `alphas` (Optional[ndarray]) – Coefficients in the superposition.

• `agg` (str) – Aggregation method. Supported values ‘global’ or ‘pairwise’.

• `model_type` (str) – Type of task. Supported values ‘regressor’ or ‘classifier’.

**Return type** ndarray

**Returns** Linearity measure

---

### alibi.confidence.trustscore module

```python
class alibi.confidence.trustscore.TrustScore(k_filter=10, alpha=0.0, filter_type=None, leaf_size=40, metric='euclidean', dist_filter_type='point')
```

**Bases:** object

**__init__**(k_filter=10, alpha=0.0, filter_type=None, leaf_size=40, metric='euclidean', dist_filter_type='point')

Initialize trust scores.

**Parameters**

• `k_filter` (int) – Number of neighbors used during either kNN distance or probability filtering.

• `alpha` (float) – Fraction of instances to filter out to reduce impact of outliers.

• `filter_type` (Optional[str]) – Filter method; either ‘distance_knn’ or ‘probability_knn’
• **leaf_size** *(int)* – Number of points at which to switch to brute-force. Affects speed and memory required to build trees. Memory to store the tree scales with \(n\_samples / \text{leaf\_size}\).

• **metric** *(str)* – Distance metric used for the tree. See sklearn’s DistanceMetric class for a list of available metrics.

• **dist_filter_type** *(str)* – Use either the distance to the k-nearest point \((\text{dist\_filter\_type} = \text{'point'})\) or the average distance from the first to the k-nearest point in the data \((\text{dist\_filter\_type} = \text{'mean'})\).

Return type None

**filter_by_distance_knn** \((X)\)
Filter out instances with low kNN density. Calculate distance to k-nearest point in the data for each instance and remove instances above a cutoff distance.

Parameters

- **X** *(ndarray)* – Data

Return type ndarray

Returns Filtered data.

**filter_by_probability_knn** \((X, Y)\)
Filter out instances with high label disagreement amongst its k nearest neighbors.

Parameters

- **X** *(ndarray)* – Data
- **Y** *(ndarray)* – Predicted class labels

Return type Tuple[ndarray, ndarray]

Returns Filtered data and labels.

**fit** \((X, Y, classes=\text{None})\)
Build KDTrees for each prediction class.

Parameters

- **X** *(ndarray)* – Data
- **Y** *(ndarray)* – Target labels, either one-hot encoded or the actual class label.
- **classes** *(Optional[int]*) – Number of prediction classes, needs to be provided if Y equals the predicted class.

Return type None

**score** \((X, Y, k=2, dist\_type='point')\)
Calculate trust scores = ratio of distance to closest class other than the predicted class to distance to predicted class.

Parameters

- **X** *(ndarray)* – Instances to calculate trust score for.
- **Y** *(ndarray)* – Either prediction probabilities for each class or the predicted class.
- **k** *(int)* – Number of nearest neighbors used for distance calculation.
- **dist\_type** *(str)* – Use either the distance to the k-nearest point \((\text{dist\_type} = \text{'point'})\) or the average distance from the first to the k-nearest point in the data \((\text{dist\_type} = \text{'mean'})\).

Return type Tuple[ndarray, ndarray]
**Returns**  
*Batch with trust scores and the closest not predicted class.*

### alibi.explainers package

The `alibi.explainers` module includes feature importance, counterfactual and anchor-based explainers.

#### class  
`alibi.explainers.ALE(predictor, feature_names=None, target_names=None, check_feature_resolution=True, low_resolution_threshold=10, extrapolate_constant=True, extrapolate_constant_perc=10.0, extrapolate_constant_min=0.1)`  
*Bases: alibi.api.interfaces.Explainer*

__init__ (predictor, feature_names=None, target_names=None, check_feature_resolution=True, low_resolution_threshold=10, extrapolate_constant=True, extrapolate_constant_perc=10.0, extrapolate_constant_min=0.1)

Accumulated Local Effects for tabular datasets. Current implementation supports first order feature effects of numerical features.

**Parameters**

- **predictor** (*Callable*) – A callable that takes in an NxF array as input and outputs an NxT array (N - number of data points, F - number of features, T - number of outputs/targets (e.g. 1 for single output regression, >=2 for classification).

- **feature_names** (*Optional[List[str]]*) – A list of feature names used for displaying results.

- **target_names** (*Optional[List[str]]*) – A list of target/output names used for displaying results.

- **check_feature_resolution** (*bool*) – If true, the number of unique values is calculated for each feature and if it is less than `low_resolution_threshold` then the feature values are used for gridpoints instead of quantiles. This may increase the runtime of the algorithm for large datasets.

- **low_resolution_threshold** (*int*) – If a feature has at most this many unique values, these are used as the grid points instead of quantiles. This is to avoid situations when the quantile algorithm returns quantiles between discrete values which can result in jumps in the ALE plot obscuring the true effect. Only used if `check_feature_resolution` is True.

- **extrapolate_constant** (*bool*) – If a feature is constant, only one quantile exists where all the data points lie. In this case the ALE value at that point is zero, however this may be misleading if the feature does have an effect on the model. If this parameter is set to True, the ALE values are calculated on an interval surrounding the constant value. The interval length is controlled by the `extrapolate_constant_perc` and `extrapolate_constant_min` arguments.

- **extrapolate_constant_perc** (*float*) – Percentage by which to extrapolate a constant feature value to create an interval for ALE calculation. If $q$ is the constant feature value, creates an interval $[q - q/\text{extrapolate_constant_perc}, q + q/\text{extrapolate_constant_perc}]$ for which ALE is calculated. Only relevant if `extrapolate_constant` is set to True.

- **extrapolate_constant_min** (*float*) – Controls the minimum extrapolation length for constant features. An interval constructed for constant features is guaranteed to be $2*\text{extrapolate_constant_min}$ wide centered on the feature value. This allows for
capturing model behaviour around constant features which have small value so that extrapolate_constant_perc is not so helpful. Only relevant if extrapolate_constant is set to True.

Return type None

build_explanation(ale_values, ale0, constant_value, feature_values, feature_deciles, feature_names)
Helper method to build the Explanation object.

Return type Explanation

explain(X, features=None, min_bin_points=4)
Calculate the ALE curves for each feature with respect to the dataset X.

Parameters
- X(ndarray) – An NxF tabular dataset used to calculate the ALE curves. This is typically the training dataset or a representative sample.
- features(Optional[List[int]]) – Features for which to calculate ALE.
- min_bin_points(int) – Minimum number of points each discretized interval should contain to ensure more precise ALE estimation.

Return type Explanation

Returns An Explanation object containing the data and the metadata of the calculated ALE curves.

class alibi.explainers.AnchorTabular(predictor, feature_names, categorical_names=None, seed=None)
Bases: alibi.api.interfaces.Explainer, alibi.api.interfaces.FitMixin

__init__(predictor, feature_names, categorical_names=None, seed=None)
Parameters
- predictor(Callable) – A callable that takes a tensor of N data points as inputs and returns N outputs.
- feature_names(list) – List with feature names.
- categorical_names(Optional[dict]) – Dictionary where keys are feature columns and values are the categories for the feature.
- seed(Optional[int]) – Used to set the random number generator for repeatability purposes.

Return type None

add_names_to_exp(explanation)
Add feature names to explanation dictionary.

Parameters explanation(dict) – Dict with anchors and additional metadata.

Return type None

build_explanation(X, result, predicted_label, params)
Preprocess search output and return an explanation object containing metadata.

Parameters
- X(ndarray) – Instance to be explained.
- result(dict) – Dictionary with explanation search output and metadata.
• **predicted_label** *(int)* – Label of the instance to be explained (inferred if not given).
• **params** *(dict)* – Parameters passed to *explain*

**Return type** *Explanation*

**Returns**

• *Explanation* object containing human readable explanation, metadata, and precision/coverage
• *info* as attributes.

```python
explain(X, threshold=0.95, delta=0.1, tau=0.15, batch_size=100, coverage_samples=10000, beam_size=1, stop_on_first=False, max_anchor_size=None, min_samples_start=100, n_covered_ex=10, binary_cache_size=10000, cache_margin=1000, verbose=False, verbose_every=1, **kwargs)
```

Explain prediction made by classifier on instance X.

**Parameters**

• **X** *(ndarray)* – Instance to be explained.
• **threshold** *(float)* – Minimum precision threshold.
• **delta** *(float)* – Used to compute beta.
• **tau** *(float)* – Margin between lower confidence bound and minimum precision or upper bound.
• **batch_size** *(int)* – Batch size used for sampling.
• **coverage_samples** *(int)* – Number of samples used to estimate coverage from during result search.
• **beam_size** *(int)* – The number of anchors extended at each step of new anchors construction.
• **stop_on_first** *(bool)* – If True, the beam search algorithm will return the first anchor that has satisfies the probability constraint.
• **max_anchor_size** *(Optional[int]*) – Maximum number of features in result.
• **min_samples_start** *(int)* – Min number of initial samples.
• **n_covered_ex** *(int)* – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with desired_label are stored).
• **binary_cache_size** *(int)* – The result search pre-allocates binary_cache_size batches for storing the binary arrays returned during sampling.
• **cache_margin** *(int)* – When only max(cache_margin, batch_size) positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.
• **verbose** *(bool)* – Display updates during the anchor search iterations.
• **verbose_every** *(int)* – Frequency of displayed iterations during anchor search process.

**Return type** *Explanation*

**Returns** *explanation* – *Explanation* object containing the result explaining the instance with additional metadata as attributes.
fit \((\text{train\_data, disc\_perc}=(25, 50, 75), **\text{kwargs})\)

Fit discretizer to train data to bin numerical features into ordered bins and compute statistics for numerical features. Create a mapping between the bin numbers of each discretised numerical feature and the row id in the training set where it occurs.

Parameters

- \textbf{train\_data} (ndarray) – Representative sample from the training data.
- \textbf{disc\_perc} (Tuple[Union[int, float], ...]) – List with percentiles (int) used for discretization.

Return type \texttt{AnchorTabular}

class \texttt{alibi.explainers.DistributedAnchorTabular} \((\text{predictor, feature\_names, categorical\_names=None, seed=None})\)

Bases: \texttt{alibi.explainers.anchor\_tabular.AnchorTabular}

explain \((X, \text{threshold}=0.95, \text{delta}=0.1, \text{tau}=0.15, \text{batch\_size}=100, \text{coverage\_samples}=10000, \text{beam\_size}=1, \text{stop\_on\_first}=False, \text{max\_anchor\_size}=None, \text{min\_samples\_start}=1, \text{n\_covered\_ex}=10, \text{binary\_cache\_size}=10000, \text{cache\_margin}=1000, \text{verbose}=False, \text{verbose\_every}=1, **\text{kwargs})\)

Explains the prediction made by a classifier on instance \(X\). Sampling is done in parallel over a number of cores specified in kwargs["ncpu"].

Parameters superclass implementation. \texttt{(See –)}

Return type \texttt{Explanation}

Returns See superclass implementation.

fit \((\text{train\_data, disc\_perc}=(25, 50, 75), **\text{kwargs})\)

Creates a list of handles to parallel processes handles that are used for submitting sampling tasks.

Parameters superclass implementation. \texttt{(See –)}

Return type \texttt{AnchorTabular}

class \texttt{alibi.explainers.AnchorText} \((\text{nlp, predictor, seed=None})\)

Bases: \texttt{alibi.api.interfaces.Explainer}

\texttt{UNK = 'UNK'}

\texttt{\_\_init\_\_} \((\text{nlp, predictor, seed=None})\)

Initialize anchor text explainer.

Parameters

- \textbf{nlp} (\texttt{spacy.language.Language}) – spaCy object.
- \textbf{predictor} (\texttt{Callable}) – A callable that takes a tensor of \(N\) data points as inputs and returns \(N\) outputs.
- \textbf{seed} (\texttt{int}) – If set, ensures identical random streams.

Return type \texttt{None}

build\_explanation \((\text{text, result, predicted\_label, params})\)

Uses the metadata returned by the anchor search algorithm together with the instance to be explained to build an explanation object.

Parameters

- \textbf{text} (\texttt{str}) – Instance to be explained.
- \textbf{result} (\texttt{dict}) – Dictionary containing the search result and metadata.
• **predicted_label** *(int)* – Label of the instance to be explained. Inferred if not received.

• **params** *(dict)* – Parameters passed to `explain`

**Return type** *Explanation*

### compare_labels *(samples)*

Compute the agreement between a classifier prediction on an instance to be explained and the prediction on a set of samples which have a subset of features fixed to a given value (aka compute the precision of anchors).

**Parameters** *samples* *(ndarray)* – Samples whose labels are to be compared with the instance label.

**Return type** *ndarray*

**Returns** *A boolean array indicating whether the prediction was the same as the instance label.*

### explain *(text, use_unk=True, use_similarity_proba=False, sample_proba=0.5, top_n=100, temperature=1.0, threshold=0.95, delta=0.1, tau=0.15, batch_size=100, coverage_samples=10000, beam_size=1, stop_on_first=True, max_anchor_size=None, min_samples_start=100, n_covered_ex=10, binary_cache_size=10000, cache_margin=1000, verbose=False, verbose_every=1, **kwargs)*

Explain instance and return anchor with metadata.

**Parameters**

• **text** *(str)* – Text instance to be explained.

• **use_unk** *(bool)* – If True, perturbation distribution will replace words randomly with UNKS. If False, words will be replaced by similar words using word embeddings.

• **use_similarity_proba** *(bool)* – Sample according to a similarity score with the corpus embeddings use_unk needs to be False in order for this to be used.

• **sample_proba** *(float)* – Sample probability if use_similarity_proba is False.

• **top_n** *(int)* – Number of similar words to sample for perturbations, only used if use_unk=False.

• **temperature** *(float)* – Sample weight hyperparameter if use_similarity_proba equals True.

• **threshold** *(float)* – Minimum precision threshold.

• **delta** *(float)* – Used to compute beta.

• **tau** *(float)* – Margin between lower confidence bound and minimum precision or upper bound.

• **batch_size** *(int)* – Batch size used for sampling.

• **coverage_samples** *(int)* – Number of samples used to estimate coverage from during anchor search.

• **beam_size** *(int)* – Number of options kept after each stage of anchor building.

• **stop_on_first** *(bool)* – If True, the beam search algorithm will return the first anchor that has satisfies the probability constraint.

• **max_anchor_size** *(Optional[int])* – Maximum number of features to include in an anchor.

• **min_samples_start** *(int)* – Number of samples used for anchor search initialisation.
• **n_covered_ex** *(int)* – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with predicted label are stored).

• **binary_cache_size** *(int)* – The anchor search pre-allocates binary_cache_size batches for storing the boolean arrays returned during sampling.

• **cache_margin** *(int)* – When only max(cache_margin, batch_size) positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.

• **kwargs** *(Any)* – Other keyword arguments passed to the anchor beam search and the text sampling and perturbation functions.

• **verbose** *(bool)* – Display updates during the anchor search iterations.

• **verbose_every** *(int)* – Frequency of displayed iterations during anchor search process.

**Return type** *Explanation*

**Returns** *explanation* – *Explanation* object containing the anchor explaining the instance with additional metadata as attributes.

**find_similar_words()**

This function queries a spaCy nlp model to find n similar words with the same part of speech for each word in the instance to be explained. For each word the search procedure returns a dictionary containing an np.array of words (‘words’) and an np.array of word similarities (‘similarities’).

**Return type** *None*

**perturb_sentence**(present, n, sample_proba=0.5, forbidden=frozenset({}), forbidden_tags=frozenset({'PRP$'}), forbidden_words=frozenset({'be'}), temperature=1.0, pos=frozenset({'ADJ', 'ADP', 'ADV', 'DET', 'NOUN', 'VERB'}), use_similarity_proba=True)

Perturb the text instance to be explained.

**Parameters**

• **present** *(tuple)* – Word index in the text for the words in the proposed anchor.

• **n** *(int)* – Number of samples used when sampling from the corpus.

• **sample_proba** *(float)* – Sample probability for a word if use_similarity_proba is False.

• **forbidden** *(frozenset)* – Forbidden lemmas.

• **forbidden_tags** *(frozenset)* – Forbidden POS tags.

• **forbidden_words** *(frozenset)* – Forbidden words.

• **pos** *(frozenset)* – POS that can be changed during perturbation.

• **use_similarity_proba** *(bool)* – Bool whether to sample according to a similarity score with the corpus embeddings.

• **temperature** *(float)* – Sample weight hyperparameter if use_similarity_proba equals True.

**Return type** *Tuple[ndarray, ndarray]*

**Returns**

• **raw_data** – Array of perturbed text instances.
• `data` – Matrix with 1s and 0s indicating whether a word in the text has not been perturbed for each sample.

`sampler(anchor, num_samples, compute_labels=True)`

Generate perturbed samples while maintaining features in positions specified in anchor unchanged.

**Parameters**

- `anchor` ([`int, tuple`]) – int: the position of the anchor in the input batch tuple: the anchor itself, a list of words to be kept unchanged
- `num_samples` (`int`) – Number of generated perturbed samples.
- `compute_labels` (`bool`) – If True, an array of comparisons between predictions on perturbed samples and instance to be explained is returned.

**Return type** `Union[List[Union[ndarray, float, int]], List[ndarray]]`

**Returns**

- *If compute_labels=True, a list containing the following is returned*
  - `covered_true`: perturbed examples where the anchor applies and the model prediction on perturbation is the same as the instance prediction
  - `covered_false`: perturbed examples where the anchor applies and the model prediction is NOT the same as the instance prediction
  - `labels`: `num_samples` ints indicating whether the prediction on the perturbed sample matches (1) the label of the instance to be explained or not (0)
  - `data`: Matrix with 1s and 0s indicating whether a word in the text has been perturbed for each sample
    - 1.0: indicates exact coverage is not computed for this algorithm
    - anchor[0]: position of anchor in the batch request
  - *Otherwise, a list containing the data matrix only is returned.*

`set_data_type(use_unk)`

Working with numpy arrays of strings requires setting the data type to avoid truncating examples. This function estimates the longest sentence expected during the sampling process, which is used to set the number of characters for the samples and examples arrays. This depends on the perturbation method used for sampling.

**Parameters**

- `use_unk` (`bool`) – See explain method.

**Return type** None

`set_sampler_perturbation(use_unk, perturb_opts, top_n)`

Initialises the explainer by setting the perturbation function and parameters necessary to sample according to the perturbation method.

**Parameters**

- `use_unk` (`bool`) – see explain method
- `perturb_opts` (`dict`) – A dict with keys:
  - `top_n`: the max number of alternatives to sample from for replacement
  - `use_similarity_proba`: if True the probability of selecting a replacement word is proportional to the similarity between the word and the word to be replaced
- 'sample_proba': given a feature and n sentences, this parameter is the mean of a Bernoulli distribution used to decide how many sentences will have that feature perturbed.
- 'temperature': a temperature used to calibrate the softmax distribution over the sampling weights.

- **top_n (int)** – Number of similar words to sample for perturbations, only used if `use_unk=False`.

Return type None

**set_words_and_pos** *(text)*

Process the sentence to be explained into spaCy token objects, a list of words, punctuation marks and a list of positions in input sentence.

Parameters **text** *(str)* – The instance to be explained.

Return type None

### class alibi.explainers.AnchorImage

**init** *(predictor, image_shape, segmentation_fn='slic', segmentation_kwargs=None, images_background=None, seed=None)*

Initialize anchor image explainer.

Parameters

- **predictor** *(Callable)* – A callable that takes a tensor of N data points as inputs and returns N outputs.
- **image_shape** *(tuple)* – Shape of the image to be explained.
- **segmentation_fn** *(Any)* – Any of the built-in segmentation function strings: ‘felzenszwalb’, ‘slic’ or ‘quickshift’ or a custom segmentation function (callable) which returns an image mask with labels for each superpixel. See [http://scikit-image.org/docs/dev/api/skimage.segmentation.html](http://scikit-image.org/docs/dev/api/skimage.segmentation.html) for more info.
- **segmentation_kwargs** *(Optional[dict])* – Keyword arguments for the built-in segmentation functions.
- **images_background** *(Optional[ndarray])* – Images to overlay superpixels on.
- **seed** *(Optional[int])* – If set, ensures different runs with the same input will yield same explanation.

Return type None

**build_explanation** *(image, result, predicted_label, params)*

Uses the metadata returned by the anchor search algorithm together with the instance to be explained to build an explanation object.

Parameters

- **image** *(ndarray)* – Instance to be explained.
- **result** *(dict)* – Dictionary containing the search anchor and metadata.
- **predicted_label** *(int)* – Label of the instance to be explained.
- **params** *(dict)* – Parameters passed to `explain`
Return type  **Explanation**

**compare_labels** *(samples)*

Compute the agreement between a classifier prediction on an instance to be explained and the prediction on a set of samples which have a subset of perturbed superpixels.

**Parameters**

- **samples** *(ndarray)* – Samples whose labels are to be compared with the instance label.

**Return type**  **ndarray**

**Returns**

A boolean array indicating whether the prediction was the same as the instance label.

**explain** *(image, p_sample=0.5, threshold=0.95, delta=0.1, tau=0.15, batch_size=100, coverage_samples=10000, beam_size=1, stop_on_first=False, max_anchor_size=None, min_samples_start=100, n_covered_ex=10, binary_cache_size=10000, cache_margin=1000, verbose=False, verbose_every=1, **kwargs)*

Explain instance and return anchor with metadata.

**Parameters**

- **image** *(ndarray)* – Image to be explained.
- **p_sample** *(float)* – Probability for a pixel to be represented by the average value of its superpixel.
- **threshold** *(float)* – Minimum precision threshold.
- **delta** *(float)* – Used to compute beta.
- **tau** *(float)* – Margin between lower confidence bound and minimum precision of upper bound.
- **batch_size** *(int)* – Batch size used for sampling.
- **coverage_samples** *(int)* – Number of samples used to estimate coverage from during result search.
- **beam_size** *(int)* – The number of anchors extended at each step of new anchors construction.
- **stop_on_first** *(bool)* – If True, the beam search algorithm will return the first anchor that has satisfies the probability constraint.
- **max_anchor_size** *(Optional[int])* – Maximum number of features in result.
- **min_samples_start** *(int)* – Min number of initial samples.
- **n_covered_ex** *(int)* – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with desired_label are stored).
- **binary_cache_size** *(int)* – The result search pre-allocates binary_cache_size batches for storing the binary arrays returned during sampling.
- **cache_margin** *(int)* – When only max(cache_margin, batch_size) positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.
- **verbose** *(bool)* – Display updates during the anchor search iterations.
- **verbose_every** *(int)* – Frequency of displayed iterations during anchor search process.

**Return type**  **Explanation**
Returns explanation – Explanation object containing the anchor explaining the instance with additional metadata as attributes.

**generate_superpixels** (*image*)
Generates superpixels from (i.e., segments) an image.

Parameters image (ndarray) – A grayscale or RGB image.

Return type ndarray

Returns A \([H, W]\) array of integers. Each integer is a segment (superpixel) label.

**overlay_mask** (*image, segments, mask_features, scale=(0, 255)*)
Overlay image with mask described by the mask features.

Parameters

- image (ndarray) – Image to be explained.
- segments (ndarray) – Superpixels
- mask_features (list) – List with superpixels present in mask.
- scale (tuple) – Pixel scale for masked image.

Return type ndarray

Returns masked_image – Image overlaid with mask.

**perturbation** (*anchor, num_samples*)
Perturbs an image by altering the values of selected superpixels. If a dataset of image backgrounds is provided to the explainer, then the superpixels are replaced with the equivalent superpixels from the background image. Otherwise, the superpixels are replaced by their average value.

Parameters

- anchor (tuple) – Contains the superpixels whose values are not going to be perturbed.
- num_samples (int) – Number of perturbed samples to be returned.

Return type Tuple[ndarray, ndarray]

Returns

- imgs – A \([\text{num_samples}, H, W, C]\) array of perturbed images.
- segments_mask – A \([\text{num_samples}, M]\) binary mask, where \(M\) is the number of image superpixels segments. 1 indicates the values in that particular superpixels are not perturbed.

**sampler** (*anchor, num_samples, compute_labels=True*)
Sample images from a perturbation distribution by masking randomly chosen superpixels from the original image and replacing them with pixel values from superimposed images if background images are provided to the explainer. Otherwise, the superpixels from the original image are replaced with their average values.

Parameters

- anchor (Tuple[int, tuple]) – int: order of anchor in the batch tuple: features (= superpixels) present in the proposed anchor
- num_samples (int) – Number of samples used
- compute_labels (bool) – If True, an array of comparisons between predictions on perturbed samples and instance to be explained is returned.
Return type Union[List[Union[ndarray, float, int]], List[ndarray]]

Returns

- If compute_labels=True, a list containing the following is returned –
  - covered_true: perturbed examples where the anchor applies and the model prediction on perturbed is the same as the instance prediction
  - covered_false: perturbed examples where the anchor applies and the model prediction on perturbed sample is NOT the same as the instance prediction
  - labels: num_samples ints indicating whether the prediction on the perturbed sample matches (1) the label of the instance to be explained or not (0)
  - data: Matrix with 1s and 0s indicating whether the values in a superpixel will remain unchanged (1) or will be perturbed (0), for each sample
  - 1.0: indicates exact coverage is not computed for this algorithm
  - anchor[0]: position of anchor in the batch request
- Otherwise, a list containing the data matrix only is returned.

class alibi.explainers.CEM(predict, mode, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0, 10000000000.0), gamma=0.0, ae_model=None, learning_rate_init=0.01, max_iterations=1000, c_init=10.0, c_steps=10, eps=(0.001, 0.001), clip=(-100.0, 100.0), update_num_grad=1, no_info_val=None, write_dir=None, sess=None)


Parameters

- predict (Union[Callable, Model, ForwardRef]) – Keras or TensorFlow model or any other model’s prediction function returning class probabilities
- mode (str) – Find pertinent negatives (‘PN’) or pertinent positives (‘PP’)
- shape (tuple) – Shape of input data starting with batch size
- kappa (float) – Confidence parameter for the attack loss term
- beta (float) – Regularization constant for L1 loss term
- feature_range (tuple) – Tuple with min and max ranges to allow for perturbed instances. Min and max ranges can be floats or numpy arrays with dimension (1x nb of features) for feature-wise ranges
- gamma (float) – Regularization constant for optional auto-encoder loss term
- ae_model (Union[Model, ForwardRef]) – Optional auto-encoder model used for loss regularization
- learning_rate_init (float) – Initial learning rate of optimizer
- max_iterations (int) – Maximum number of iterations for finding a PN or PP
- c_init (float) – Initial value to scale the attack loss term
- **c_steps** (*int*) – Number of iterations to adjust the constant scaling the attack loss term
- **eps** (*tuple*) – If numerical gradients are used to compute \(\frac{dL}{dx} = (\frac{dL}{dp}) \cdot (\frac{dp}{dx})\), then eps[0] is used to calculate \(\frac{dL}{dp}\) and eps[1] is used for \(\frac{dp}{dx}\). eps[0] and eps[1] can be a combination of float values and numpy arrays. For eps[0], the array dimension should be (1 x nb of prediction categories) and for eps[1] it should be (1 x nb of features)
- **clip** (*tuple*) – Tuple with min and max clip ranges for both the numerical gradients and the gradients obtained from the TensorFlow graph
- **update_num_grad** (*int*) – If numerical gradients are used, they will be updated every update_num_grad iterations
- **no_info_val** (*Union[float, ndarray]*) – Global or feature-wise value considered as containing no information
- **write_dir** (*str*) – Directory to write tensorboard files to
- **sess** (*Session*) – Optional Tensorflow session that will be used if passed instead of creating or inferring one internally

**Return type** None

**attack** *(X, Y, verbose=False)*
Find pertinent negative or pertinent positive for instance X using a fast iterative shrinkage-thresholding algorithm (FISTA).

**Parameters**
- **X** (*ndarray*) – Instance to attack
- **Y** (*ndarray*) – Labels for X
- **verbose** (*bool*) – Print intermediate results of optimization if True

**Return type** *Tuple[*ndarray*, *Tuple[*ndarray*, *ndarray*]]*

**Returns** Overall best attack and gradients for that attack.

**explain** *(X, Y=None, verbose=False)*
Explain instance and return PP or PN with metadata.

**Parameters**
- **X** (*ndarray*) – Instances to attack
- **Y** (*Optional[*ndarray*]*) – Labels for X
- **verbose** (*bool*) – Print intermediate results of optimization if True

**Return type** *Explanation*

**Returns** explanation – *Explanation* object containing the PP or PN with additional metadata as attributes.

**fit** *(train_data, no_info_type='median')*
Get ‘no information’ values from the training data.

**Parameters**
- **train_data** (*ndarray*) – Representative sample from the training data
- **no_info_type** (*str*) – Median or mean value by feature supported

**Return type** *CEM*
get_gradients \( (X, Y) \)

Compute numerical gradients of the attack loss term: \( \frac{dL}{dx} = \frac{dL}{dP} \cdot \frac{dP}{dx} \) with \( L = \text{loss}_{\text{attack}} \); \( P = \text{predict} \); \( x = \text{adv}_s \)

**Parameters**

- \( X \) (ndarray) – Instance around which gradient is evaluated
- \( Y \) (ndarray) – One-hot representation of instance labels

**Return type** ndarray

**Returns** Array with gradients.

loss_fn \( (\text{pred_proba}, Y) \)

Compute the attack loss.

**Parameters**

- \( \text{pred_proba} \) (ndarray) – Prediction probabilities of an instance
- \( Y \) (ndarray) – One-hot representation of instance labels

**Return type** ndarray

**Returns** Loss of the attack.

perturb \( (X, \text{eps}, \text{proba}=\text{False}) \)

Apply perturbation to instance or prediction probabilities. Used for numerical calculation of gradients.

**Parameters**

- \( X \) (ndarray) – Array to be perturbed
- \( \text{eps} \) (Union[\text{float}, \text{ndarray}]) – Size of perturbation
- \( \text{proba} \) (bool) – If True, the net effect of the perturbation needs to be 0 to keep the sum of the probabilities equal to 1

**Return type** Tuple[ndarray, ndarray]

**Returns** Instances where a positive and negative perturbation is applied.

class alibi.explainers.CounterFactual \( (\text{predict_fn}, \text{shape}, \text{distance_fn}=\text{’l1’}, \text{target_proba}=1.0, \text{target_class}=\text{’other’}, \text{max_iter}=1000, \text{early_stop}=50, \text{lam_init}=0.1, \text{max_lam_steps}=10, \text{tol}=0.05, \text{learning_rate_init}=0.1, \text{feature_range}=(-1000000000.0, 1000000000.0), \text{eps}=0.01, \text{init}=\text{’identity’}, \text{decay}=\text{True}, \text{write_dir}=\text{None}, \text{debug}=\text{False}, \text{sess}=\text{None}) \)

**Bases:** alibi.api.interfaces.Explainer

**__init__** \( (\text{predict_fn}, \text{shape}, \text{distance_fn}=\text{’l1’}, \text{target_proba}=1.0, \text{target_class}=\text{’other’}, \text{max_iter}=1000, \text{early_stop}=50, \text{lam_init}=0.1, \text{max_lam_steps}=10, \text{tol}=0.05, \text{learning_rate_init}=0.1, \text{feature_range}=(-1000000000.0, 1000000000.0), \text{eps}=0.01, \text{init}=\text{’identity’}, \text{decay}=\text{True}, \text{write_dir}=\text{None}, \text{debug}=\text{False}, \text{sess}=\text{None}) \)

Initialize counterfactual explanation method based on Wachter et al. (2017)

**Parameters**

- \( \text{predict_fn} \) (Union[Callable, Model, ForwardRef]) – Keras or TensorFlow model or any other model’s prediction function returning class probabilities
- \( \text{shape} \) (Tuple[int, ...]) – Shape of input data starting with batch size
- \( \text{distance_fn} \) (str) – Distance function to use in the loss term
alibi Documentation, Release 0.5.6

- **target_proba (float)** – Target probability for the counterfactual to reach
- **target_class (Union[str, int])** – Target class for the counterfactual to reach, one of 'other', 'same' or an integer denoting desired class membership for the counterfactual instance
- **max_iter (int)** – Maximum number of interations to run the gradient descent for (inner loop)
- **early_stop (int)** – Number of steps after which to terminate gradient descent if all or none of found instances are solutions
- **lam_init (float)** – Initial regularization constant for the prediction part of the Wachter loss
- **max_lam_steps (int)** – Maximum number of times to adjust the regularization constant (outer loop) before terminating the search
- **tol (float)** – Tolerance for the counterfactual target probability
- **learning_rate_init** – Initial learning rate for each outer loop of lambda
- **feature_range (Union[ Tuple, str])** – Tuple with min and max ranges to allow for perturbed instances. Min and max ranges can be floats or numpy arrays with dimension (1 x nb of features) for feature-wise ranges
- **eps (Union[float, ndarray])** – Gradient step sizes used in calculating numerical gradients, defaults to a single value for all features, but can be passed an array for feature-wise step sizes
- **init (str)** – Initialization method for the search of counterfactuals, currently must be 'identity'
- **decay (bool)** – Flag to decay learning rate to zero for each outer loop over lambda
- **write_dir (str)** – Directory to write Tensorboard files to
- **debug (bool)** – Flag to write Tensorboard summaries for debugging
- **sess (Session)** – Optional Tensorflow session that will be used if passed instead of creating or inferring one internally

**Return type** None

**explain (X)**

Explain an instance and return the counterfactual with metadata.

**Parameters**

- X (ndarray) – Instance to be explained

**Return type** Explanation

**Returns**

Explanation object containing the counterfactual with additional metadata as attributes.

**fit (X, y)**

Fit method - currently unused as the counterfactual search is fully unsupervised.

**Return type** CounterFactual
class alibi.explainers.CounterFactualProto(predict, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0, 10000000000.0), gamma=0.0, ae_model=None, enc_model=None, theta=0.0, cat_vars=None, ohe=False, use_kdtree=False, learning_rate_init=0.01, max_iterations=1000, c_init=10.0, c_steps=10, eps=(0.001, 0.001), clip=(-1000.0, 1000.0), update_num_grad=1, write_dir=None, sess=None)

Bases: alibi.api.interfaces.Explainer, alibi.api.interfaces.FitMixin

__init__(predict, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0, 10000000000.0), gamma=0.0, ae_model=None, enc_model=None, theta=0.0, cat_vars=None, ohe=False, use_kdtree=False, learning_rate_init=0.01, max_iterations=1000, c_init=10.0, c_steps=10, eps=(0.001, 0.001), clip=(-1000.0, 1000.0), update_num_grad=1, write_dir=None, sess=None)

Initialize prototypical counterfactual method.

Parameters

- **predict** (Union[Callable, Model, ForwardRef]) – Keras or TensorFlow model or any other model’s prediction function returning class probabilities
- **shape** (tuple) – Shape of input data starting with batch size
- **kappa** (float) – Confidence parameter for the attack loss term
- **beta** (float) – Regularization constant for L1 loss term
- **feature_range** (tuple) – Tuple with min and max ranges to allow for perturbed instances. Min and max ranges can be floats or numpy arrays with dimension (1x nb of features) for feature-wise ranges
- **gamma** (float) – Regularization constant for optional auto-encoder loss term
- **ae_model** (Union[Model, ForwardRef]) – Optional auto-encoder model used for loss regularization
- **enc_model** (Union[Model, ForwardRef]) – Optional encoder model used to guide instance perturbations towards a class prototype
- **theta** (float) – Constant for the prototype search loss term
- **cat_vars** (dict) – Dict with as keys the categorical columns and as values the number of categories per categorical variable.
- **ohe** (bool) – Whether the categorical variables are one-hot encoded (OHE) or not. If not OHE, they are assumed to have ordinal encodings.
- **use_kdtree** (bool) – Whether to use k-d trees for the prototype loss term if no encoder is available
- **learning_rate_init** (float) – Initial learning rate of optimizer
- **max_iterations** (int) – Maximum number of iterations for finding a counterfactual
- **c_init** (float) – Initial value to scale the attack loss term
- **c_steps** (int) – Number of iterations to adjust the constant scaling the attack loss term
• **eps** (tuple) – If numerical gradients are used to compute $dL/dx = (dL/dp) \times (dp/dx)$, then eps[0] is used to calculate $dL/dp$ and eps[1] is used for $dp/dx$. eps[0] and eps[1] can be a combination of float values and numpy arrays. For eps[0], the array dimension should be (1x nb of prediction categories) and for eps[1] it should be (1x nb of features)

• **clip** (tuple) – Tuple with min and max clip ranges for both the numerical gradients and the gradients obtained from the TensorFlow graph

• **update_num_grad** (int) – If numerical gradients are used, they will be updated every update_num_grad iterations

• **write_dir** (str) – Directory to write tensorboard files to

• **sess** (Session) – Optional Tensorflow session that will be used if passed instead of creating or inferring one internally

**Return type** None

**attack** *(X, Y, target_class=None, k=None, k_type='mean', threshold=0.0, verbose=False, print_every=100, log_every=100)*

Find a counterfactual (CF) for instance X using a fast iterative shrinkage-thresholding algorithm (FISTA).

**Parameters**

• **X** (ndarray) – Instance to attack

• **Y** (ndarray) – Labels for X as one-hot-encoding

• **target_class** (Optional[list]) – List with target classes used to find closest prototype. If None, the nearest prototype except for the predict class on the instance is used.

• **k** (Optional[int]) – Number of nearest instances used to define the prototype for a class. Defaults to using all instances belonging to the class if an encoder is used and to 1 for k-d trees.

• **k_type** (str) – Use either the average encoding of the k nearest instances in a class (k_type='mean') or the k-nearest encoding in the class (k_type='point') to define the prototype of that class. Only relevant if an encoder is used to define the prototypes.

• **threshold** (float) – Threshold level for the ratio between the distance of the counterfactual to the prototype of the predicted class for the original instance over the distance to the prototype of the predicted class for the counterfactual. If the trust score is below the threshold, the proposed counterfactual does not meet the requirements.

• **verbose** (bool) – Print intermediate results of optimization if True

• **print_every** (int) – Print frequency if verbose is True

• **log_every** (int) – Tensorboard log frequency if write directory is specified

**Return type** Tuple[ndarray, Tuple[ndarray, ndarray]]

**Returns** Overall best attack and gradients for that attack.

**explain** *(X, Y=None, target_class=None, k=None, k_type='mean', threshold=0.0, verbose=False, print_every=100, log_every=100)*

Explain instance and return counterfactual with metadata.

**Parameters**

• **X** (ndarray) – Instances to attack

• **Y** (Optional[ndarray]) – Labels for X as one-hot-encoding
• **target_class** *(Optional)* List with target classes used to find closest prototype. If None, the nearest prototype except for the predict class on the instance is used.

• **k** *(Optional)* Number of nearest instances used to define the prototype for a class. Defaults to using all instances belonging to the class if an encoder is used and to 1 for k-d trees.

• **k_type** *(str)* Use either the average encoding of the k nearest instances in a class (k_type='mean') or the k-nearest encoding in the class (k_type='point') to define the prototype of that class. Only relevant if an encoder is used to define the prototypes.

• **threshold** *(float)* Threshold level for the ratio between the distance of the counterfactual to the prototype of the predicted class for the original instance over the distance to the prototype of the predicted class for the counterfactual. If the trust score is below the threshold, the proposed counterfactual does not meet the requirements.

• **verbose** *(bool)* – Print intermediate results of optimization if True

• **print_every** *(int)* – Print frequency if verbose is True

• **log_every** *(int)* – Tensorboard log frequency if write directory is specified

**Return type** Explanation

**Returns** explanation – Explanation object containing the counterfactual with additional metadata as attributes.

**fit** *(train_data, trustscore_kwargs=None, d_type='abdm', w=None, disc_perc=(25, 50, 75), standardize_cat_vars=False, smooth=1.0, center=True, update_feature_range=True)*

Get prototypes for each class using the encoder or k-d trees. The prototypes are used for the encoder loss term or to calculate the optional trust scores.

**Parameters**

• **train_data** *(ndarray)* – Representative sample from the training data.

• **trustscore_kwargs** *(Optional)* – Optional arguments to initialize the trust scores method.

• **d_type** *(str)* – Pairwise distance metric used for categorical variables. Currently, ‘abdm’, ‘mvdm’ and ‘abdm-mvdm’ are supported. ‘abdm’ infers context from the other variables while ‘mvdm’ uses the model predictions. ‘abdm-mvdm’ is a weighted combination of the two metrics.

• **w** *(Optional)* Weight on ‘abdm’ (between 0. and 1.) distance if d_type equals ‘abdm-mvdm’.

• **disc_perc** *(Sequence)* – List with percentiles used in binning of numerical features used for the ‘abdm’ and ‘abdm-mvdm’ pairwise distance measures.

• **standardize_cat_vars** *(bool)* – Standardize numerical values of categorical variables if True.

• **smooth** *(float)* – Smoothing exponent between 0 and 1 for the distances. Lower values of l will smooth the difference in distance metric between different features.

• **center** *(bool)* – Whether to center the scaled distance measures. If False, the min distance for each feature except for the feature with the highest raw max distance will be the lower bound of the feature range, but the upper bound will be below the max feature range.

• **update_feature_range** *(bool)* – Update feature range with scaled values.
Return type  \texttt{CounterFactualProto}

\texttt{get\_gradients}(X, Y, \texttt{grads\_shape}, \texttt{cat\_vars\_ord=None})

Compute numerical gradients of the attack loss term: \( \frac{dL}{dx} = (\frac{dL}{dP})(\frac{dP}{dx}) \) with \( L = \text{loss\_attack\_s} \); \( P = \text{predict} \); \( x = \text{adv\_s} \)

Parameters

- \( X \) (ndarray) \quad \text{Instance around which gradient is evaluated}
- \( Y \) (ndarray) \quad \text{One-hot representation of instance labels}
- \( \text{grads\_shape} \) (tuple) \quad \text{Shape of gradients.}
- \( \text{cat\_vars\_ord} \) (Optional[dict]) \quad \text{Dict with as keys the categorical columns and as values the number of categories per categorical variable.}

Return type  \texttt{ndarray}

Returns  \text{Array with gradients.}

\texttt{loss\_fn}(\texttt{pred\_proba}, Y)

Compute the attack loss.

Parameters

- \( \text{pred\_proba} \) (ndarray) \quad \text{Prediction probabilities of an instance}
- \( Y \) (ndarray) \quad \text{One-hot representation of instance labels}

Return type  \texttt{ndarray}

Returns  \text{Loss of the attack.}

\texttt{score}(X, \texttt{adv\_class}, \texttt{orig\_class}, \texttt{eps=1e-10})

Parameters

- \( X \) (ndarray) \quad \text{Instance to encode and calculate distance metrics for}
- \( \text{adv\_class} \) (int) \quad \text{Predicted class on the perturbed instance}
- \( \text{orig\_class} \) (int) \quad \text{Predicted class on the original instance}
- \( \text{eps} \) (float) \quad \text{Small number to avoid dividing by 0}

Return type  \texttt{float}

Returns

- \text{Ratio between the distance to the prototype of the predicted class for the original instance and}
- \text{the prototype of the predicted class for the perturbed instance.}

\textbf{class}  \texttt{alibi.explainers.KernelShap}(\texttt{predictor}, \texttt{link='identity'}, \texttt{feature\_names=None}, \texttt{categorical\_names=None}, \texttt{task='classification'}, \texttt{seed=None}, \texttt{distributed\_opts=None})

\texttt{Bases: alibi.api.interfaces.Explainer, alibi.api.interfaces.FitMixin}

\texttt{__init__}(\texttt{predictor}, \texttt{link='identity'}, \texttt{feature\_names=None}, \texttt{categorical\_names=None}, \texttt{task='classification'}, \texttt{seed=None}, \texttt{distributed\_opts=None})

A wrapper around the \texttt{shap.KernelExplainer} class. It extends the current \texttt{shap} library functionality by allowing the user to specify variable groups in order to treat one-hot encoded categorical as one during sampling. The user can also specify whether to aggregate the \texttt{shap} values estimate for the encoded levels of categorical variables as an optional argument to \texttt{explain}, if grouping arguments are not passed to \texttt{fit}.

Parameters
• **predictor** (*Callable*) – A callable that takes as an input a samples x features array and outputs a samples x n_outputs model outputs. The n_outputs should represent model output in margin space. If the model outputs probabilities, then the link should be set to ‘logit’ to ensure correct force plots.

• **link** (*str*) – Valid values are ‘identity’ or ‘logit’. A generalized linear model link to connect the feature importance values to the model output. Since the feature importance values, $\phi$, sum up to the model output, it often makes sense to connect them to the output with a link function where $\text{link} (\text{output} - \text{expected_value}) = \text{sum} (\phi)$. Therefore, for a model which outputs probabilities, link='logit' makes the feature effects have log-odds (evidence) units and link='identity' means that the feature effects have probability units. Please see this example for an in-depth discussion about the semantics of explaining the model in the probability or margin space.

• **feature_names** (*Union[List[str], Tuple[str], None]*) – Used to infer group names when categorical data is treated by grouping and group_names input to fit is not specified, assuming it has the same length as the groups argument of fit method. It is also used to compute the names field, which appears as a key in each of the values of explanation.data['raw']['importances'].

• **categorical_names** (*Optional[Dict[int, List[str]]]*) – Keys are feature column indices in the background_data matrix (see fit). Each value contains strings with the names of the categories for the feature. Used to select the method for background data summarisation (if specified, subsampling is performed as opposed to k-means clustering). In the future it may be used for visualisation.

• **task** (*str*) – Can have values ‘classification’ and ‘regression’. It is only used to set the contents of explanation.data['raw']['prediction']

• **seed** (*Optional[int]*) – Fixes the random number stream, which influences which subsets are sampled during shap value estimation.

• **distributed_opts** (*Optional[Dict]*) – A dictionary that controls the algorithm distributed execution. See DISTRIBUTED_OPTS documentation for details.

**build_explanation**(*X, shap_values, expected_value, **kwargs*)

Create an explanation object. If output summarisation is required and all inputs necessary for this operation are passed, the raw shap values are summed first so that a single shap value is returned for each categorical variable, as opposed to a shap value per dimension of categorical variable encoding.

Parameters

• **X** (*Union[ndarray, DataFrame, spmatrix]*) – Instances to be explained.

• **shap_values** (*List[ndarray]*) – Each entry is a n_instances x n_features array, and the length of the list equals the dimensionality of the predictor output. The rows of each array correspond to the shap values for the instances with the corresponding row index in X. The length of the list equals the number of model outputs.

• **expected_value** (*List[float]*) – A list containing the expected value of the prediction for each class. Its length should be equal to that of shap_values.

Return type **Explanation**

Returns

**explanation** – An explanation object containing the shap values and prediction in the data field, along with a meta field containing additional data. See usage examples for details.

**explain**(*X, summarise_result=False, cat_vars_start_idx=None, cat_vars_enc_dim=None, **kwargs*)

Explains the instances in the array X.
Parameters

- **X** (Union[ndarray, DataFrame, spmatrix]) – Instances to be explained.
- **summarise_result** (bool) – Specifies whether the shap values corresponding to dimensions of encoded categorical variables should be summed so that a single shap value is returned for each categorical variable. Both the start indices of the categorical variables (**cat_vars_start_idx**) and the encoding dimensions (**cat_vars_enc_dim**) have to be specified.
- **cat_vars_start_idx** (Optional[Sequence[int]]) – The start indices of the categorical variables. If specified, **cat_vars_enc_dim** should also be specified.
- **cat_vars_enc_dim** (Optional[Sequence[int]]) – The length of the encoding dimension for each categorical variable. If specified **cat_vars_start_idx** should also be specified.
- **kwargs** – Keyword arguments specifying explain behaviour. Valid arguments are:
  - **nsamples**: controls the number of predictor calls and therefore runtime.
  - **l1_reg**: the algorithm is exponential in the feature dimension. If set to `auto` the algorithm will first run a feature selection algorithm to select the top features, provided the fraction of sampled sets of missing features is less than 0.2 from the number of total subsets. The Akaike Information Criterion is used in this case. See our examples for more details about available settings for this parameter. Note that by first running a feature selection step, the shapley values of the remainder of the features will be different to those estimated from the entire set.

  For more details, please see the shap library documentation.

Return type: **Explanation**

Returns

- **explanation** – An explanation object containing the shap values and prediction in the **data** field, along with a **meta** field containing additional data. See usage examples for details.

**fit** *(background_data, summarise_background=False, n_background_samples=300, group_names=None, groups=None, weights=None, **kwargs)*

This takes a background dataset (usually a subsample of the training set) as an input along with several user specified options and initialises a **KernelShap** explainer. The runtime of the algorithm depends on the number of samples in this dataset and on the number of features in the dataset. To reduce the size of the dataset, the **summarise_background** option and `n_background_samples` should be used. To reduce the feature dimensionality, encoded categorical variables can be treated as one during the feature perturbation process; this decreases the effective feature dimensionality, can reduce the variance of the shap values estimation and reduces slightly the number of calls to the predictor. Further runtime savings can be achieved by changing the **nsamples** parameter in the call to explain. Runtime reduction comes with an accuracy trade-off, so it is better to experiment with a runtime reduction method and understand results stability before using the system.

Parameters

- **background_data** (Union[ndarray, spmatrix, DataFrame, Data]) – Data used to estimate feature contributions and baseline values for force plots. The rows of the background data should represent samples and the columns features.
- **summarise_background** (Union[bool, str]) – A large background dataset impacts the runtime and memory footprint of the algorithm. By setting this argument to **True**, only `n_background_samples` from the provided data are selected. If **group_names** or **groups** arguments are specified, the algorithm assumes that the data
contains categorical variables so the records are selected uniformly at random. Otherwise, `shap.kmeans` (a wrapper around `sklearn` k-means implementation) is used for selection. If set to ‘auto’, a default of `KERNEL_SHAP_BACKGROUND_THRESHOLD` samples is selected.

- **n_background_samples** (`int`) – The number of samples to keep in the background dataset if `summarise_background=True`.

- **groups** (`Optional[List[Union[Tuple[int], List[int]]]]`) – A list containing sub-lists specifying the indices of features belonging to the same group.

- **group_names** (`Union[List[str], Tuple[str], None]`) – If specified, this array is used to treat groups of features as one during feature perturbation. This feature can be useful, for example, to treat encoded categorical variables as one and can result in computational savings (this may require adjusting the `nsamples` parameter).

- **weights** (`Union[List[float], Tuple[float], ndarray, None]`) – A sequence or array of weights. This is used only if grouping is specified and assigns a weight to each point in the dataset.

- **kwargs** – Expected keyword arguments include `keep_index` (bool) and should be used if a data frame containing an index column is passed to the algorithm.

Return type: `KernelShap`

```python
class alibi.explainers.TreeShap(predictor, model_output='raw', feature_names=None, categorical_names=None, task='classification', seed=None)
```

Bases: `alibi.api.interfaces.Explainer`, `alibi.api.interfaces.FitMixin`

__init__ (predictor, model_output='raw', feature_names=None, categorical_names=None, task='classification', seed=None)

A wrapper around the `shap.TreeExplainer` class. It adds the following functionality:

1. Input summarisation options to allow control over background dataset size and hence runtime
2. Output summarisation for sklearn models with one-hot encoded categorical variables.

Users are strongly encouraged to familiarise themselves with the algorithm by reading the method overview in the documentation.

Parameters

- **predictor** (`Any`) – A fitted model to be explained. XGBoost, LightGBM, CatBoost and most tree-based scikit-learn models are supported. In the future, Pyspark could also be supported. Please open an issue if this is a use case for you.

- **model_output** (`str`) – Supported values are: ‘raw’, ‘probability’, ‘probability_doubled’, ‘log_loss’:
  
  - ‘raw’: the raw model of the output, which varies by task, is explained. This option should always be used if the `fit` is called without arguments. It should also be set to compute shap model values. For regression models it is the standard output, for binary classification in XGBoost it is the log odds ratio.

  - ‘probability’: the probability output is explained. This option should only be used if `fit` was called with the `background_data` argument set. The effect of specifying this parameter is that the `shap` library will use this information to transform the shap values computed in margin space (aka using the raw output) to shap values that sum to the probability output by the model plus the model expected output probability. This requires knowledge of the type of output for `predictor` which is inferred by the `shap` library from the model type (e.g., most sklearn models with exception of `sklearn.tree.DecisionTreeClassifier`, `sklearn.ensemble.RandomForestClassifier`, `sklearn.linear_model.LogisticRegression`, etc.).
sklearn.ensemble.ExtraTreesClassifier output logits) or on the basis of the mapping implemented in the `shap.TreeEnsemble` constructor. Only trees that output log odds and probabilities are supported currently.

- `probability_doubled`: used for binary classification problem in situations where the model outputs the logits/probabilities for the positive class but shap values for both outcomes are desired. This option should be used only if `fit` was called with the `background_data` argument set. In this case the expected value for the negative class is 1 - expected_value for positive class and the shap values for the negative class are the negative values of the positive class shap values. As before, the explanation happens in the margin space, and the shap values are subsequently adjusted. convert the model output to probabilities. The same considerations as for probability apply for this output type too.

- `log_loss`: logarithmic loss is explained. This option should be used only if `fit` was called with the `background_data` argument set and requires specifying labels, `y`, when calling `explain`. If the objective is squared error, then the transformation $(output - y)^2$ is applied. For binary cross-entropy objective, the transformation $\log(1 + \exp(output)) - y * output$ with $y \in \{0, 1\}$. Currently only binary cross-entropy and squared error losses can be explained.

- `feature_names` (Union[List[str], Tuple[str], None]) – Used to compute the `names` field, which appears as a key in each of the values of the `importances` sub-field of the response raw field.

- `categorical_names` (Optional[Dict[int, List[str]]]) – Keys are feature column indices. Each value contains strings with the names of the categories for the feature. Used to select the method for background data summarisation (if specified, subsampling is performed as opposed to kmeans clustering). In the future it may be used for visualisation.

- `task` (str) – Can have values ‘classification’ and ‘regression’. It is only used to set the contents of the prediction field in the data['raw'] response field.

**Notes**

Tree SHAP is an additive attribution method so it is best suited to explaining output in margin space (the entire real line). For discussion related to explaining models in output vs probability space, please consult this resource.

`build_explanation` (*X*, `shap_output`, `expected_value`, **kwargs)

Create an explanation object. If output summarisation is required and all inputs necessary for this operation are passed, the raw shap values are summed first so that a single shap value is returned for each categorical variable, as opposed to a shap value per dimension of categorical variable encoding. Similarly, the shap interaction values are summed such that they represent the interaction between categorical variables as opposed to levels of categorical variables. If the interaction option has been specified during `explain`, this method computes the shap values given the interactions prior to creating the response.

**Parameters**

- `X` (Union[ndarray, DataFrame, ForwardRef]) – Instances to be explained.

- `shap_output` (List[ndarray]) – If `explain` is called with `interactions=True` then the list contains tensors of dimensionality `n_instances x n_features x n_features` of shap interaction values. Otherwise, it contains tensors of dimension `n_instances x n_features` representing shap values. The length of the list equals the number of model outputs.
- **expected_value** (*List[float]*) – A list containing the expected value of the prediction for each class. Its length is equal to that of `shap_output`.

**Return type** *Explanation*

**Returns**

`explanation` – An *Explanation* object containing the shap values and prediction in the *data* field, along with a *meta* field containing additional data. See usage examples [here](#) for details.

`explain`(*X*, *y=None*, *interactions=False*, *approximate=False*, *check_additivity=True*, *tree_limit=None*, *summarise_result=False*, *cat_vars_start_idx=None*, *cat_vars_enc_dim=None*, **kwargs)

Explains the instances in *X*. *y* should be passed if the model loss function is to be explained, which can be useful in order to understand how various features affect model performance over time. This is only possible if the explainer has been fitted with a background dataset and requires setting `model_output='log_loss'`.

**Parameters**

- *X* (*Union*[ndarray, DataFrame, ForwardRef]*) – Instances to be explained.
- *y* (*Optional*[ndarray]*) – Labels corresponding to rows of *X*. Should be passed only if a background dataset was passed to the *fit* method.
- *interactions* (*bool*) – If True, the shap value for every feature of every instance in *X* is decomposed into $(X.shape[1] - 1)$ shap value interactions and one main effect. This is only supported if *fit* is called with `background_dataset=None`.
- *approximate* (*bool*) – If True, an approximation to the shap values that does not account for feature order is computed. This was proposed by Ando Sabaas [here](#). Check this resource for more details. This option is currently only supported for *xgboost* and *sklearn* models.
- *check_additivity* (*bool*) – If True, output correctness is ensured if `model_output='raw'` has been passed to the constructor.
- *tree_limit* (*Optional*[int]*) – Explain the output of a subset of the first *tree_limit* trees in an ensemble model.
- *summarise_result* (*bool*) – This should be set to True only when some of the columns in *X* represent encoded dimensions of a categorical variable and one single shap value per categorical variable is desired. Both *cat_vars_start_idx* and *cat_vars_enc_dim* should be specified as detailed below to allow this.
- *cat_vars_start_idx* (*Optional*[Sequence[int]]) – The start indices of the categorical variables.
- *cat_vars_enc_dim* (*Optional*[Sequence[int]]) – The length of the encoding dimension for each categorical variable.

**Return type** *Explanation*

**Returns**

`explanation` – An *Explanation* object containing the shap values and prediction in the *data* field, along with a *meta* field containing additional data. See usage examples [here](#) for details.

`fit`(*background_data=None*, *summarise_background=False*, *n_background_samples=1000*, **kwargs)

This function instantiates an explainer which can then be use to explain instances using the `explain`
method. If no background dataset is passed, the explainer uses the path-dependent feature perturbation algorithm to explain the values. As such, only the model raw output can be explained and this should be reflected by passing `model_output='raw'` when instantiating the explainer. If a background dataset is passed, the interventional feature perturbation algorithm is used. Using this algorithm, probability outputs can also be explained. Additionally, if the `model_output='log_loss'` option is passed to the explainer constructor, then the model loss function can be explained by passing the labels as the `y` argument to the explain method. A limited number of loss functions are supported, as detailed in the constructor documentation.

### Parameters

- **background_data** *(Union[ndarray, DataFrame, None])* – Data used to estimate feature contributions and baseline values for force plots. The rows of the background data should represent samples and the columns features.

- **summarise_background** *(Union[bool, str])* – A large background dataset may impact the runtime and memory footprint of the algorithm. By setting this argument to `True`, only `n_background_samples` from the provided data are selected. If the `categorical_names` argument has been passed to the constructor, subsampling of the data is used. Otherwise, `shap.kmeans` (a wrapper around `sklearn.kmeans` implementation) is used for selection. If set to `'auto'`, a default of `TREE_SHAP_BACKGROUND_WARNING_THRESHOLD` samples is selected.

- **n_background_samples** *(int)* – The number of samples to keep in the background dataset if `summarise_background=True`.

### Return type

`TreeShap`

### alibi.explainers.plot_ale

```python
alibi.explainers.plot_ale(exp, features='all', targets='all', n_cols=3, sharey='all', constant=False, ax=None, line_kw=None, fig_kw=None)
```

Plot ALE curves on matplotlib axes.

### Parameters

- **exp** – An `Explanation` object produced by a call to the ALE `explain` method.
- **features** – A list of features for which to plot the ALE curves or `all` for all features. Can be a mix of integers denoting feature index or strings denoting entries in `exp.feature_names`. Defaults to `‘all’`.
- **targets** – A list of targets for which to plot the ALE curves or `all` for all targets. Can be a mix of integers denoting target index or strings denoting entries in `exp.target_names`. Defaults to `‘all’`.
- **n_cols** – Number of columns to organize the resulting plot into.
- **sharey** – A parameter specifying whether the y-axis of the ALE curves should be on the same scale for several features. Possible values are `all`, `row`, `None`.
- **constant** – A parameter specifying whether the constant zeroth order effects should be added to the ALE first order effects.
- **ax** – A `matplotlib` axes object or a numpy array of `matplotlib` axes to plot on.
- **line_kw** – Keyword arguments passed to the `plt.plot` function.
- **fig_kw** – Keyword arguments passed to the `fig.set` function.

### Returns

An array of `matplotlib` axes with the resulting ALE plots.

### class alibi.explainers.IntegratedGradients

```python
class alibi.explainers.IntegratedGradients(model, layer=None, method='gausslegendre', n_steps=50, internal_batch_size=100)
```

Bases: `alibi.api.interfaces.Explainer`
__init__ (model, layer=None, method='gausslegendre', n_steps=50, internal_batch_size=100)
An implementation of the integrated gradients method for Tensorflow and Keras models.
For details of the method see the original paper: https://arxiv.org/abs/1703.01365.

Parameters

• model (Union[Model, ForwardRef]) – Tensorflow or Keras model.
• layer (Union[Layer, ForwardRef, List[Layer], List[ForwardRef]]) – Layers with respect to which the gradients are calculated. It can be a single layer or a list of layers. If not provided, the gradients are calculated with respect to the input.
• n_steps (int) – Number of step in the path integral approximation from the baseline to the input instance.
• internal_batch_size (int) – Batch size for the internal batching.

Return type None

build_explanation (X, baselines, target, attributions)

Return type Explanation

explain (X, baselines=None, target=None)
Calculates the attributions for each input feature or element of layer and returns an Explanation object.

Parameters

• X (Union[ndarray, List[ndarray]]) – Instance for which integrated gradients attribution are computed.
• baselines (Union[int, float, ndarray, List[int], List[float], List[ndarray], None]) – Baselines (starting point of the path integral) for each instance. If the passed value is an np.ndarray must have the same shape as X. If not provided, all features values for the baselines are set to 0.
• target (Union[int, list, ndarray, None]) – Defines which element of the model output is considered to compute the gradients. It can be a list of integers or a numeric value. If a numeric value is passed, the gradients are calculated for the same element of the output for all data points. It must be provided if the model output dimension is higher than 1. For regression models whose output is a scalar, target should not be provided. For classification models target can be either the true classes or the classes predicted by the model.

Return type Explanation

Returns

• Explanation object including meta and data attributes with integrated gradients attributions
• for each feature.
Submodules

alibi.explainers.ale module

class alibi.explainers.ale.ALE(predictor, feature_names=None, target_names=None, check_feature_resolution=True, low_resolution_threshold=10, extrapolate_constant=True, extrapolate_constant_perc=10.0, extrapolate_constant_min=0.1)

Bases: alibi.api.interfaces.Explainer

__init__(predictor, feature_names=None, target_names=None, check_feature_resolution=True, low_resolution_threshold=10, extrapolate_constant=True, extrapolate_constant_perc=10.0, extrapolate_constant_min=0.1)

Accumulated Local Effects for tabular datasets. Current implementation supports first order feature effects of numerical features.

Parameters

- **predictor** *(Callable)* – A callable that takes in an NxF array as input and outputs an NxT array (N - number of data points, F - number of features, T - number of outputs/targets (e.g. 1 for single output regression, >=2 for classification)).

- **feature_names** *(Optional[List[str]])* – A list of feature names used for displaying results.

- **target_names** *(Optional[List[str]])* – A list of target/output names used for displaying results.

- **check_feature_resolution** *(bool)* – If true, the number of unique values is calculated for each feature and if it is less than low_resolution_threshold then the feature values are used for gridpoints instead of quantiles. This may increase the runtime of the algorithm for large datasets.

- **low_resolution_threshold** *(int)* – If a feature has at most this many unique values, these are used as the grid points instead of quantiles. This is to avoid situations when the quantile algorithm returns quantiles between discrete values which can result in jumps in the ALE plot obscuring the true effect. Only used if check_feature_resolution is True.

- **extrapolate_constant** *(bool)* – If a feature is constant, only one quantile exists where all the data points lie. In this case the ALE value at that poiny is zero, however this may be misleading if the feature does have an effect on the model. If this parameter is set to True, the ALE values are calculated on an interval surrounding the constant value. The interval length is controlled by the extrapolate_constant_perc and extrapolate_constant_min arguments.

- **extrapolate_constant_perc** *(float)* – Percentage by which to extrapolate a constant feature value to create an interval for ALE calculation. If q is the constant feature value, creates an interval \([q - q/extrapolate_constant_perc, q + q/extrapolate_constant_perc]\) for which ALE is calculated. Only relevant if extrapolate_constant is set to True.

- **extrapolate_constant_min** *(float)* – Controls the minimum extrapolation length for constant features. An interval constructed for constant features is guaranteed to be \(2\times\)extrapolate_constant_min wide centered on the feature value. This allows for capturing model behaviour around constant features which have small value so that extrapolate_constant_perc is not so helpful. Only relevant if extrapolate_constant is set to True.
**Return type** None

`build_explanation(ale_values, ale0, constant_value, feature_values, feature_deciles, feature_names)`

Helper method to build the Explanation object.

**Return type** Explanation

`explain(X, features=None, min_bin_points=4)`

Calculate the ALE curves for each feature with respect to the dataset X.

**Parameters**

- `X (ndarray)` – An NxF tabular dataset used to calculate the ALE curves. This is typically the training dataset or a representative sample.
- `features (Optional[List[int]])` – Features for which to calculate ALE.
- `min_bin_points (int)` – Minimum number of points each discretized interval should contain to ensure more precise ALE estimation.

**Return type** Explanation

**Returns** An Explanation object containing the data and the metadata of the calculated ALE curves.

`alibi.explainers.ale.adaptive_grid(values, min_bin_points=1)`

Find the optimal number of quantiles for the range of values so that each resulting bin contains at least `min_bin_points`. Uses bisection.

**Parameters**

- `values (ndarray)` – Array of feature values.
- `min_bin_points (int)` – Minimum number of points each discretized interval should contain to ensure more precise ALE estimation.

**Return type** Tuple

**Returns**

- `q` – Unique quantiles.
- `num_quantiles` – Number of non-unique quantiles the feature array was subdivided into.

**Notes**

This is a heuristic procedure since the bisection algorithm is applied to a function which is not monotonic. This will not necessarily find the maximum number of bins the interval can be subdivided into to satisfy the minimum number of points in each resulting bin.

`alibi.explainers.ale.ale_num(predictor, X, feature, min_bin_points=4, check_feature_resolution=True, low_resolution_threshold=10, extrapolate_constant=True, extrapolate_constant_perc=10.0, extrapolate_constant_min=0.1)`

Calculate the first order ALE curve for a numerical feature.

**Parameters**

- `predictor (Callable)` – Model prediction function.
- `X (ndarray)` – Dataset for which ALE curves are computed.
- `feature (int)` – Index of the numerical feature for which to calculate ALE.
• **min_bin_points** (*int*) – Minimum number of points each discretized interval should contain to ensure more precise ALE estimation.

• **check_feature_resolution** (*bool*) – Refer to ALE documentation.

• **low_resolution_threshold** (*int*) – Refer to ALE documentation.

• **extrapolate_constant** (*bool*) – Refer to ALE documentation.

• **extrapolate_constant_perc** (*float*) – Refer to ALE documentation.

• **extrapolate_constant_min** (*float*) – Refer to ALE documentation.

**Return type** `Tuple[ndarray, ...]`

**Returns**

- `q` – Array of quantiles of the input values.
- `ale` – ALE values for each feature at each of the points in `q`.
- `ale0` – The constant offset used to center the ALE curves.

```python
alibi.explainers.ale.bisect_fun(fun, target, lo, hi)
```

Bisection algorithm for function evaluation with integer support.

Assumes the function is non-decreasing on the interval `[lo, hi]`. Return an integer value `v` such that for all `x<v, fun(x)<target` and for all `x>=v fun(x)>=target`. This is equivalent to the library function `bisect.bisect_left` but for functions defined on integers.

**Parameters**

- **fun** (*Callable*) – A function defined on integers in the range `[lo, hi]` and returning floats.
- **target** (*float*) – Target value to be searched for.
- **lo** (*int*) – Lower bound of the domain.
- **hi** (*int*) – Upper bound of the domain.

**Return type** `int`

**Returns** `Integer index`.

```python
alibi.explainers.ale.get_quantiles(values, num_quantiles=11, interpolation='linear')
```

Calculate quantiles of values in an array.

**Parameters**

- **values** (*ndarray*) – Array of values.
- **num_quantiles** (*int*) – Number of quantiles to calculate.

**Return type** `ndarray`

**Returns** `Array of quantiles of the input values`.

```python
alibi.explainers.ale.minimum_satisfied(values, min_bin_points, n)
```

Calculates whether the partition into bins induced by `n` quantiles has the minimum number of points in each resulting bin.

**Parameters**

- **values** (*ndarray*) – Array of feature values.
- **min_bin_points** (*int*) – Minimum number of points each discretized interval needs to contain.
• n (int) – Number of quantiles.

**Return type** int

**Returns** Integer encoded boolean with 1 - each bin has at least min_bin_points and 0 otherwise.

alibi.explainers.ale.plot_ale(exp, features='all', targets='all', n_cols=3, sharey='all', constant=False, ax=None, line_kw=None, fig_kw=None)

Plot ALE curves on matplotlib axes.

**Parameters**

- **exp** – An Explanation object produced by a call to the ALE.explain method.
- **features** – A list of features for which to plot the ALE curves or all for all features. Can be a mix of integers denoting feature index or strings denoting entries in exp.feature_names. Defaults to ‘all’.
- **targets** – A list of targets for which to plot the ALE curves or all for all targets. Can be a mix of integers denoting target index or strings denoting entries in exp.target_names. Defaults to ‘all’.
- **n_cols** – Number of columns to organize the resulting plot into.
- **sharey** – A parameter specifying whether the y-axis of the ALE curves should be on the same scale for several features. Possible values are all, row, None.
- **constant** – A parameter specifying whether the constant zeroth order effects should be added to the ALE first order effects.
- **ax** – A matplotlib axes object or a numpy array of matplotlib axes to plot on.
- **line_kw** – Keyword arguments passed to the plt.plot function.
- **fig_kw** – Keyword arguments passed to the fig.set function.

**Returns** An array of matplotlib axes with the resulting ALE plots.

### alibi.explainers.anchor_base module

#### class alibi.explainers.anchor_base.AnchorBaseBeam(samplers, **kwargs)

**Bases:** object

**__init__** (samplers, **kwargs)

**Parameters** samplers (List[Callable]) – Objects that can be called with args (result, n_samples) tuple to draw samples.

**Return type** None

**anchor_beam** (delta=0.05, epsilon=0.1, desired_confidence=1.0, beam_size=1, epsilon_stop=0.05, min_samples_start=100, max_anchor_size=None, stop_on_first=False, batch_size=100, coverage_samples=10000, verbose=False, verbose_every=1, **kwargs)

Uses the KL-LUCB algorithm (Kaufmann and Kalyanakrishnan, 2013) together with additional sampling to search feature sets (anchors) that guarantee the prediction made by a classifier model. The search is greedy if beam_size=1. Otherwise, at each of the max_anchor_size steps, beam_size solutions are explored. By construction, solutions found have high precision (defined as the expected of number of times the classifier makes the same prediction when queried with the feature subset combined with arbitrary samples drawn from a noise distribution) The algorithm maximises the coverage of the solution found - the frequency of occurrence of records containing the feature subset in set of samples.

**Parameters**
• **delta** *(float)* – Used to compute beta.
• **epsilon** *(float)* – Precision bound tolerance for convergence.
• **desired_confidence** *(float)* – Desired level of precision (tau in paper).
• **beam_size** *(int)* – Beam width.
• **epsilon_stop** *(float)* – Confidence bound margin around desired precision.
• **min_samples_start** *(int)* – Min number of initial samples.
• **max_anchor_size** *(Optional[int])* – Max number of features in result.
• **stop_on_first** *(bool)* – Stop on first valid result found.
• **coverage_samples** *(int)* – Number of samples from which to build a coverage set.
• **batch_size** *(int)* – Number of samples used for an arm evaluation.
• **verbose** *(bool)* – Whether to print intermediate LUCB & anchor selection output.
• **verbose_every** *(int)* – Print intermediate output every verbose_every steps.

Return type: **dict**

Returns

• Explanation dictionary containing anchors with metadata like coverage and precision and examples.

**static** **compute_beta** *(n_features, t, delta)*

Parameters

• **n_features** *(int)* – Number of candidate anchors.
• **t** *(int)* – Iteration number.
• **delta** *(float)* –

Return type: **float**

Returns

Level used to update upper and lower precision bounds.

**static** **dlow_bernoulli** *(p, level, n_iter=17)*

Update lower precision bound for a candidate anchors dependent on the KL-divergence.

Parameters

• **p** *(ndarray)* – Precision of candidate anchors.
• **level** *(ndarray)* – beta / nb of samples for each result.
• **n_iter** *(int)* – Number of iterations during lower bound update.

Return type: **ndarray**

Returns

Updated lower precision bounds array.

**draw_samples** *(anchors, batch_size)*

Parameters

• **anchors** *(list)* – Anchors on which samples are conditioned.
• **batch_size** *(int)* – The number of samples drawn for each result.

Return type: **Tuple[tuple, tuple]**
alibi Documentation, Release 0.5.6

Returns

- A tuple of positive samples (for which prediction matches desired label)
- and a tuple of total number of samples drawn.

static dup_bernoulli (p, level, n_iter=17)
Update upper precision bound for a candidate anchors dependent on the KL-divergence.

Parameters

- p (ndarray) – Precision of candidate anchors.
- level (ndarray) – beta / nb of samples for each result.
- n_iter (int) – Number of iterations during lower bound update.

Return type ndarray

Returns Updated upper precision bounds array.

get_anchor_metadata (features, success, batch_size=100)
Given the features contained in a result, it retrieves metadata such as the precision and coverage of the result and partial anchors and examples where the result/partial anchors apply and yield the same prediction as on the instance to be explained (covered_true) or a different prediction (covered_false).

Parameters

- features (tuple) – Sorted indices of features in result.
- success – Indicates whether an anchor satisfying precision threshold was met or not.
- batch_size (int) – Number of samples among which positive and negative examples for partial anchors are selected if partial anchors have not already been explicitly sampled.

Return type dict

Returns

- Anchor dictionary with result features and additional metadata.
- param success:

get_init_stats (anchors, coverages=False)
Finds the number of samples already drawn for each result in anchors, their comparisons with the instance to be explained and, optionally, coverage.

Parameters

- anchors (list) – Candidate anchors.
- coverages – If True, the statistics returned contain the coverage of the specified anchors.

Return type dict

Returns

- Dictionary with lists containing nb of samples used and where sample predictions equal
- the desired label.

kllucb (anchors, init_stats, epsilon, delta, batch_size, top_n, verbose=False, verbose_every=1)
Implements the KL-LUCB algorithm (Kaufmann and Kalyanakrishnan, 2013).
Parameters

- **anchors** *(list)* – A list of anchors from which two critical anchors are selected (see Kaufmann and Kalyanakrishnan, 2013).
- **init_stats** *(dict)* – Dictionary with lists containing nb of samples used and where sample predictions equal the desired label.
- **epsilon** *(float)* – Precision bound tolerance for convergence.
- **delta** *(float)* – Used to compute beta.
- **batch_size** *(int)* – Number of samples.
- **top_n** *(int)* – Min of beam width size or number of candidate anchors.
- **verbose** *(bool)* – Whether to print intermediate output.
- **verbose_every** *(int)* – Whether to print intermediate output every verbose_every steps.

Return type: **ndarray**

Returns: Indices of best result options. Number of indices equals min of beam width or nb of candidate anchors.

**propose_anchors** *(previous_best)*

Parameters **previous_best** *(list)* – List with tuples of result candidates.

Return type: **list**

Returns: List with tuples of candidate anchors with additional metadata.

**select_critical_arms** *(means, ub, lb, n_samples, delta, top_n, t)*

Determines a set of two anchors by updating the upper bound for low empirical precision anchors and the lower bound for anchors with high empirical precision.

Parameters

- **means** *(ndarray)* – Empirical mean result precisions.
- **ub** *(ndarray)* – Upper bound on result precisions.
- **lb** *(ndarray)* – Lower bound on result precisions.
- **n_samples** *(ndarray)* – The number of samples drawn for each candidate result.
- **delta** *(float)* – Confidence budget, candidate anchors have close to optimal precisions with prob. 1 - delta.
- **top_n** *(int)* – Number of arms to be selected.
- **t** *(int)* – Iteration number.

Returns: Upper and lower precision bound indices.

**static to_sample** *(means, ubs, lbs, desired_confidence, epsilon_stop)*

Given an array of mean result precisions and their upper and lower bounds, determines for which anchors more samples need to be drawn in order to estimate the anchors precision with desired_confidence and error tolerance.

Parameters

- **means** *(ndarray)* – Mean precisions (each element represents a different result).
- **ubs** *(ndarray)* – Precisions’ upper bounds (each element represents a different result).
• **lbs** (*ndarray*) – Precisions’ lower bounds (each element represents a different result).

• **desired_confidence** (*float*) – Desired level of confidence for precision estimation.

• **epsilon_stop** (*float*) – Tolerance around desired precision.

**Returns**

Boolean array indicating whether more samples are to be drawn for that particular result.

**update_state**(covered_true, covered_false, labels, samples, anchor)

Updates the explainer state (see `__init__` for full state definition).

**Parameters**

• **covered_true** (*ndarray*) – Examples where the result applies and the prediction is the same as on the instance to be explained.

• **covered_false** (*ndarray*) – Examples where the result applies and the prediction is different to the instance to be explained.

• **samples** (*tuple*) – A tuple containing discretized data, coverage and the result sampled.

• **labels** (*ndarray*) – An array indicating whether the prediction on the sample matches the label of the instance to be explained.

• **anchor** (*tuple*) – The result to be updated.

**Return type** `Tuple[int, int]`

**Returns**

• A tuple containing the number of instances equals desired label of observation to be explained the total number of instances sampled, and the result that was sampled.

**class** `alibi.explainers.anchor_base.DistributedAnchorBaseBeam`(samplers, **kwargs)

**Bases:** `alibi.explainers.anchor_base.AnchorBaseBeam`

**draw_samples**(anchors, batch_size)

Distributes sampling requests among processes running sampling tasks.

**Parameters** superclass implementation. *(See)*

**Return type** `Tuple[ndarray, ndarray]`

**Returns**

Same outputs as superclass but of different types.

**class** `alibi.explainers.anchor_explanation.AnchorExplanation`(exp_type, exp_map)

**Bases:** `object`

**__init__**(exp_type, exp_map)

Class used to unpack the anchors and metadata from the explainer dictionary.

**Parameters**

• **exp_type** (*str*) – Type of explainer: tabular, text or image

• **exp_map** (*dict*) – Dictionary with the anchors and explainer metadata for an observation
Return type None

coverage (partial_index=None)

Parameters partial_index (Optional[int]) – Get the result coverage until a certain index. For example, if the result has precisions [0.1, 0.5, 0.95] and partial_index=1, this will return 0.5.

Return type float
Returns coverage – Anchor coverage

extamples (only_different_prediction=False, only_same_prediction=False, partial_index=None)

Parameters

• only_different_prediction (bool) – If True, will only return examples where the result makes a different prediction than the original model

• only_same_prediction (bool) – If True, will only return examples where the result makes the same prediction than the original model

• partial_index (Optional[int]) – Get the examples from the partial result until a certain index

Return type ndarray
Returns Examples covered by result

features (partial_index=None)

Parameters partial_index (Optional[int]) – Get the result until a certain index. For example, if the result uses segment_labels (1, 2, 3) and partial_index=1, this will return [1, 2].

Return type list
Returns segment_labels – Features used in the result conditions.

names (partial_index=None)

Parameters partial_index (Optional[int]) – Get the result until a certain index. For example, if the result is (A=1,B=2,C=2) and partial_index=1, this will return [“A=1”, “B=2”].

Return type list
Returns names – Names with the result conditions

precision (names)

Parameters partial_index (Optional[int]) – Get the result precision until a certain index. For example, if the result has precisions [0.1, 0.5, 0.95] and partial_index=1, this will return 0.5.

Return type float
Returns precision – Anchor precision
### alibi.explainers.anchor_image module

**class** alibi.explainers.anchor_image.AnchorImage(predictor, image_shape, segmentation_fn='slic', segmentation_kwargs=None, images_background=None, seed=None)

Bases: alibi.api.interfaces.Explainer

**__init__**(predictor, image_shape, segmentation_fn='slic', segmentation_kwargs=None, images_background=None, seed=None)

Initialize anchor image explainer.

**Parameters**

- *predictor*(Callable) – A callable that takes a tensor of N data points as inputs and returns N outputs.
- *image_shape*(tuple) – Shape of the image to be explained.
- *segmentation_fn*(Any) – Any of the built in segmentation function strings: ‘felzenszwalb’, ‘slic’ or ‘quickshift’ or a custom segmentation function (callable) which returns an image mask with labels for each superpixel. See [http://scikit-image.org/docs/dev/api/skimage.segmentation.html](http://scikit-image.org/docs/dev/api/skimage.segmentation.html) for more info.
- *segmentation_kwargs*(Optional[dict]) – Keyword arguments for the built in segmentation functions.
- *images_background*(Optional[ndarray]) – Images to overlay superpixels on.
- *seed*(Optional[int]) – If set, ensures different runs with the same input will yield same explanation.

**Return type** None

**build_explanation**(image, result, predicted_label, params)

Uses the metadata returned by the anchor search algorithm together with the instance to be explained to build an explanation object.

**Parameters**

- *image*(ndarray) – Instance to be explained.
- *result*(dict) – Dictionary containing the search anchor and metadata.
- *predicted_label*(int) – Label of the instance to be explained.
- *params*(dict) – Parameters passed to explain

**Return type** Explanation

**compare_labels**(samples)

Compute the agreement between a classifier prediction on an instance to be explained and the prediction on a set of samples which have a subset of perturbed superpixels.

**Parameters**

- *samples*(ndarray) – Samples whose labels are to be compared with the instance label.

**Return type** ndarray

**Returns** A boolean array indicating whether the prediction was the same as the instance label.
**explain**

```python
(explain, p_sample=0.5, threshold=0.95, delta=0.1, tau=0.15, batch_size=100, coverage_samples=10000, beam_size=1, stop_on_first=False, max_anchor_size=None, min_samples_start=100, n_covered_ex=10, binary_cache_size=10000, cache_margin=1000, verbose=False, verbose_every=1, **kwargs)
```

Explain instance and return anchor with metadata.

**Parameters**

- `image` *(ndarray)* – Image to be explained.
- `p_sample` *(float)* – Probability for a pixel to be represented by the average value of its superpixel.
- `threshold` *(float)* – Minimum precision threshold.
- `delta` *(float)* – Used to compute beta.
- `tau` *(float)* – Margin between lower confidence bound and minimum precision of upper bound.
- `batch_size` *(int)* – Batch size used for sampling.
- `coverage_samples` *(int)* – Number of samples used to estimate coverage from during result search.
- `beam_size` *(int)* – The number of anchors extended at each step of new anchors construction.
- `stop_on_first` *(bool)* – If True, the beam search algorithm will return the first anchor that has satisfies the probability constraint.
- `max_anchor_size` *(Optional[int]*) – Maximum number of features in result.
- `min_samples_start` *(int)* – Min number of initial samples.
- `n_covered_ex` *(int)* – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with desired label are stored).
- `binary_cache_size` *(int)* – The result search pre-allocates binary_cache_size batches for storing the binary arrays returned during sampling.
- `cache_margin` *(int)* – When only max(cache_margin, batch_size) positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.
- `verbose` *(bool)* – Display updates during the anchor search iterations.
- `verbose_every` *(int)* – Frequency of displayed iterations during anchor search process.

**Return type** *Explanation*

**Returns** *explanation* – *Explanation* object containing the anchor explaining the instance with additional metadata as attributes.

**generate_superpixels** *(image)*

Generates superpixels from (i.e., segments) an image.

**Parameters**

- `image` *(ndarray)* – A grayscale or RGB image.

**Return type** *ndarray*

**overlay_mask** *(image, segments, mask_features, scale=(0, 255))*

Overlay image with mask described by the mask features.

**Parameters**

- **image** *(ndarray)* – Image to be explained.
- **segments** *(ndarray)* – Superpixels
- **mask_features** *(list)* – List with superpixels present in mask.
- **scale** *(tuple)* – Pixel scale for masked image.

**Return type** **ndarray**

**Returns**  
*masked_image* – Image overlaid with mask.

**perturbation** *(anchor, num_samples)*

Perturbs an image by altering the values of selected superpixels. If a dataset of image backgrounds is provided to the explainer, then the superpixels are replaced with the equivalent superpixels from the background image. Otherwise, the superpixels are replaced by their average value.

**Parameters**

- **anchor** *(tuple)* – Contains the superpixels whose values are not going to be perturbed.
- **num_samples** *(int)* – Number of perturbed samples to be returned.

**Return type** **Tuple[ndarray, ndarray]**

**Returns**

- **segments_mask** – A [num_samples, M] binary mask, where M is the number of image superpixels segments. 1 indicates the values in that particular superpixels are not perturbed.

**sampler** *(anchor, num_samples, compute_labels=True)*

Sample images from a perturbation distribution by masking randomly chosen superpixels from the original image and replacing them with pixel values from superimposed images if background images are provided to the explainer. Otherwise, the superpixels from the original image are replaced with their average values.

**Parameters**

- **anchor** *(Tuple[int, tuple])* – int: order of anchor in the batch tuple: features (= superpixels) present in the proposed anchor
- **num_samples** *(int)* – Number of samples used
- **compute_labels** *(bool)* – If True, an array of comparisons between predictions on perturbed samples and instance to be explained is returned.

**Return type** **Union[List[Union[ndarray, float, int]], List[ndarray]]**

**Returns**

- If **compute_labels**=True, a list containing the following is returned –
  - **covered_true**: perturbed examples where the anchor applies and the model prediction on perturbed is the same as the instance prediction
  - **covered_false**: perturbed examples where the anchor applies and the model prediction on perturbed sample is NOT the same as the instance prediction
– labels: num_samples ints indicating whether the prediction on the perturbed sample matches (1) the label of the instance to be explained or not (0)
– data: Matrix with 1s and 0s indicating whether the values in a superpixel will remain unchanged (1) or will be perturbed (0), for each sample
– 1.0: indicates exact coverage is not computed for this algorithm
– anchor[0]: position of anchor in the batch request

• Otherwise, a list containing the data matrix only is returned.

**алиби**.explainers.anchor_tabular module

class alihi.explainers.anchor_tabular.AnchorTabular(predictor,   feature_names,   categorical_names=None,   seed=None)

Bases: alihi.api.interfaces.Explainer, alihi.api.interfaces.FitMixin

__init__(predictor, feature_names, categorical_names=None, seed=None)

Parameters

• predictor (Callable) – A callable that takes a tensor of N data points as inputs and returns N outputs.
• feature_names (list) – List with feature names.
• categorical_names (Optional[dict]) – Dictionary where keys are feature columns and values are the categories for the feature.
• seed (Optional[int]) – Used to set the random number generator for repeatability purposes.

Return type None

add_names_to_exp(explanation)

Add feature names to explanation dictionary.

Parameters explanation (dict) – Dict with anchors and additional metadata.

Return type None

build_explanation(X, result, predicted_label, params)

Preprocess search output and return an explanation object containing metadata.

Parameters

• X (ndarray) – Instance to be explained.
• result (dict) – Dictionary with explanation search output and metadata.
• predicted_label (int) – Label of the instance to be explained (inferred if not given).
• params (dict) – Parameters passed to explain

Return type Explanation

Returns

• Explanation object containing human readable explanation, metadata, and precision/coverage
• info as attributes.
**explain**(X, threshold=0.95, delta=0.1, tau=0.15, batch_size=100, coverage_samples=10000, beam_size=1, stop_on_first=False, max_anchor_size=None, min_samples_start=100, n_covered_ex=10, binary_cache_size=10000, cache_margin=1000, verbose=False, verbose_every=1, **kwargs)

Explain prediction made by classifier on instance X.

**Parameters**

- **X** *(ndarray)* – Instance to be explained.
- **threshold** *(float)* – Minimum precision threshold.
- **delta** *(float)* – Used to compute beta.
- **tau** *(float)* – Margin between lower confidence bound and minimum precision or upper bound.
- **batch_size** *(int)* – Batch size used for sampling.
- **coverage_samples** *(int)* – Number of samples used to estimate coverage from during result search.
- **beam_size** *(int)* – The number of anchors extended at each step of new anchors construction.
- **stop_on_first** *(bool)* – If True, the beam search algorithm will return the first anchor that has satisfies the probability constraint.
- **max_anchor_size** *(Optional[int])* – Maximum number of features in result.
- **min_samples_start** *(int)* – Min number of initial samples.
- **n_covered_ex** *(int)* – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with desired_label are stored).
- **binary_cache_size** *(int)* – The result search pre-allocates binary_cache_size batches for storing the binary arrays returned during sampling.
- **cache_margin** *(int)* – When only max(cache_margin, batch_size) positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.
- **verbose** *(bool)* – Display updates during the anchor search iterations.
- **verbose_every** *(int)* – Frequency of displayed iterations during anchor search process.

**Return type** *Explanation*

**Returns** *explanation* – *Explanation* object containing the result explaining the instance with additional metadata as attributes.

**fit**(train_data, disc_perc=(25, 50, 75), **kwargs)

Fit discretizer to train data to bin numerical features into ordered bins and compute statistics for numerical features. Create a mapping between the bin numbers of each discretised numerical feature and the row id in the training set where it occurs.

**Parameters**

- **train_data** *(ndarray)* – Representative sample from the training data.
- **disc_perc** *(Tuple[Union[int, float], ...])* – List with percentiles (int) used for discretization.

**Return type** *AnchorTabular*
class alibi.explainers.anchor_tabular.DistributedAnchorTabular(predictor, feature_names, categorical_names=None, seed=None)

Bases: alibi.explainers.anchor_tabular.AnchorTabular

explain(X, threshold=0.95, delta=0.1, tau=0.15, batch_size=100, coverage_samples=10000, beam_size=1, stop_on_first=False, max_anchor_size=None, min_samples_start=1, n_covered_ex=10, binary_cache_size=10000, cache_margin=1000, verbose=False, verbose_every=1, **kwargs)

Explains the prediction made by a classifier on instance X. Sampling is done in parallel over a number of cores specified in kwargs['ncpu'].

Parameters superclass implementation. (See)

Return type Explanation

Returns See superclass implementation.

fit(train_data, disc_perc=(25, 50, 75), **kwargs)

Creates a list of handles to parallel processes handles that are used for submitting sampling tasks.

Parameters superclass implementation. (See)

Return type AnchorTabular

class alibi.explainers.anchor_tabular.RemoteSampler(*args)

Bases: object

A wrapper that facilitates the use of TabularSampler for distributed sampling.

__call__(anchors_batch, num_samples, compute_labels=True)

Wrapper around TabularSampler.__call__. It allows sampling a batch of anchors in the same process, which can improve performance.

Parameters

• anchors_batch (Union[Tuple[int, tuple], List[Tuple[int, tuple]]]) – A list of result tuples. see TabularSampler.__call__ for details.

• num_samples (int) – See TabularSampler.__call__.

• compute_labels (bool) – See TabularSampler.__call__.

Return type List

build_lookups(X)

Wrapper around TabularSampler.build_lookups.

Parameters X – See TabularSampler.build_lookups.

Returns See TabularSampler.build_lookups.

set_instance_label(X)

Sets the remote sampler instance label.

Parameters X (ndarray) – The instance to be explained.

Return type int

Returns label – The label of the instance to be explained.

set_n_covered(n_covered)

Sets the remote sampler number of examples to save for inspection.
Parameters \( n_{\text{covered}} \) (int) – Number of examples where the result (and partial anchors) apply.

Return type None

class alibi.explainers.anchor_tabular.TabularSampler(predictor, disc_perc, numerical_features, categorical_features, feature_names, feature_values, n_covered_ex=10, seed=None)

Bases: object

A sampler that uses an underlying training set to draw records that have a subset of features with values specified in an instance to be explained, \( X \).

__call__ (anchor, num_samples, compute_labels=True)

Obtain perturbed records by drawing samples from training data that contain the categorical labels and discretized numerical features and replacing the remainder of the record with arbitrary values.

Parameters

• anchor (Tuple[int, tuple]) – The integer represents the order of the result in a request array. The tuple contains encoded feature indices.

• num_samples (int) – Number of samples used when sampling from training set.

• compute_labels – If True, an array of comparisons between predictions on perturbed samples and instance to be explained is returned.

Return type Union[List[Union[ndarray, float, int]], List[ndarray]]

Returns

• If compute_labels=True, a list containing the following is returned –
  – covered_true: perturbed examples where the anchor applies and the model prediction on perturbation is the same as the instance prediction
  – covered_false: perturbed examples where the anchor applies and the model prediction is NOT the same as the instance prediction
  – labels: num_samples ints indicating whether the prediction on the perturbed sample matches (1) the label of the instance to be explained or not (0)
  – data: Sampled data where ordinal features are binned (1 if in bin, 0 otherwise)
  – coverage: the coverage of the anchor
  – anchor[0]: position of anchor in the batch request

• Otherwise, a list containing the data matrix only is returned.

__init__ (predictor, disc_perc, numerical_features, categorical_features, feature_names, feature_values, n_covered_ex=10, seed=None)

Parameters

• predictor (Callable) – A callable that takes a tensor of N data points as inputs and returns N outputs.

• disc_perc (Tuple[Union[int, float, ...]]) – Percentiles used for numerical feat. discretisation.

• numerical_features (List[int]) – Numerical features column IDs.
• **categorical_features** ([List][int]) – Categorical features column IDs.
• **feature_names** ([list]) – Feature names.
• **feature_values** ([dict]) – Key: categorical feature column ID, value: values for the feature.
• **n_covered_ex** ([int]) – For each result, a number of samples where the prediction agrees/disagrees with the prediction on instance to be explained are stored.
• **seed** ([Optional][int]) – If set, fixes the random number sequence.

**Return type** None

**build_lookups** (**X**)  
An encoding of the feature IDs is created by assigning each bin of a discretized numerical variable and each categorical variable a unique index. For a dataset containing, e.g., a numerical variable with 5 bins and 3 categorical variables, indices 0 - 4 represent bins of the numerical variable whereas indices 5, 6, 7 represent the encoded indices of the categorical variables (but see note for caviats). The encoding is necessary so that the different ranges of the numerical variable can be sampled during result construction. Note that the encoded indices represent the predicates used during the anchor construction process (i.e., anchor is a collection of encoded indices).

Note: Each continuous variable has n_bins - 1 corresponding entries in ord_lookup.

**Parameters**  
**X** ([ndarray]) – instance to be explained

**Return type** [List][Dict]

**Returns**

*a list containing three dictionaries, whose keys are encoded feature IDs –*

• **cat_lookup**: maps categorical variables to their value in **X**
• **ord_lookup**: maps discretized numerical variables to the bins they can be sampled from given **X**
• **enc2feat_idx**: maps the encoded IDs to the original (training set) feature column IDs

**compare_labels** (**samples**)  
Compute the agreement between a classifier prediction on an instance to be explained and the prediction on a set of samples which have a subset of features fixed to specific values.

**Parameters**  
**samples** ([ndarray]) – Samples whose labels are to be compared with the instance label.

**Return type** [ndarray]

**Returns**  
An array of integers indicating whether the prediction was the same as the instance label.

**deferred_init** (**train_data, d_train_data**)  
Initialise the Tabular sampler object with data, discretizer, feature statistics and build an index from feature values and bins to database rows for each feature.

**Parameters**

• **train_data** ([Union][ndarray, Any]) – Data from which samples are drawn. Can be a numpy array or a ray future.
• **d_train_data** ([Union][array, Any]) – Discretized version for training data. Can be a numpy array or a ray future.

**Return type** Any
Returns: An initialised sampler.

get_features_index(\texttt{anchor})

Given an anchor, this function finds the row indices in the training set where the feature has the same value as the feature in the instance to be explained (for ordinal variables, the row indices are those of rows which contain records with feature values in the same bin). The algorithm uses both the feature \textit{encoded} ids in anchor and the feature ids in the input data set. The two are mapped by \texttt{self.enc2feat_idx}.

Parameters:
\textbf{anchor (tuple)} – The anchor for which the training set row indices are to be retrieved. The ints represent encoded feature ids.

Return type: \texttt{Tuple[Dict[int, Set[int]], Dict[int, Any], List[Tuple[int, str, Union[Any, int]]]]}

Returns:
- \textit{allowed_bins} – Maps original feature ids to the bins that the feature should be sampled from given the input anchor.
- \textit{allowed_rows} – Maps original feature ids to the training set rows where these features have the same value as the anchor.
- \textit{unk_feat_values} – When a categorical variable with the specified value/discretized variable in the specified bin is not found in the training set, a tuple is added to \textit{unk_feat_values} to indicate the original feature id, its type (‘c’=categorical, o=’discretized continuous’) and the value/bin it should be sampled from.

handle_unk_features(\texttt{allowed_bins, num_samples, samples, unk_feature_values})

Replaces unknown feature values with defaults. For categorical variables, the replacement value is the same as the value of the unknown feature. For continuous variables, a value is sampled uniformly at random from the feature range.

Parameters:
- \textit{allowed_bins (Dict[int, Set[int]])} – See get_feature_index method.
- \textit{num_samples (int)} – Number of replacement values.
- \textit{samples (ndarray)} – Contains the samples whose values are to be replaced.
- \textit{unk_feature_values (List[Tuple[int, str, Union[Any, int]]])} – List of tuples where: [0] is original feature id, [1] feature type, [2] if var is categorical, replacement value, otherwise None

Return type: None

perturbation(\texttt{anchor, num_samples})

Implements functionality described in \texttt{__call__}.

Parameters:
- \textit{anchor (tuple)} – Each int is an encoded feature id.
- \textit{num_samples (int)} – Number of samples.

Return type: \texttt{Tuple[ndarray, ndarray, float]}

Returns:
- \textit{samples} – Sampled data from training set.
- \textit{d_samples} – Like samples, but continuous data is converted to ordinal discrete data (binned).
- \textit{coverage} – The coverage of the result in the training data.
replace_features(samples, allowed_rows, uniq_feat_ids, partial_anchor_rows, nb_partial_anchors, num_samples)
The method creates perturbed samples by first replacing all partial anchors with partial anchors drawn from the training set. Then remainder of the features are then replaced with random values drawn from the same bin for discretized continuous features and same value for categorical features.

Parameters

- samples (ndarray) – Randomly drawn samples, where the anchor does not apply.
- allowed_rows (Dict[int, Any]) – Maps feature ids to the rows indices in training set where the feature has same value as instance (cat.) or is in the same bin.
- uniq_feat_ids (List[int]) – Multiple encoded features in the anchor can map to the same original feature id. Unique features in the anchor. This is the list of unique original features id in the anchor.
- partial_anchor_rows (List[ndarray]) – The rows in the training set where each partial anchor applies. Last entry is an array of row indices where the entire anchor applies.
- nb_partial_anchors (ndarray) – The number of training records which contain each partial anchor.
- num_samples (int) – Number of perturbed samples to be returned.

Return type None

set_instance_label(X)
Sets the sampler label. Necessary for setting the remote sampling process state during explain call.

Parameters X (ndarray) – Instance to be explained.

Return type None

set_n_covered(n_covered)
Set the number of examples to be saved for each result and partial result during search process. The same number of examples is saved in the case where the predictions on perturbed samples and original instance agree or disagree.

Parameters n_covered (int) – Number of examples to be saved.

Return type None

alibi.explainers.anchor_text module

class alibi.explainers.anchor_text.AnchorText(nlp, predictor, seed=None)
Bases: alibi.api.interfaces.Explainer

UNK = 'UNK'

__init__(nlp, predictor, seed=None)
Initialize anchor text explainer.

Parameters

- nlp (spacy.language.Language) – spaCy object.
- predictor (Callable) – A callable that takes a tensor of N data points as inputs and returns N outputs.
- seed (int) – If set, ensures identical random streams.

Return type None
**build_explanation** *(text, result, predicted_label, params)*

Uses the metadata returned by the anchor search algorithm together with the instance to be explained to build an explanation object.

**Parameters**

- **text** *(str)* – Instance to be explained.
- **result** *(dict)* – Dictionary containing the search result and metadata.
- **predicted_label** *(int)* – Label of the instance to be explained. Inferred if not received.
- **params** *(dict)* – Parameters passed to explain

**Return type** *Explanation*

**compare_labels** *(samples)*

Compute the agreement between a classifier prediction on an instance to be explained and the prediction on a set of samples which have a subset of features fixed to a given value (aka compute the precision of anchors).

**Parameters**

- **samples** *(ndarray)* – Samples whose labels are to be compared with the instance label.

**Return type** *ndarray*

**Returns** A boolean array indicating whether the prediction was the same as the instance label.

**explain** *(text, use_unk=True, use_similarity_proba=False, sample_proba=0.5, top_n=100, temperature=1.0, threshold=0.95, delta=0.1, tau=0.15, batch_size=100, coverage_samples=10000, beam_size=1, stop_on_first=True, max_anchor_size=None, min_samples_start=100, n_covered_ex=10, binary_cache_size=10000, cache_margin=1000, verbose=False, verbose_every=1, **kwargs)*

Explain instance and return anchor with metadata.

**Parameters**

- **text** *(str)* – Text instance to be explained.
- **use_unk** *(bool)* – If True, perturbation distribution will replace words randomly with UNKs. If False, words will be replaced by similar words using word embeddings.
- **use_similarity_proba** *(bool)* – Sample according to a similarity score with the corpus embeddings use_unk needs to be False in order for this to be used.
- **sample_proba** *(float)* – Sample probability if use_similarity_proba is False.
- **top_n** *(int)* – Number of similar words to sample for perturbations, only used if use_unk=False.
- **temperature** *(float)* – Sample weight hyperparameter if use_similarity_proba equals True.
- **threshold** *(float)* – Minimum precision threshold.
- **delta** *(float)* – Used to compute beta.
- **tau** *(float)* – Margin between lower confidence bound and minimum precision or upper bound.
- **batch_size** *(int)* – Batch size used for sampling.
- **coverage_samples** *(int)* – Number of samples used to estimate coverage from during anchor search.
• **beam_size** (**int**) – Number of options kept after each stage of anchor building.

• **stop_on_first** (**bool**) – If True, the beam search algorithm will return the first anchor that has satisfies the probability constraint.

• **max_anchor_size** (**Optional[int]**) – Maximum number of features to include in an anchor.

• **min_samples_start** (**int**) – Number of samples used for anchor search initialisation.

• **n_covered_ex** (**int**) – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with predicted label are stored).

• **binary_cache_size** (**int**) – The anchor search pre-allocates binary_cache_size batches for storing the boolean arrays returned during sampling.

• **cache_margin** (**int**) – When only max(cache_margin, batch_size) positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.

• **kwargs** (**Any**) – Other keyword arguments passed to the anchor beam search and the text sampling and perturbation functions.

• **verbose** (**bool**) – Display updates during the anchor search iterations.

• **verbose_every** (**int**) – Frequency of displayed iterations during anchor search process.

**Return type** *Explanation*

**Returns** *explanation* – *Explanation* object containing the anchor explaining the instance with additional metadata as attributes.

**find_similar_words** ()

This function queries a spaCy nlp model to find n similar words with the same part of speech for each word in the instance to be explained. For each word the search procedure returns a dictionary containing an np.array of words (‘words’) and an np.array of word similarities (‘similarities’).

**Return type** *None*

**perturb_sentence** (**present**, **n**, **sample_proba=0.5**, **forbidden=frozenset({})**, **forbidden_tags=frozenset({'PRP$'})**, **forbidden_words=frozenset({'be'})**, **temperature=1.0**, **pos=frozenset({'ADJ', 'ADP', 'ADV', 'DET', 'NOUN', 'VERB'})**, **use_similarity_proba=True**) Perturb the text instance to be explained.

**Parameters**

• **present** (**tuple**) – Word index in the text for the words in the proposed anchor.

• **n** (**int**) – Number of samples used when sampling from the corpus.

• **sample_proba** (**float**) – Sample probability for a word if use_similarity_proba is False.

• **forbidden** (**frozenset**) – Forbidden lemmas.

• **forbidden_tags** (**frozenset**) – Forbidden POS tags.

• **forbidden_words** (**frozenset**) – Forbidden words.

• **pos** (**frozenset**) – POS that can be changed during perturbation.
• **use_similarity_proba** (bool) – Bool whether to sample according to a similarity score with the corpus embeddings.

• **temperature** (float) – Sample weight hyperparameter if use_similarity_proba equals True.

[Return type] Tuple [ndarray, ndarray]

[Returns]

• **raw_data** – Array of perturbed text instances.

• **data** – Matrix with 1s and 0s indicating whether a word in the text has not been perturbed for each sample.

**sampler** *(anchor, num_samples, compute_labels=True)*

Generate perturbed samples while maintaining features in positions specified in anchor unchanged.

[Parameters]

• **anchor** (Tuple [int, tuple]) – int: the position of the anchor in the input batch tuple: the anchor itself, a list of words to be kept unchanged

• **num_samples** (int) – Number of generated perturbed samples.

• **compute_labels** (bool) – If True, an array of comparisons between predictions on perturbed samples and instance to be explained is returned.

[Return type] Union [List [Union [ndarray, float, int]], List [ndarray]]

[Returns]

• If compute_labels=True, a list containing the following is returned –
  – **covered_true**: perturbed examples where the anchor applies and the model prediction on perturbation is the same as the instance prediction
  – **covered_false**: perturbed examples where the anchor applies and the model prediction is NOT the same as the instance prediction
  – **labels**: num_samples ints indicating whether the prediction on the perturbed sample matches (1) the label of the instance to be explained or not (0)
  – **data**: Matrix with 1s and 0s indicating whether a word in the text has been perturbed for each sample
    – 1.0: indicates exact coverage is not computed for this algorithm
    – anchor[0]: position of anchor in the batch request
  – Otherwise, a list containing the data matrix only is returned.

**set_data_type** *(use_unk)*

Working with numpy arrays of strings requires setting the data type to avoid truncating examples. This function estimates the longest sentence expected during the sampling process, which is used to set the number of characters for the samples and examples arrays. This depends on the perturbation method used for sampling.

[Parameters] use_unk (bool) – See explain method.

[Return type] None

**set_sampler_perturbation** *(use_unk, perturb_opts, top_n)*

Initialises the explainer by setting the perturbation function and parameters necessary to sample according to the perturbation method.
Parameters

- **use_unk** (bool) – see explain method
- **perturb_opts** (dict) – A dict with keys:
  - 'top_n': the max number of alternatives to sample from for replacement
  - 'use_similarity_proba': if True the probability of selecting a replacement word is proportional to the similarity between the word and the word to be replaced
  - 'sample_proba': given a feature and n sentences, this parameter is the mean of a Bernoulli distribution used to decide how many sentences will have that feature perturbed
  - 'temperature': a temperature used to calibrate the softmax distribution over the sampling weights.
- **top_n** (int) – Number of similar words to sample for perturbations, only used if use_unk=False.

Return type None

**set_words_and_pos** (text)
Process the sentence to be explained into spaCy token objects, a list of words, punctuation marks and a list of positions in input sentence.

Parameters **text** (str) – The instance to be explained.

Return type None

class alibi.explainers.anchor_text.Neighbors(nlp_obj, n_similar=500, w_prob=-15.0)
Bases: object

__init__ (nlp_obj, n_similar=500, w_prob=-15.0)
Initialize class identifying neighbouring words from the embedding for a given word.

Parameters

- **nlp_obj** (spacy.language.Language) – spaCy model
- **n_similar** (int) – Number of similar words to return.
- **w_prob** (float) – Smoothed log probability estimate of token’s type.

Return type None

**neighbors** (word, tag, top_n)
Find similar words for a certain word in the vocabulary.

Parameters

- **word** (str) – Word for which we need to find similar words.
- **tag** (str) – Part of speech tag for the words.
- **top_n** (int) – Return only top_n neighbors.

Return type **dict**

Returns

- A dict with two fields. The ‘words’ field contains a numpy array of the top_n most similar words, whereas the fields similarity is a numpy array with corresponding word similarities.
alibi.explainers.cem module

```python
class alibi.explainers.cem.CEM(predict, mode, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0, 10000000000.0), gamma=0.0, ae_model=None, learning_rate_init=0.01, max_iterations=1000, c_init=10.0, c_steps=10, eps=(0.001, 0.001), clip=(-100.0, 100.0), update_num_grad=1, no_info_val=None, write_dir=None, sess=None)
```

Bases: `alibi.api.interfaces.Explainer`, `alibi.api.interfaces.FitMixin`

```python
__init__(predict, mode, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0, 10000000000.0), gamma=0.0, ae_model=None, learning_rate_init=0.01, max_iterations=1000, c_init=10.0, c_steps=10, eps=(0.001, 0.001), clip=(-100.0, 100.0), update_num_grad=1, no_info_val=None, write_dir=None, sess=None)
```


**Parameters**

- `predict` (Union[Callable, Model, ForwardRef]) – Keras or TensorFlow model or any other model’s prediction function returning class probabilities
- `mode` (str) – Find pertinent negatives (‘PN’) or pertinent positives (‘PP’)
- `shape` (tuple) – Shape of input data starting with batch size
- `kappa` (float) – Confidence parameter for the attack loss term
- `beta` (float) – Regularization constant for L1 loss term
- `feature_range` (tuple) – Tuple with min and max ranges to allow for perturbed instances. Min and max ranges can be floats or numpy arrays with dimension (1x nb of features) for feature-wise ranges
- `gamma` (float) – Regularization constant for optional auto-encoder loss term
- `ae_model` (Union[Model, ForwardRef]) – Optional auto-encoder model used for loss regularization
- `learning_rate_init` (float) – Initial learning rate of optimizer
- `max_iterations` (int) – Maximum number of iterations for finding a PN or PP
- `c_init` (float) – Initial value to scale the attack loss term
- `c_steps` (int) – Number of iterations to adjust the constant scaling the attack loss term
- `eps` (tuple) – If numerical gradients are used to compute dL/dx = (dL/dp) * (dp/dx), then eps[0] is used to calculate dL/dp and eps[1] is used for dp/dx. eps[0] and eps[1] can be a combination of float values and numpy arrays. For eps[0], the array dimension should be (1x nb of prediction categories) and for eps[1] it should be (1x nb of features)
- `clip` (tuple) – Tuple with min and max clip ranges for both the numerical gradients and the gradients obtained from the TensorFlow graph
- `update_num_grad` (int) – If numerical gradients are used, they will be updated every update_num_grad iterations
- `no_info_val` (Union[float, ndarray]) – Global or feature-wise value considered as containing no information
- `write_dir` (str) – Directory to write tensorboard files to
**sess** *(Session)* – Optional Tensorflow session that will be used if passed instead of creating or inferring one internally

**Return type** None

**attack** *(X, Y, verbose=False)*
Find pertinent negative or pertinent positive for instance X using a fast iterative shrinkage-thresholding algorithm (FISTA).

**Parameters**

- **X** *(ndarray)* – Instance to attack
- **Y** *(ndarray)* – Labels for X
- **verbose** *(bool)* – Print intermediate results of optimization if True

**Return type** Tuple[ndarray, Tuple[ndarray, ndarray]]

**Returns** Overall best attack and gradients for that attack.

**explain** *(X, Y=None, verbose=False)*
Explain instance and return PP or PN with metadata.

**Parameters**

- **X** *(ndarray)* – Instances to attack
- **Y** *(Optional[ndarray])* – Labels for X
- **verbose** *(bool)* – Print intermediate results of optimization if True

**Return type** Explanation

**Returns** explanation – Explanation object containing the PP or PN with additional metadata as attributes.

**fit** *(train_data, no_info_type='median')*
Get ‘no information’ values from the training data.

**Parameters**

- **train_data** *(ndarray)* – Representative sample from the training data
- **no_info_type** *(str)* – Median or mean value by feature supported

**Return type** CEM

**get_gradients** *(X, Y)*
Compute numerical gradients of the attack loss term: \( \frac{dL}{dx} = (\frac{dL}{dP}) \cdot (\frac{dP}{dx}) \) with \( L = \text{loss\_attack\_s}; P = \text{predict}; x = \text{adv\_s} \)

**Parameters**

- **X** *(ndarray)* – Instance around which gradient is evaluated
- **Y** *(ndarray)* – One-hot representation of instance labels

**Return type** ndarray

**Returns** Array with gradients.

**loss_fn** *(pred\_proba, Y)*
Compute the attack loss.

**Parameters**

- **pred\_proba** *(ndarray)* – Prediction probabilities of an instance
• \( Y \) (ndarray) – One-hot representation of instance labels

**Return type** ndarray

**Returns** Loss of the attack.

**perturb** \((X, \text{eps}, \text{proba}=\text{False})\)
Apply perturbation to instance or prediction probabilities. Used for numerical calculation of gradients.

**Parameters**

• \( X \) (ndarray) – Array to be perturbed

• \( \text{eps} \) (Union[float, ndarray]) – Size of perturbation

• \( \text{proba} \) (bool) – If True, the net effect of the perturbation needs to be 0 to keep the sum of the probabilities equal to 1

**Return type** Tuple[ndarray, ndarray]

**Returns** Instances where a positive and negative perturbation is applied.

### alibi.explainers.cfproto module

**class** alibi.explainers.cfproto.CounterFactualProto\((\text{predict, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0, 10000000000.0), gamma=0.0, ae_model=None, enc_model=None, theta=0.0, cat_vars=None, ohe=False, use_kdtree=False, learning_rate_init=0.01, max_iterations=1000, c_init=10.0, c_steps=10, eps=(0.001, 0.001), clip=(-1000.0, 1000.0), update_num_grad=1, write_dir=None, sess=None})\)

**Bases:** alibi.api.interfaces.Explainer, alibi.api.interfaces.FitMixin

**__init__**\((\text{predict, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0, 10000000000.0), gamma=0.0, ae_model=None, enc_model=None, theta=0.0, cat_vars=None, ohe=False, use_kdtree=False, learning_rate_init=0.01, max_iterations=1000, c_init=10.0, c_steps=10, eps=(0.001, 0.001), clip=(-1000.0, 1000.0), update_num_grad=1, write_dir=None, sess=None})\)

Initialize prototypical counterfactual method.

**Parameters**

• \( \text{predict} \) (Union[Callable, Model, ForwardRef]) – Keras or TensorFlow model or any other model’s prediction function returning class probabilities

• \( \text{shape} \) (tuple) – Shape of input data starting with batch size

• \( \text{kappa} \) (float) – Confidence parameter for the attack loss term

• \( \text{beta} \) (float) – Regularization constant for L1 loss term

• \( \text{feature_range} \) (tuple) – Tuple with min and max ranges to allow for perturbed instances. Min and max ranges can be floats or numpy arrays with dimension (1x nb of features) for feature-wise ranges
• **gamma** (float) – Regularization constant for optional auto-encoder loss term

• **ae_model** (Union[Model, ForwardRef]) – Optional auto-encoder model used for loss regularization

• **enc_model** (Union[Model, ForwardRef]) – Optional encoder model used to guide instance perturbations towards a class prototype

• **theta** (float) – Constant for the prototype search loss term

• **cat_vars** (dict) – Dict with as keys the categorical columns and as values the number of categories per categorical variable.

• **ohe** (bool) – Whether the categorical variables are one-hot encoded (OHE) or not. If not OHE, they are assumed to have ordinal encodings.

• **use_kdtree** (bool) – Whether to use k-d trees for the prototype loss term if no encoder is available

• **learning_rate_init** (float) – Initial learning rate of optimizer

• **max_iterations** (int) – Maximum number of iterations for finding a counterfactual

• **c_init** (float) – Initial value to scale the attack loss term

• **c_steps** (int) – Number of iterations to adjust the constant scaling the attack loss term

• **eps** (tuple) – If numerical gradients are used to compute \( \frac{dL}{dx} = \frac{dL}{dp} \ast \frac{dp}{dx} \), then \( \text{eps}[0] \) is used to calculate \( \frac{dL}{dp} \) and \( \text{eps}[1] \) is used for \( \frac{dp}{dx} \). \( \text{eps}[0] \) and \( \text{eps}[1] \) can be a combination of float values and numpy arrays. For \( \text{eps}[0] \), the array dimension should be \( (1 \times \text{nb of prediction categories}) \) and for \( \text{eps}[1] \) it should be \( (1 \times \text{nb of features}) \)

• **clip** (tuple) – Tuple with min and max clip ranges for both the numerical gradients and the gradients obtained from the TensorFlow graph

• **update_num_grad** (int) – If numerical gradients are used, they will be updated every **update_num_grad** iterations

• **write_dir** (str) – Directory to write tensorboard files to

• **sess** (Session) – Optional Tensorflow session that will be used if passed instead of creating or inferring one internally

**Return type** None

**attack** (X, Y, target_class=None, k=None, k_type='mean', threshold=0.0, verbose=False, print_every=100, log_every=100)

Find a counterfactual (CF) for instance X using a fast iterative shrinkage-thresholding algorithm (FISTA).

**Parameters**

• **X** (ndarray) – Instance to attack

• **Y** (ndarray) – Labels for X as one-hot-encoding

• **target_class** (Optional[list]) – List with target classes used to find closest prototype. If None, the nearest prototype except for the predict class on the instance is used.

• **k** (Optional[int]) – Number of nearest instances used to define the prototype for a class. Defaults to using all instances belonging to the class if an encoder is used and to 1 for k-d trees.
• **k_type** (str) – Use either the average encoding of the k nearest instances in a class (k_type='mean') or the k-nearest encoding in the class (k_type='point') to define the prototype of that class. Only relevant if an encoder is used to define the prototypes.

• **threshold** (float) – Threshold level for the ratio between the distance of the counterfactual to the prototype of the predicted class for the original instance over the distance to the prototype of the predicted class for the counterfactual. If the trust score is below the threshold, the proposed counterfactual does not meet the requirements.

• **verbose** (bool) – Print intermediate results of optimization if True

• **print_every** (int) – Print frequency if verbose is True

• **log_every** (int) – Tensorboard log frequency if write directory is specified

**Return type** Tuple[ndarray, Tuple[ndarray, ndarray]]

**Returns** Overall best attack and gradients for that attack.

**explain** *(X, Y=None, target_class=None, k=None, k_type='mean', threshold=0.0, verbose=False, print_every=100, log_every=100)*

Explain instance and return counterfactual with metadata.

**Parameters**

• **X** (ndarray) – Instances to attack

• **Y** (Optional[ndarray]) – Labels for X as one-hot-encoding

• **target_class** (Optional[list]) – List with target classes used to find closest prototype. If None, the nearest prototype except for the predict class on the instance is used.

• **k** (Optional[int]) – Number of nearest instances used to define the prototype for a class. Defaults to using all instances belonging to the class if an encoder is used and to 1 for k-d trees.

• **k_type** (str) – Use either the average encoding of the k nearest instances in a class (k_type='mean') or the k-nearest encoding in the class (k_type='point') to define the prototype of that class. Only relevant if an encoder is used to define the prototypes.

• **threshold** (float) – Threshold level for the ratio between the distance of the counterfactual to the prototype of the predicted class for the original instance over the distance to the prototype of the predicted class for the counterfactual. If the trust score is below the threshold, the proposed counterfactual does not meet the requirements.

• **verbose** (bool) – Print intermediate results of optimization if True

• **print_every** (int) – Print frequency if verbose is True

• **log_every** (int) – Tensorboard log frequency if write directory is specified

**Return type** Explanation

**Returns** explanation – Explanation object containing the counterfactual with additional metadata as attributes.

**fit** *(train_data, trustscore_kwargs=None, d_type='abdm', w=None, disc_perc=(25, 50, 75), standardize_cat_vars=False, smooth=1.0, center=True, update_feature_range=True)*

Get prototypes for each class using the encoder or k-d trees. The prototypes are used for the encoder loss term or to calculate the optional trust scores.

**Parameters**

• **train_data** (ndarray) – Representative sample from the training data.
- **trustscore_kwargs (Optional[dict])** – Optional arguments to initialize the trust scores method.

- **d_type (str)** – Pairwise distance metric used for categorical variables. Currently, ‘abdm’, ‘mvdm’ and ‘abdm-mvdm’ are supported. ‘abdm’ infers context from the other variables while ‘mvdm’ uses the model predictions. ‘abdm-mvdm’ is a weighted combination of the two metrics.

- **w (Optional[float])** – Weight on ‘abdm’ (between 0. and 1.) distance if d_type equals ‘abdm-mvdm’.

- **disc_perc (Sequence[Union[int, float]])** – List with percentiles used in binning of numerical features used for the ‘abdm’ and ‘abdm-mvdm’ pairwise distance measures.

- **standardize_cat_vars (bool)** – Standardize numerical values of categorical variables if True.

- **smooth (float)** – Smoothing exponent between 0 and 1 for the distances. Lower values of 1 will smooth the difference in distance metric between different features.

- **center (bool)** – Whether to center the scaled distance measures. If False, the min distance for each feature except for the feature with the highest raw max distance will be the lower bound of the feature range, but the upper bound will be below the max feature range.

- **update_feature_range (bool)** – Update feature range with scaled values.

Return type **CounterFactualProto**

**get_gradients (X, Y, grads_shape, cat_vars_ord=) –**

Compute numerical gradients of the attack loss term: \( \frac{dL}{dx} = (\frac{dL}{dP}) \cdot (\frac{dP}{dx}) \) with \( L = \text{loss}_{-\text{attack}}; \ P = \text{predict}; \ x = \text{adv}_{-}\text{s} \)

Parameters

- **X (ndarray)** – Instance around which gradient is evaluated

- **Y (ndarray)** – One-hot representation of instance labels

- **grads_shape (tuple)** – Shape of gradients.

- **cat_vars_ord (Optional[dict])** – Dict with as keys the categorical columns and as values the number of categories per categorical variable.

Return type **ndarray**

Returns **Array with gradients.**

**loss_fn (pred_proba, Y)**

Compute the attack loss.

Parameters

- **pred_proba (ndarray)** – Prediction probabilities of an instance

- **Y (ndarray)** – One-hot representation of instance labels

Return type **ndarray**

Returns **Loss of the attack.**

**score (X, adv_class, orig_class, eps=1e-10)**

Parameters
• `X (ndarray)` – Instance to encode and calculate distance metrics for
• `adv_class (int)` – Predicted class on the perturbed instance
• `orig_class (int)` – Predicted class on the original instance
• `eps (float)` – Small number to avoid dividing by 0

**Return type** float

**Returns**

• Ratio between the distance to the prototype of the predicted class for the original instance and
• the prototype of the predicted class for the perturbed instance.

**alibi.explainers.counterfactual module**

```python
class alibi.explainers.counterfactual.CounterFactual(
predict_fn, shape, distance_fn='l1', target_proba=1.0, target_class='other',
max_iter=1000, early_stop=50, lam_init=0.1, max_lam_steps=10, tol=0.05,
learning_rate_init=0.1, feature_range=(-10000000000.0, 10000000000.0), eps=0.01,
init='identity', decay=True, write_dir=None, debug=False, sess=None)
```

**Bases:** `alibi.api.interfaces.Explainer`

**__init__** (predict_fn, shape, distance_fn='l1', target_proba=1.0, target_class='other',
max_iter=1000, early_stop=50, lam_init=0.1, max_lam_steps=10, tol=0.05, learning_rate_init=0.1,
feature_range=(-10000000000.0, 10000000000.0), eps=0.01,
init='identity', decay=True, write_dir=None, debug=False, sess=None)

Initialize counterfactual explanation method based on Wachter et al. (2017)

**Parameters**

• `predict_fn` (Union[Callable, Model, ForwardRef]) – Keras or TensorFlow model or any other model’s prediction function returning class probabilities
• `shape` (Tuple[int, ...]) – Shape of input data starting with batch size
• `distance_fn` (str) – Distance function to use in the loss term
• `target_proba` (float) – Target probability for the counterfactual to reach
• `target_class` (Union[str, int]) – Target class for the counterfactual to reach, one of ‘other’, ‘same’ or an integer denoting desired class membership for the counterfactual instance
• `max_iter` (int) – Maximum number of iterations to run the gradient descent for (inner loop)
• `early_stop` (int) – Number of steps after which to terminate gradient descent if all or none of found instances are solutions
• **lam_init** *(float)* – Initial regularization constant for the prediction part of the Wachter loss
• **max_lam_steps** *(int)* – Maximum number of times to adjust the regularization constant (outer loop) before terminating the search
• **tol** *(float)* – Tolerance for the counterfactual target probability
• **learning_rate_init** – Initial learning rate for each outer loop of lambda
• **feature_range** *(Union[Tuple, str])* – Tuple with min and max ranges to allow for perturbed instances. Min and max ranges can be floats or numpy arrays with dimension (1 x nb of features) for feature-wise ranges
• **eps** *(Union[float, ndarray])* – Gradient step sizes used in calculating numerical gradients, defaults to a single value for all features, but can be passed an array for feature-wise step sizes
• **init** *(str)* – Initialization method for the search of counterfactuals, currently must be ‘identity’
• **decay** *(bool)* – Flag to decay learning rate to zero for each outer loop over lambda
• **write_dir** *(str)* – Directory to write Tensorboard files to
• **debug** *(bool)* – Flag to write Tensorboard summaries for debugging
• **sess** *(Session)* – Optional Tensorflow session that will be used if passed instead of creating or inferring one internally

**Return type** None

**explain** *(X)*

Explain an instance and return the counterfactual with metadata.

**Parameters**

- **X** *(ndarray)* – Instance to be explained

**Return type** Explanation

**Returns** Explanation object containing the counterfactual with additional metadata as attributes.

**fit** *(X, y)*

Fit method - currently unused as the counterfactual search is fully unsupervised.

**Return type** CounterFactual

### alibi.explainers.integrated_gradients module

**class** alibi.explainers.integrated_gradients.IntegratedGradients *(model, layer=None, method='gausslegendre', n_steps=50, internal_batch_size=100)*

**Bases:** alibi.api.interfaces.Explainer

**__init__** *(model, layer=None, method='gausslegendre', n_steps=50, internal_batch_size=100)*

An implementation of the integrated gradients method for Tensorflow and Keras models.

For details of the method see the original paper: https://arxiv.org/abs/1703.01365.

**Parameters**
- **model** (`Union[Model, ForwardRef]`) – Tensorflow or Keras model.
- **layer** (`Union[Layer, ForwardRef, List[Layer], List[ForwardRef]]`) – Layers with respect to which the gradients are calculated. It can be a single layer or a list of layers. If not provided, the gradients are calculated with respect to the input.
- **n_steps** (`int`) – Number of step in the path integral approximation from the baseline to the input instance.
- **internal_batch_size** (`int`) – Batch size for the internal batching.

**Return type** None

**build_explanation** (`X, baselines, target, attributions`)  
**Return type** *Explanation*

**explain** (`X, baselines=None, target=None`)  
Calculates the attributions for each input feature or element of layer and returns an Explanation object.

**Parameters**
- **X** (`Union[ndarray, List[ndarray]]`) – Instance for which integrated gradients attribution are computed.
- **baselines** (`Union[int, float, ndarray, List[int], List[float], List[ndarray], None]`) – Baselines (starting point of the path integral) for each instance. If the passed value is an np.ndarray must have the same shape as X. If not provided, all features values for the baselines are set to 0.
- **target** (`Union[int, list, ndarray, None]`) – Defines which element of the model output is considered to compute the gradients. It can be a list of integers or a numeric value. If a numeric value is passed, the gradients are calculated for the same element of the output for all data points. It must be provided if the model output dimension is higher than 1. For regression models whose output is a scalar, target should not be provided. For classification models target can be either the true classes or the classes predicted by the model.

**Return type** *Explanation*

**Returns**
- *Explanation* object including meta and data attributes with integrated gradients attributions
- *for each feature.*

**alibi.explainers.shap_wrappers module**

**alibi.explainers.shap_wrappers.DISTRIBUTED_OPTS: dict = {'batch_size': 1, 'n_cpus': None}**
Default distributed options for KernelShap:
- 'ncpus': int, number of available CPUs available to parallelize explanations. Performance is significantly boosted when the number specified represents physical CPUs, but small (nonlinear) gains are observed when virtual CPUs are specified. If set to None, the code will run sequentially.
- 'batch_size': int, how many instances are explained in the same remote process at once. The shap library of KernelShap is not vectorised, so no significant gains are made by specifying batches. See blog
post for batch size experiments results. If set to None, an input array is split in (roughly) equal parts and distributed across the available CPUs.

```python
class alibi.explainers.shap_wrappers.KernelExplainerWrapper(*args, **kwargs)
```

A wrapper around `shap.KernelExplainer` that supports:

- fixing the seed when instantiating the KernelExplainer in a separate process
- passing a batch index to the explainer so that a parallel explainer pool can return batches in arbitrary order

```python
__init__(*args, **kwargs)
```

**Parameters**

- `kwargs` — Arguments and keyword arguments for `shap.KernelExplainer` constructor.

```python
get_explanation(X, **kwargs)
```

Wrapper around `shap.KernelExplainer.shap_values` that allows calling the method with a tuple containing a batch index and a batch of instances.

**Parameters**

- `X` (Union[Tuple[int, ndarray], ndarray]) — When called from a distributed context, it is a tuple containing a batch index and a batch to be explained. Otherwise, it is an array of instances to be explained.
- `kwargs` — `shap.KernelExplainer.shap_values` keyword values.

**Return type**

Union[Tuple[int, ndarray], Tuple[int, List[ndarray]], ndarray, List[ndarray]]

```python
return_attribute(name)
```

Returns an attribute specified by its name. Used in a distributed context where the actor properties cannot be accessed using the dot syntax.

**Return type**

Any

```python
class alibi.explainers.shap_wrappers.KernelShap(predictor, link='identity',
feature_names=None, categorical_names=None, task='classification', seed=None,
distributed_opts=None)
```

A wrapper around the `shap.KernelExplainer` class. It extends the current `shap` library functionality by allowing the user to specify variable groups in order to treat one-hot encoded categorical as one during sampling. The user can also specify whether to aggregate the `shap` values estimate for the encoded levels of categorical variables as an optional argument to `explain`, if grouping arguments are not passed to `fit`.

**Parameters**

- `predictor` (Callable) — A callable that takes as an input a samples x features array and outputs a samples x n_outputs model outputs. The n_outputs should represent model output in margin space. If the model outputs probabilities, then the link should be set to ‘logit’ to ensure correct force plots.
- `link` (str) — Valid values are ‘identity’ or ‘logit’. A generalized linear model link to connect the feature importance values to the model output. Since the feature importance values, $\phi$, sum up to the model output, it often makes sense to connect them to the output with a link function where $link(output - expected\_value) = sum(\phi)$. 

432 Chapter 44. alibi
Therefore, for a model which outputs probabilities, `link='logit'` makes the feature effects have log-odds (evidence) units and `link='identity'` means that the feature effects have probability units. Please see this example for an in-depth discussion about the semantics of explaining the model in the probability or margin space.

- **feature_names** (Union[List[str], Tuple[str], None]) – Used to infer group names when categorical data is treated by grouping and `group_names` input to `fit` is not specified, assuming it has the same length as the `groups` argument of `fit` method. It is also used to compute the `names` field, which appears as a key in each of the values of `explanation.data[‘raw’][‘importances’].`

- **categorical_names** (Optional[Dict[int, List[str]]]) – Keys are feature column indices in the `background_data` matrix (see `fit`). Each value contains strings with the names of the categories for the feature. Used to select the method for background data summarisation (if specified, subsampling is performed as opposed to k-means clustering). In the future it may be used for visualisation.

- **task** (str) – Can have values ‘classification’ and ‘regression’. It is only used to set the contents of `explanation.data[‘raw’][‘prediction’]`

- **seed** (Optional[int]) – Fixes the random number stream, which influences which subsets are sampled during shap value estimation.

- **distributed_opts** (Optional[Dict]) – A dictionary that controls the algorithm distributed execution. See `DISTRIBUTED_OPTS documentation for details.`

**build_explanation**(X, shap_values, expected_value, **kwargs)
Create an explanation object. If output summarisation is required and all inputs necessary for this operation are passed, the raw shap values are summed first so that a single shap value is returned for each categorical variable, as opposed to a shap value per dimension of categorical variable encoding.

Parameters

- **X** (Union[ndarray, DataFrame, spmatrix]) – Instances to be explained.

- **shap_values** (List[ndarray]) – Each entry is a n_instances x n_features array, and the length of the list equals the dimensionality of the predictor output. The rows of each array correspond to the shap values for the instances with the corresponding row index in X. The length of the list equals the number of model outputs.

- **expected_value** (List[float]) – A list containing the expected value of the prediction for each class. Its length should be equal to that of `shap_values`.

Return type **Explanation**

Returns

explanation – An explanation object containing the shap values and prediction in the data field, along with a meta field containing additional data. See usage examples for details.

**explain**(X, summarise_result=False, cat_vars_start_idx=None, cat_vars_enc_dim=None, **kwargs)
Explains the instances in the array X.

Parameters

- **X** (Union[ndarray, DataFrame, spmatrix]) – Instances to be explained.

- **summarise_result** (bool) – Specifies whether the shap values corresponding to dimensions of encoded categorical variables should be summed so that a single shap value is returned for each categorical variable. Both the start indices of the categorical variables (cat_vars_start_idx) and the encoding dimensions (cat_vars_enc_dim) have to be specified.
• **cat_vars_start_idx** *(Optional[Sequence[int]])* – The start indices of the categorical variables. If specified, **cat_vars_enc_dim** should also be specified.

• **cat_vars_enc_dim** *(Optional[Sequence[int]])* – The length of the encoding dimension for each categorical variable. If specified **cat_vars_start_idx** should also be specified.

• **kwargs** – Keyword arguments specifying explain behaviour. Valid arguments are:
  
  – **nsamples**: controls the number of predictor calls and therefore runtime.
  
  – **l1_reg**: the algorithm is exponential in the feature dimension. If set to **auto** the algorithm will first run a feature selection algorithm to select the top features, provided the fraction of sampled sets of missing features is less than 0.2 from the number of total subsets. The Akaike Information Criterion is used in this case. See our examples for more details about available settings for this parameter. Note that by first running a feature selection step, the shapley values of the remainder of the features will be different to those estimated from the entire set.

  For more details, please see the shap library [documentation](https://shap.readthedocs.io/en/latest/).

Return type **Explanation**

Returns

`explanation` – An explanation object containing the shap values and prediction in the `data` field, along with a `meta` field containing additional data. See usage examples for details.

**fit** *(background_data, summarise_background=False, n_background_samples=300, group_names=None, groups=None, weights=None, **kwargs)*

This takes a background dataset (usually a subsample of the training set) as an input along with several user specified options and initialises a *KernelShap* explainer. The runtime of the algorithm depends on the number of samples in this dataset and on the number of features in the dataset. To reduce the size of the dataset, the `summarise_background` option and `n_background_samples` should be used. To reduce the feature dimensionality, encoded categorical variables can be treated as one during the feature perturbation process; this decreases the effective feature dimensionality, can reduce the variance of the shap values estimation and reduces slightly the number of calls to the predictor. Further runtime savings can be achieved by changing the `nsamples` parameter in the call to explain. Runtime reduction comes with an accuracy trade-off, so it is better to experiment with a runtime reduction method and understand results stability before using the system.

Parameters

• **background_data** *(Union[ndarray, spmatrix, DataFrame, Data])* – Data used to estimate feature contributions and baseline values for force plots. The rows of the background data should represent samples and the columns features.

• **summarise_background** *(Union[bool, str])* – A large background dataset impacts the runtime and memory footprint of the algorithm. By setting this argument to **True**, only `n_background_samples` from the provided data are selected. If `group_names` or `groups` arguments are specified, the algorithm assumes that the data contains categorical variables so the records are selected uniformly at random. Otherwise, *sklearn* k-means (a wrapper around *sklearn* k-means implementation) is used for selection. If set to ‘**auto**’, a default of `KERNEL_SHAP_BACKGROUND_THRESHOLD` samples is selected.

• **n_background_samples** *(int)* – The number of samples to keep in the background dataset if `summarise_background=True`. 
• **groups** *(Optional[List[Union[Tuple[int], List[int]]]])* – A list containing sub-lists specifying the indices of features belonging to the same group.

• **group_names** *(Union[List[str], Tuple[str], None]*) – If specified, this array is used to treat groups of features as one during feature perturbation. This feature can be useful, for example, to treat encoded categorical variables as one and can result in computational savings (this may require adjusting the nsamples parameter).

• **weights** *(Union[List[float], Tuple[float], ndarray, None]*) – A sequence or array of weights. This is used only if grouping is specified and assigns a weight to each point in the dataset.

• **kwargs** – Expected keyword arguments include keep_index (bool) and should be used if a data frame containing an index column is passed to the algorithm.

Return type **KernelShap**

```python
class alibi.explainers.shap_wrappers.TreeShap(predictor, model_output='raw', feature_names=None, categorical_names=None, task='classification', seed=None)
```

Bases: alibi.api.interfaces.Explainer, alibi.api.interfaces.FitMixin

__init__ (predictor, model_output='raw', feature_names=None, categorical_names=None, task='classification', seed=None)

A wrapper around the shap.TreeExplainer class. It adds the following functionality:

1. Input summarisation options to allow control over background dataset size and hence runtime
2. Output summarisation for sklearn models with one-hot encoded categorical variables.

Users are strongly encouraged to familiarise themselves with the algorithm by reading the method overview in the documentation.

Parameters

• **predictor** *(Any)* – A fitted model to be explained. XGBoost, LightGBM, CatBoost and most tree-based scikit-learn models are supported. In the future, Pyspark could also be supported. Please open an issue if this is a use case for you.

• **model_output** *(str)* – Supported values are: ‘raw’, ‘probability’, ‘probability_doubled’, ‘log_loss’:
  - ‘raw’: the raw model of the output, which varies by task, is explained. This option should always be used if the fit is called without arguments. It should also be set to compute shap interaction values. For regression models it is the standard output, for binary classification in XGBoost it is the log odds ratio.
  - ‘probability’: the probability output is explained. This option should only be used if fit was was called with the background_data argument set. The effect of specifying this parameter is that the shap library will use this information to transform the shap values computed in margin space (aka using the raw output) to shap values that sum to the probability output by the model plus the model expected output probability. This requires knowledge of the type of output for predictor which is inferred by the shap library from the model type (e.g., most sklearn models with exception of sklearn.tree.DecisionTreeClassifier, sklearn.ensemble.RandomForestClassifier, sklearn.ensemble.ExtraTreesClassifier output logits) or on the basis of the mapping implemented in the shap.TreeEnsemble constructor. Only trees that output log odds and probabilities are supported currently.
  - ‘probability_doubled’: used for binary classification problem in situations where the model outputs the logits/probabilities for the positive class but shap values for
both outcomes are desired. This option should be used only if `fit` was called with the `background_data` argument set. In this case the expected value for the negative class is 1 - expected_value for positive class and the shap values for the negative class are the negative values of the positive class shap values. As before, the explanation happens in the margin space, and the shap values are subsequently adjusted. convert the model output to probabilities. The same considerations as for probability apply for this output type too.

- `log_loss`: logarithmic loss is explained. This option should be used only if `fit` was called with the `background_data` argument set and requires specifying labels, `y`, when calling `explain`. If the objective is squared error, then the transformation \((output - y)^2\) is applied. For binary cross-entropy objective, the transformation \(\log(1 + \exp(output)) - y \times output\) with \(y \in \{0, 1\}\). Currently only binary cross-entropy and squared error losses can be explained.

• `feature_names` (Union[List[str], Tuple[str, None]) – Used to compute the `names` field, which appears as a key in each of the values of the `importances` sub-field of the response `raw` field.

• `categorical_names` (Optional[Dict[int, List[str]]]) – Keys are feature column indices. Each value contains strings with the names of the categories for the feature. Used to select the method for background data summarisation (if specified, subsampling is performed as opposed to kmeans clustering). In the future it may be used for visualisation.

• `task` (str) – Can have values ‘classification’ and ‘regression’. It is only used to set the contents of the `prediction` field in the `data['raw']` response field.

Notes

Tree SHAP is an additive attribution method so it is best suited to explaining output in margin space (the entire real line). For discussion related to explaining models in output vs probability space, please consult this resource.

`build_explanation` (X, shap_output, expected_value, **kwargs)

Create an explanation object. If output summarisation is required and all inputs necessary for this operation are passed, the raw shap values are summed first so that a single shap value is returned for each categorical variable, as opposed to a shap value per dimension of categorical variable encoding. Similarly, the shap interaction values are summed such that they represent the interaction between categorical variables as opposed to levels of categorical variables. If the interaction option has been specified during `explain`, this method computes the shap values given the interactions prior to creating the response.

Parameters

• `X` (Union[ndarray, DataFrame, ForwardRef]) – Instances to be explained.

• `shap_output` (List[ndarray]) – If `explain` is called with `interactions=True` then the list contains tensors of dimensionality `n_instances x n_features x n_features` of shap interaction values. Otherwise, it contains tensors of dimension `n_instances x n_features` representing shap values. The length of the list equals the number of model outputs.

• `expected_value` (List[float]) – A list containing the expected value of the prediction for each class. Its length is equal to that of `shap_output`.

Return type Explanation

Returns
Explanation – An Explanation object containing the shap values and prediction in the data field, along with a meta field containing additional data. See usage examples here for details.

```
explain(X, y=None, interactions=False, approximate=False, check_additivity=True, tree_limit=None, summarise_result=False, cat_vars_start_idx=None, cat_vars_enc_dim=None, **kwargs)
```

Explains the instances in X. y should be passed if the model loss function is to be explained, which can be useful in order to understand how various features affect model performance over time. This is only possible if the explainer has been fitted with a background dataset and requires setting model_output='log_loss'.

Parameters

- **X** (*Union[ndarray, DataFrame, ForwardRef]*) – Instances to be explained.
- **y** (*Optional[ndarray]*) – Labels corresponding to rows of X. Should be passed only if a background dataset was passed to the fit method.
- **interactions** (*bool*) – If True, the shap value for every feature of every instance in X is decomposed into \(X.shape[1] - 1\) shap value interactions and one main effect. This is only supported if fit is called with background_dataset=None.
- **approximate** (*bool*) – If True, an approximation to the shap values that does not account for feature order is computed. This was proposed by Ando Sabaas here. Check this resource for more details. This option is currently only supported for xgboost and sklearn models.
- **check_additivity** (*bool*) – If True, output correctness is ensured if model_output='raw' has been passed to the constructor.
- **tree_limit** (*Optional[int]*) – Explain the output of a subset of the first tree_limit trees in an ensemble model.
- **summarise_result** (*bool*) – This should be set to True only when some of the columns in X represent encoded dimensions of a categorical variable and one single shap value per categorical variable is desired. Both cat_vars_start_idx and cat_vars_enc_dim should be specified as detailed below to allow this.
- **cat_vars_start_idx** (*Optional[Sequence[int]*) – The start indices of the categorical variables.
- **cat_vars_enc_dim** (*Optional[Sequence[int]*) – The length of the encoding dimension for each categorical variable.

Return type **Explanation**

Returns

```
explanation – An Explanation object containing the shap values and prediction in the data field, along with a meta field containing additional data. See usage examples here for details.
```

```
fit(background_data=None, summarise_background=False, n_background_samples=1000, **kwargs)
```

This function instantiates an explainer which can then be used to explain instances using the explain method. If no background dataset is passed, the explainer uses the path-dependent feature perturbation algorithm to explain the values. As such, only the model raw output can be explained and this should be reflected by passing model_output='raw' when instantiating the explainer. If a background dataset is passed, the interventional feature perturbation algorithm is used. Using this algorithm, probability outputs can also be explained. Additionally, if the model_output='log_loss' option is passed to the explainer constructor, then the model loss function can be explained by passing the labels as the y argument to
the explain method. A limited number of loss functions are supported, as detailed in the constructor
documentation.

**Parameters**

- **background_data** *(Union[ndarray, DataFrame, None])* – Data used to estimate feature contributions and baseline values for force plots. The rows of the background data should represent samples and the columns features.

- **summarise_background** *(Union[bool, str])* – A large background dataset may impact the runtime and memory footprint of the algorithm. By setting this argument to `True`, only `n_background_samples` from the provided data are selected. If the `categorical_names` argument has been passed to the constructor, subsampling of the data is used. Otherwise, `shap.kmeans` (a wrapper around `sklearn.kmeans` implementation) is used for selection. If set to ‘auto’, a default of `TREE_SHAP_BACKGROUND_WARNING_THRESHOLD` samples is selected.

- **n_background_samples** *(int)* – The number of samples to keep in the background dataset if `summarise_background=True`.

**Return type** `TreeShap`

*alibi.explainers.shap_wrappers.rank_by_importance* *(shap_values, feature_names=None)*

Given the shap values estimated for a multi-output model, this function ranks features according to their importance. The feature importance is the average absolute value for a given feature.

**Parameters**

- **shap_values** *(List[ndarray])* – Each element corresponds to a samples x features array of shap values corresponding to each model output.

- **feature_names** *(Union[List[str], Tuple[str], None])* – Each element is the name of the column with the corresponding index in each of the arrays in the `shap_values` list.

**Return type** `Dict`

**Returns**

`importances` –

A dictionary of the form:

```python
{
    '0': {'ranked_effect': array([0.2, 0.5, ...]), 'names': ['feat_3', 'feat_5', ...]},
    '1': {'ranked_effect': array([0.3, 0.2, ...]), 'names': ['feat_6', 'feat_1', ...]},
    ...
    'aggregated': {'ranked_effect': array([0.9, 0.7, ...]), 'names': ['feat_3', 'feat_6', ...]}
}
```

The keys of the first level represent the index of the model output. The feature effects in `ranked_effect` and the corresponding feature names in `names` are sorted from highest (most important) to lowest (least important). The values in the `aggregated` field are obtained by summing the shap values for all the model outputs and then computing the effects. Given an output, the effects are defined as the average magnitude of the shap values across the instances to be explained.
This function is used to reduce specified slices in a two- or three- dimensional tensor.

For two-dimensional values arrays, for each entry in start_idx, the function sums the following k columns where k is the corresponding entry in the enc_feat_dim sequence. The columns whose indices are not in start_idx are left unchanged. This arises when the slices contain the shap values for each dimension of an encoded categorical variable and a single shap value for each variable is desired.

For three-dimensional values arrays, the reduction is applied for each rank 2 subtensor, first along the column dimension and then across the row dimension. This arises when summarising shap interaction values. Each rank 2 tensor is a E x E matrix of shap interaction values, where E is the dimension of the data after one-hot encoding. The result of applying the reduction yields a rank 2 tensor of dimension F x F, where F is the number of features (i.e., the feature dimension of the data matrix before encoding). By applying this transformation, a single value describing the interaction of categorical features i and j and a single value describing the interaction of j and i is returned.

Parameters

- values (ndarray) – A two or three dimensional array to be reduced, as described above.
- start_idx (Sequence[int]) – The start indices of the columns to be summed.
- enc_feat_dim (Sequence[int]) – The number of columns to be summed, one for each start index.

Returns new_values – An array whose columns have been summed according to the entries in start_idx and enc_feat_dim.

alibi.tests package

Submodules

alibi.tests.utils module

class alibi.tests.utils.MockPredictor(out_dim, out_type='proba', model_type=None, seed=None)

Bases: object

A class the mimicks the output of a classifier or regressor to allow testing of functionality that depends on it without inference overhead.

__init__(out_dim, out_type='proba', model_type=None, seed=None)

Parameters

- out_dim (int) – The number of output classes.
- out_type (str) – Indicates if probabilities, class predictions or continuous outputs are generated.

Return type None

predict (*args, **kwargs)

alibi.tests.utils.assert_message_in_logs(msg, records)

Helper function to check if a msg is present in any of the records (an iterable of strings).

alibi.tests.utils.issorted(arr, reverse=False)

Checks if a numpy array is sorted.
alibiDocumentation, Release 0.5.6

alibi.tests.utils.

```
not_raises(ExpectedException)
```

A context manager used to check that `ExpectedException` does not occur during testing.

alibi.utils package

Submodules

alibi.utils.approximation_methods module

```
class alibi.utils.approximation_methods.Riemann
    Bases: enum.Enum
    An enumeration.
    left = 1
    middle = 3
    right = 2
    trapezoid = 4
```

```
alibi.utils.approximation_methods.approximation_parameters(method)
```

Retrieves parameters for the input approximation method

```
Parameters method (str) – The name of the approximation method. Currently only `riemann` and `gauss_legendre` are

Return type Tuple[Callable[[int], List[float]], Callable[[int], List[float]]]
```

```
aliби.utils.approximation_methods.

```

```
```

```

Gauss Legendre quadrature rule for approximating the integrals was originally proposed by [Xue Feng and her intern Hauroun Habeeb](https://research.fb.com/people/feng-xue/).

```
Parameters n – The number of integration steps

Return type Tuple[Callable[[int], List[float]], Callable[[int], List[float]]]
```

```

Returns
```

```
```

```

Step sizes are identical and alphas are scaled in [0, 1]

```
```

Parameters
```

```
```

```

```

```

```

440 Chapter 44. alibi
Returns

2-element tuple of **step_sizes**, **alphas** –

- **step_sizes (callable)**: `step_sizes` takes the number of steps as an input argument and returns an array of steps sizes which sum is smaller than or equal to one.

- **alphas (callable)**: `alphas` takes the number of steps as an input argument and returns the multipliers/coefficients for the inputs of integrand in the range of [0, 1]

alibi.utils.data module

class alibi.utils.data.Bunch(**kwargs)

Bases: dict

Container object for internal datasets Dictionary-like object that exposes its keys as attributes.

alibi.utils.data.gen_category_map(data, categorical_columns=None)

Parameters

- **data** (Union[DataFrame, ndarray]) – 2-dimensional pandas dataframe or numpy array.

- **categorical_columns** (Union[List[int], List[str], None]) – A list of columns indicating categorical variables. Optional if passing a pandas dataframe as inference will be used based on dtype ‘O’. If passing a numpy array this is compulsory.

Return type Dict[int, list]

Returns category_map – A dictionary with keys being the indices of the categorical columns and values being lists of categories for that column. Implicitly each category is mapped to the index of its position in the list.

alibi.utils.discretizer module

class alibi.utils.discretizer.Discretizer(data, numerical_features, feature_names, percentiles=(25, 50, 75))

Bases: object

__init__ (data, numerical_features, feature_names, percentiles=(25, 50, 75))

Initialize the discretizer

Parameters

- **data** (ndarray) – Data to discretize

- **numerical_features** (List[int]) – List of indices corresponding to the continuous feature columns. Only these features will be discretized.

- **feature_names** (List[str]) – List with feature names

- **percentiles** (Sequence[Union[int, float]]) – Percentiles used for discretization

Return type None

bins(data)

Parameters data (ndarray) – Data to discretize.

Return type List[ndarray]
Returns List with bin values for each feature that is discretized.

discretize(data)

Parameters data (ndarray) – Data to discretize

Return type ndarray

Returns Discretized version of data with the same dimension.

static get_percentiles(x, qts)

Discretizes the data in x using the quantiles in qts. This is achieved by searching for the index of each value in x into qts, which is assumed to be a 1-D sorted array.

Parameters

• x (ndarray) – A tensor of data to be discretized

• qts (ndarray) – A percentiles array. This should be a 1-D array sorted in ascending order.

Return type ndarray

Returns A discretized data tensor.

alibi.utils.distance module

alibi.utils.distance.abdm(X, cat_vars, cat_vars_bin={})


Parameters

• X (ndarray) – Batch of arrays.

• cat_vars (dict) – Dict with as keys the categorical columns and as optional values the number of categories per categorical variable.

• cat_vars_bin (dict) – Dict with as keys the binned numerical columns and as optional values the number of bins per variable.

Returns Dict with as keys the categorical columns and as values the pairwise distance matrix for the variable.

alibi.utils.distance.cityblock_batch(X, y)

Calculate the L1 distances between a batch of arrays X and an array of the same shape y.

Parameters

• X (ndarray) – Batch of arrays to calculate the distances from

• y (ndarray) – Array to calculate the distance to

Return type ndarray

Returns Array of distances from each array in X to y

alibi.utils.distance.multidim_scaling(d_pair, n_components=2, use_metric=True, standardize_cat_vars=True, feature_range=None, smooth=1.0, center=True, update_feature_range=True)

Apply multidimensional scaling to pairwise distance matrices.

Parameters
• **d_pair** *(dict)* – Dict with as keys the column index of the categorical variables and as values a pairwise distance matrix for the categories of the variable.

• **n_components** *(int)* – Number of dimensions in which to immerse the dissimilarities.

• **use_metric** *(bool)* – If True, perform metric MDS; otherwise, perform nonmetric MDS.

• **standardize_cat_vars** *(bool)* – Standardize numerical values of categorical variables if True.

• **feature_range** *(Optional[tuple])* – Tuple with min and max ranges to allow for perturbed instances. Min and max ranges can be floats or numpy arrays with dimension (1 x nb of features) for feature-wise ranges.

• **smooth** *(float)* – Smoothing exponent between 0 and 1 for the distances. Lower values of 1 will smooth the difference in distance metric between different features.

• **center** *(bool)* – Whether to center the scaled distance measures. If False, the min distance for each feature except for the feature with the highest raw max distance will be the lower bound of the feature range, but the upper bound will be below the max feature range.

• **update_feature_range** *(bool)* – Update feature range with scaled values.

**Return type** *Tuple[dict, tuple]*

**Returns** *Dict with multidimensional scaled version of pairwise distance matrices.*

`alibi.utils.distance.mvdm(X, y, cat_vars, alpha=1)`


**Parameters**

• **X** *(ndarray)* – Batch of arrays.

• **y** *(ndarray)* – Batch of labels or predictions.

• **cat_vars** *(dict)* – Dict with as keys the categorical columns and as optional values the number of categories per categorical variable.

• **alpha** *(int)* – Power of absolute difference between conditional probabilities.

**Return type** *ndarray*

**Returns** *Dict with as keys the categorical columns and as values the pairwise distance matrix for the variable.*

### `alibi.utils.distributed module`

**class** `alibi.utils.distributed.ActorPool(actors)`

**Bases:** *object*

**__init__**(actors)

Taken from the ray repository: [https://github.com/ray-project/ray/pull/5945](https://github.com/ray-project/ray/pull/5945) Create an Actor pool from a list of existing actors. An actor pool is a utility class similar to multiprocessing.Pool that lets you schedule Ray tasks over a fixed pool of actors.

**Parameters**

: *actors*:
  - List of Ray actor handles to use in this pool.

**Return type** *list*
Examples

```python
>>> a1, a2 = Actor.remote(), Actor.remote()
>>> pool = ActorPool([a1, a2])
>>> print(pool.map(lambda a, v: a.double.remote(v), [1, 2, 3, 4]))
[2, 4, 6, 8]
```

**get_next** *(timeout=None)*

Returns the next pending result in order. This returns the next result produced by submit(), blocking for up to the specified timeout until it is available. :returns: The next result.

Raises TimeoutError if the timeout is reached.

Examples

```python
>>> pool = ActorPool(...)
>>> pool.submit(lambda a, v: a.double.remote(v), 1)
>>> print(pool.get_next())
2
```

**get_next_unordered** *(timeout=None)*

Returns any of the next pending results. This returns some result produced by submit(), blocking for up to the specified timeout until it is available. Unlike get_next(), the results are not always returned in same order as submitted, which can improve performance. :returns: The next result.

Raises TimeoutError if the timeout is reached.

Examples

```python
>>> pool = ActorPool(...)
>>> pool.submit(lambda a, v: a.double.remote(v), 1)
>>> pool.submit(lambda a, v: a.double.remote(v), 2)
>>> print(pool.get_next_unordered())
4
>>> print(pool.get_next_unordered())
2
```

**has_next**

Returns whether there are any pending results to return.

Returns True if there are any pending results not yet returned.

Examples

```python
>>> pool = ActorPool(...)
>>> pool.submit(lambda a, v: a.double.remote(v), 1)
>>> print(pool.has_next())
True
>>> print(pool.get_next())
2
>>> print(pool.has_next())
False
```
map \((fn, values, chunksize=1)\)

Apply the given function in parallel over the actors and values. This returns an ordered iterator that will return results of the map as they finish. Note that you must iterate over the iterator to force the computation to finish.

Parameters

- (func) \((fn)\) – Function that takes (actor, value) as argument and returns an ObjectID computing the result over the value. The actor will be considered busy until the ObjectID completes.
- (list) \((values)\) – List of values that fn(actor, value) should be applied to.
- (int) \((chunksize)\) – splits the list of values to be submitted to the parallel process into sublists of size chunksize or less

Returns Iterator over results from applying fn to the actors and values.

Examples

```python
>>> pool = ActorPool(...)
>>> print(pool.map(lambda a, v: a.double.remote(v), [1, 2, 3, 4]))
[2, 4, 6, 8]
```

map_unordered \((fn, values, chunksize=1)\)

Similar to map(), but returning an unordered iterator. This returns an unordered iterator that will return results of the map as they finish. This can be more efficient than map() if some results take longer to compute than others.

Parameters

- (func) \((fn)\) – Function that takes (actor, value) as argument and returns an ObjectID computing the result over the value. The actor will be considered busy until the ObjectID completes.
- (list) \((values)\) – List of values that fn(actor, value) should be applied to.
- (int) \((chunksize)\) – splits the list of values to be submitted to the parallel process into sublists of size chunksize or less

Returns Iterator over results from applying fn to the actors and values.

Examples

```python
>>> pool = ActorPool(...)
>>> print(pool.map(lambda a, v: a.double.remote(v), [1, 2, 3, 4]))
[6, 2, 4, 8]
```

submit \((fn, value)\)

Schedule a single task to run in the pool. This has the same argument semantics as map(), but takes on a single value instead of a list of values. The result can be retrieved using get_next() / get_next_unordered().

Parameters

- (func) \((fn)\) – Function that takes (actor, value) as argument and returns an ObjectID computing the result over the value. The actor will be considered busy until the ObjectID completes.
- (object) \((value)\) –
Examples

```python
>>> pool = ActorPool(...)  
>>> pool.submit(lambda a, v: a.double.remote(v), 1)  
>>> pool.submit(lambda a, v: a.double.remote(v), 2)  
>>> print(pool.get_next(), pool.get_next())  
2, 4
```

def alibi.utils.distributed.DistributedExplainer(distributed_opts, explainer_type, explainer_init_args, explainer_init_kwargs, concatenate_results=True, return_generator=False):
    Bases: object
    A class that orchestrates the execution of the execution of a batch of explanations in parallel.

    __getattr__(item)
    Accesses actor attributes. Use sparingly as this involves a remote call (that is, these attributes are of an object in a different process). The intended use is for retrieving any common state across the actor at the end of the computation in order to form the response (see notes 2 & 3).

    Parameters
    item (str) – The explainer attribute to be returned.

    Return type
    Any

    Returns
    The value of the attribute specified by item.

    Raises
    ValueError – If the actor index is invalid.

Notes

1. This method assumes that the actor implements a return_attribute method.
2. Note that we are indexing the idle actors. This means that if a pool was initialised with 5 actors and 3 are busy, indexing with index 2 will raise an IndexError.
3. The order of _idle_actors constantly changes - an actor is removed from it if there is a task to execute and appended back when the task is complete. Therefore, indexing at the same position as computation proceeds will result in retrieving state from different processes.

def __init__(distributed_opts, explainer_type, explainer_init_args, explainer_init_kwargs, concatenate_results=True, return_generator=False):
    Creates a pool of actors (i.e., replicas of an instantiated explainer_type in a separate process) which can explain batches of instances in parallel via calls to get_explanation.

    Parameters

    * concatenate_results (bool)
    * distributed_opts (Dict[Dict]) – A dictionary with the following type (minimal signature):

```python
class DistributedOpts(TypedDict):
    n_cpus: Optional[int]
    batch_size: Optional[int]
```

The dictionary may contain two additional keys:
- 'actor_cpu_frac' (float, <= 1.0, >0.0): This is used to create more than one process on one CPU/GPU. This may not speed up CPU intensive tasks but it is worth experimenting with when few physical cores are available. In particular, this is highly useful when the user wants to share a GPU for multiple tasks, with the caviat that the machine learning framework itself needs to support running multiple replicas on the same GPU. See the ray documentation here for details.

- 'algorithm': this is specified internally by the caller. It is used in order to register target function callbacks for the parallel pool. These should be implemented in the global scope. If not specified, its value will be 'default', which will select a default target function which expects the actor has a get_explanation method.

- explainer_type (Any) – Explainer class.
- explainer_init_args (Tuple) – Positional arguments to explainer constructor.
- explainer_init_kwargs (dict) – Keyword arguments to explainer constructor.
- return_generator (bool) – If True a generator that returns the results in the order the computation finishes is returned when get_explanation is called. Otherwise, the order of the results is the same as the order of the minibatches.

**Notes**

When return_generator=True, the caller has to take elements from the generator (e.g., by calling next) in order to start computing the results (because the ray pool is implemented as a generator).

**property actor_index**

Returns the index of the actor for which state is returned.

**Return type** int

**create_parallel_pool** (explainer_type, explainer_init_args, explainer_init_kwargs)

Creates a pool of actors that can explain the rows of a dataset in parallel.

**Parameters** constructor documentation. (See) –

**get_explanation** (X, **kwargs)

Performs distributed explanations of instances in X.

**Parameters**

- X (ndarray) – A batch of instances to be explained. Split into batches according to the settings passed to the constructor.
- kwargs – Any keyword-arguments for the explainer explain method.

**Return type** Union[Generator[Tuple[int, Any], None, None], List[Any], Any]

**Returns**

The explanations are returned as –

- a generator, if the return_generator option is specified. This is used so that the caller can access the results as they are computed. This is the only case when this method is non-blocking and the caller needs to call next on the generator to trigger the parallel computation
• a list of objects, whose type depends on the return type of the explainer. This is returned if no custom preprocessing function is specified

• an object, whose type depends on the return type of the concatenation function return when called with a list of minibatch results with the same order as the minibatches

return_attribute(name)

Returns an attribute specified by its name. Used in a distributed context where the properties cannot be accessed using the dot syntax.

Return type Any

set_actor_index(value)

Sets actor index. This is used when the DistributedExplainer is in a separate process because ray does not support calling property setters remotely

class alibi.utils.distributed.PoolCollection(distributed_opts, explainer_type, explainer_init_args, explainer_init_kwargs, **kwargs)

Bases: object

A wrapper object that turns a DistributedExplainer into a remote actor. This allows running multiple distributed explainers in parallel.

__getattr__(item)

Access attributes of the distributed explainer or the distributed explainer contained.

Return type Any

__init__(distributed_opts, explainer_type, explainer_init_args, explainer_init_kwargs, **kwargs)

Initialises a list of distinct distributed explainers which can explain the same batch in parallel. It generalizes the DistributedExplainer, which contains replicas of one explainer object, speeding up the task of explaining batches of instances.

Parameters

• , explainer_type, explainer_init_args, explainer_init_kwargs (distributed_opts) – See DistributedExplainer constructor documentation for explanations. Each entry in the list is a different explainer configuration (e.g., CEM in PN vs PP mode, different background dataset sizes for SHAP, etc).

• kwargs – Any other kwargs, passed to the DistributedExplainer objects.

Raises

• ResourceError – If the number of CPUs specified by the user is smaller than the number of distributed explainers.

• ValueError – If the number of entries in the explainers args/kwargs list differ.

static create_explainer_handles(distributed_opts, explainer_type, explainer_init_args, explainer_init_kwargs, **kwargs)

Creates multiple actors for DistributedExplainer so that tasks can be executed in parallel. The actors are initialised with different arguments, so they represent different explainers.

Parameters explainer_type, explainer_init_args, explainer_init_kwargs, kwargs (distributed_opts,) – See constructor.

get_explanation(X, **kwargs)

Calls a collection of distributed explainers in parallel. Each distributed explainer will explain each row in X in parallel.
Parameters

- **X** – Batch of instances to be explained.

**Return type** List

**Returns** A list of responses collected from each explainer.

**Notes**

Note that the call to `ray.get` is blocking.

**Raises** TypeError – If the user sets `return_generator=True` for the DistributedExplainer. This is because generators cannot be pickled so one cannot call `ray.get`.

**property remote_explainer_index**

Returns the index of the actor for which state is returned.

**Return type** int

**exception** alibi.utils.distributed.ResourceError

Bases: Exception

**alibi.utils.distributed.batch**(X, batch_size=None, n_batches=4)

Splits the input into sub-arrays.

**Parameters**

- **X** (ndarray) – Array to be split.

- **batch_size** (Optional[int]) – The size of each batch. In particular:
  - if `batch_size` is not None, batches of this size are created. The sizes of the batches created might vary if the 0-th dimension of `X` is not divisible by `batch_size`. For an array of length `l` that should be split into `n` sections, it returns `l % n` sub-arrays of size `l/n + 1` and the rest of size `l/n`
  - if `batch_size` is None, then `X` is split into `n_batches` sub-arrays.

- **n_batches** (int) – Number of batches in which to split the sub-array. Only used if `batch_size` = None

**Return type** List[ndarray]

**Returns** A list of sub-arrays of `X`.

**alibi.utils.distributed.check_ray()**

Checks if ray is installed.

**Return type** bool

**Returns** A bool indicating whether ray is installed or not.

**alibi.utils.distributed.concatenate_minibatches**(minibatch_results)

Merges the explanations computed on minibatches so that the distributed explainer returns the same output as the sequential version. If the type returned by the explainer is not supported by the function, expand this function by adding an appropriately named private function and use this function to check the input type and call it.

**Parameters** minibatch_results (Union[List[ndarray], List[List[ndarray]]]) – Explanations for each minibatch.

**Return type** Union[ndarray, List[ndarray]]

**Returns**

- If the input is List[np.ndarray], a single numpy array obtained by concatenating minibatch results along
• the 0th axis. If the input is `List[List[np.ndarray]]` a list of numpy arrays obtained by concatenating arrays in with the same position in the sublists along the 0th axis.

`alibi.utils.distributed.default_target_fcn(actor, instances, kwargs=None)`
A target function that is executed in parallel given an actor pool. Its arguments must be an actor and a batch of values to be processed by the actor. Its role is to execute distributed computations when an actor is available.

Parameters

- `actor (Any)` – A ray actor. This is typically a class decorated with the `@ray.remote` decorator, that has been subsequently instantiated using `cls.remote(*args, **kwargs)`.
- `instances (tuple)` – A (batch_index, batch) tuple containing the batch of instances to be explained along with a batch index.
- `kwargs (Optional[Dict])` – A list of keyword arguments for the actor `get_explanation` method.

Returns
A future that can be used to later retrieve the results of a distributed computation.

Notes
This function can be customized (e.g., if one does not desire to wrap the explainer such that it has `get_explanation` method. The customized function should be called `_target_fcn` with the wildcard being replaced by the name of the explanation method (e.g., cem, cfproto, etc). The same name should be added to the `distributed_opts` dictionary passed by the user prior to instantiating the `DistributedExplainer`.

`alibi.utils.distributed.invert_permutation(p)`
Inverts a permutation.

Parameters `p (list)` – Some permutation of 0, 1, ..., len(p)-1. Returns an array s, where s[i] gives the index of i in p.

Return type ndarray

Returns `s` – `s[i]` gives the index of `i` in `p`.

`alibi.utils.distributed.order_result (unordered_result)`
Re-orders the result of a distributed explainer so that the explanations follow the same order as the input to the explainer.

Parameters `unordered_result (Generator[Tuple[int, Any], None, None])` – Each tuple contains the batch id as the first entry and the explanations for that batch as the second.

Return type List

Returns A list with re-ordered results.
Notes

This should not be used if one wants to take advantage of the results being returned as they are calculated.

**alibi.utils.distributions module**

```python
alibi.utils.distributions.kl_bernoulli(p, q)
```

Compute KL-divergence between 2 probabilities p and q. len(p) divergences are calculated simultaneously.

**Parameters**

- `p` (ndarray) – Probability.
- `q` (ndarray) – Probability.

**Return type** ndarray

**Returns** Array with the KL-divergence between p and q.

**alibi.utils.download module**

```python
alibi.utils.download.spacy_model(model='en_core_web_md')
```

Download spaCy model.

**Parameters**

- `model` (str) – Model to be downloaded

**Return type** None

**alibi.utils.gradients module**

```python
alibi.utils.gradients.num_grad_batch(func, X, args=(), eps=1e-08)
```

Calculate the numerical gradients of a vector-valued function (typically a prediction function in classification) with respect to a batch of arrays X.

**Parameters**

- `func` (Callable) – Function to be differentiated
- `X` (ndarray) – A batch of vectors at which to evaluate the gradient of the function
- `args` (Tuple) – Any additional arguments to pass to the function
- `eps` (Union[Float, ndarray]) – Gradient step to use in the numerical calculation, can be a single float or one for each feature

**Return type** ndarray

**Returns** An array of gradients at each point in the batch X

```python
alibi.utils.gradients.perturb(X, eps=1e-08, prova=False)
```

Apply perturbation to instance or prediction probabilities. Used for numerical calculation of gradients.

**Parameters**

- `X` (ndarray) – Array to be perturbed
- `eps` (Union[Float, ndarray]) – Size of perturbation
- `proba` (bool) – If True, the net effect of the perturbation needs to be 0 to keep the sum of the probabilities equal to 1
Return type: Tuple[ndarray, ndarray]

Returns: Instances where a positive and negative perturbation is applied.

**alibi.utils.mapping module**

**alibi.utils.mapping.num_to_ord(data, dist)**

Transform numerical values into categories using the map calculated under the fit method.

Parameters:

- *data* (ndarray): Numpy array with the numerical data.
- *dist* (dict): Dict with as keys the categorical variables and as values the numerical value for each category.

Return type: ndarray

Returns: Numpy array with transformed numerical data into categories.

**alibi.utils.mapping.ohe_to_ord(X_ohe, cat_vars_ohe)**

Convert one-hot encoded variables to ordinal encodings.

Parameters:

- *X_ohe* (ndarray): Data with mixture of one-hot encoded and numerical variables.
- *cat_vars_ohe* (dict): Dict with as keys the first column index for each one-hot encoded categorical variable and as values the number of categories per categorical variable.

Return type: Tuple[ndarray, dict]

Returns: Ordinal equivalent of one-hot encoded data and dict with categorical columns and number of categories.

**alibi.utils.mapping.ohe_to_ord_shape(shape, cat_vars=None, is_ohe=False)**

Infer shape of instance if the categorical variables have ordinal instead of on-hot encoding.

Parameters:

- *shape* (tuple): Instance shape, starting with batch dimension.
- *cat_vars* (Optional[dict]): Dict with as keys the categorical columns and as values the number of categories per categorical variable.
- *is_ohe* (bool): Whether instance is OHE.

Return type: tuple

Returns: Tuple with shape of instance with ordinal encoding of categorical variables.

**alibi.utils.mapping.ord_to_num(data, dist)**

Transform categorical into numerical values using a mapping.

Parameters:

- *data* (ndarray): Numpy array with the categorical data.
- *dist* (dict): Dict with as keys the categorical variables and as values the numerical value for each category.

Return type: ndarray

Returns: Numpy array with transformed categorical data into numerical values.
**alibi.utils.mapping.ord_to_ohe** *(X_ord, cat_vars_ord)*

Convert ordinal to one-hot encoded variables.

**Parameters**

- **X_ord** *(ndarray)* – Data with mixture of ordinal encoded and numerical variables.
- **cat_vars_ord** *(dict)* – Dict with as keys the categorical columns and as values the number of categories per categorical variable.

**Return type** *Tuple[ndarray, dict]*

**Returns** One-hot equivalent of ordinal encoded data and dict with categorical columns and number of categories.

**alibi.utils.tf module**

- **alibi.utils.tf.argmax_grad**(x)
- **alibi.utils.tf.argmin_grad**(x, y)
- **alibi.utils.tf.one_hot_grad**(x, y)
- **alibi.utils.tf.round_grad**(x)

**alibi.utils.visualization module**

**alibi.utils.wrappers module**

- **class alibi.utils.wrappers.ArgmaxTransformer**(predictor)

**Bases:** *object*

A transformer for converting classification output probability tensors to class labels. It assumes the predictor is a callable that can be called with a N-tensor of data points x and produces an N-tensor of outputs.

- **class alibi.utils.wrappers.Predictor**(clf, preprocessor=None)

**Bases:** *object*

**alibi.utils.wrappers.methdispatch**(func)

A decorator that is used to support singledispatch style functionality for instance methods. By default, singledispatch selects a function to call from registered based on the type of args[0]:

```python
def wrapper(*args, **kw):
    return dispatch(args[0].__class__)(*args, **kw)
```

This uses singledispatch to do achieve this but instead uses args[1] since args[0] will always be self.

### 44.1.2 Submodules

**alibi.datasets module**

- **alibi.datasets.fetch_adult**(features_drop=None, return_X_y=False, url_id=0)


**Parameters**
• **features_drop** (Optional[list]) – List of features to be dropped from dataset, by default drops [“fnlwgt”, “Education-Num”]

• **return_X_y** (bool) – If true, return features X and labels y as numpy arrays, if False return a Bunch object

• **url_id** (int) – Index specifying which URL to use for downloading

**Return type** Union[Bunch, Tuple[ndarray, ndarray]]

**Returns**

• **Bunch** – Dataset, labels, a list of features and a dictionary containing a list with the potential categories for each categorical feature where the key refers to the feature column.

• **(data, target)** – Tuple if return_X_y is true

**alibi.datasets.fetch_fashion_mnist** (return_X_y=False)

Loads the Fashion MNIST dataset.

**Parameters**

• **return_X_y** (bool) – If True, an NxMxP array of data points and N-array of labels are returned instead of a dict.

**Returns**

• If return_X_y is False, a Bunch object with fields ‘data’, ‘targets’ and ‘target_names’ is returned. Otherwise an array with data points and an array of labels is returned.

**alibi.datasets.fetch_imagenet** (category='Persian cat', nb_images=10, target_size=(299, 299), min_std=10.0, seed=42, return_X_y=False)

Retrieve imagenet images from specified category which needs to be in the mapping dictionary.

**Parameters**

• **category** (str) – Imagenet category class name. Must be one of keys present in alibi/data/imagenet_class_names_to_id.json

• **nb_images** (int) – Number of images to be retrieved

• **target_size** (tuple) – Size of the returned images

• **min_std** (float) – Min standard deviation of image pixels. Images that are no longer available can be returned without content which is undesirable. Having a min std cutoff resolves this.

• **seed** (int) – Random seed

• **return_X_y** (bool) – If true, return features X and labels y as numpy arrays, if False return a Bunch object

**Return type** Union[Bunch, Tuple[ndarray, ndarray]]

**Returns**

• **Bunch** – List with images and the labels from imagenet.

• **(data, target)** – Tuple if return_X_y is true

**alibi.datasets.fetch_movie_sentiment** (return_X_y=False, url_id=0)

The movie review dataset, equally split between negative and positive reviews.

**Parameters**

• **return_X_y** (bool) – If true, return features X and labels y as Python lists, if False return a Bunch object

• **url_id** (int) – Index specifying which URL to use for downloading
Return type: `Union[Bunch, Tuple[list, list]]`

Returns:
- `Bunch` – Movie reviews and sentiment labels (0 means ‘negative’ and 1 means ‘positive’).
- `(data, target)` – Tuple if `return_X_y` is true
• genindex
• modindex
• search
a
alibi, 367
alibi.api, 367
alibi.api.defaults, 367
alibi.api.interfaces, 368
alibi.confidence, 369
alibi.confidence.model_linearity, 372
alibi.confidence.trustscore, 373
alibi.datasets, 453
alibi.explainers, 375
alibi.explainers.ale, 400
alibi.explainers.anchor_base, 403
alibi.explainers.anchor_explanation, 407
alibi.explainers.anchor_image, 409
alibi.explainers.anchor_tabular, 412
alibi.explainers.anchor_text, 418
alibi.explainers.cem, 423
alibi.explainers.cfproto, 425
alibi.explainers.counterfactual, 429
alibi.explainers.integrated_gradients, 430
alibi.explainers.shap_wrappers, 431
alibi.tests, 439
alibi.tests.utils, 439
alibi.utils, 440
alibi.utils.approximation_methods, 440
alibi.utils.data, 441
alibi.utils.discretizer, 441
alibi.utils.distance, 442
alibi.utils.distributed, 443
alibi.utils.distributions, 451
alibi.utils.download, 451
alibi.utils.gradients, 451
alibi.utils.mapping, 452
alibi.utils.tf, 453
alibi.utils.wrappers, 453
alibi.version, 455
Symbols

Symbol

obby.api.interfaces.Explanation method, 368
oscope
obby.explainers.anchor_tabular.RemoteSampler method, 414
roboscopes
obby.explainers.anchor_tabular.TabularSampler method, 415
oscopes
obby.utils.distributed.DistributedExplainer method, 446
oscopes
obby.utils.distributed.PoolCollection method, 448
oscopes
obby.api.interfaces.Explanation method, 368
oscopes
obby.confidence.LinearityMeasure method, 370
oscopes
obby.confidence.TrustScore method, 370
oscopes
obby.confidence.model_linearity.LinearityMeasure method, 372
oscopes
obby.confidence.trustscore.TrustScore method, 373
oscopes
obby.explainers.ALE method, 375
oscopes
obby.explainers.AnchorImage method, 382
oscopes
obby.explainers.AnchorTabular method, 376
oscopes
obby.explainers.CEM method, 385
oscopes
obby.explainers.CounterFactual method, 387
oscopes
obby.explainers.CounterFactualProto method, 389
oscopes
obby.explainers.IntegratedGradients method, 398
oscopes
obby.explainers.KernelShap method, 392
oscopes
obby.explainers.TreeShap method, 395
oscopes
obby.explainers.ale.ALE method, 400
oscopes
obby.explainers.anchor_base.AnchorBaseBeam method, 403
oscopes
obby.explainers.anchor_explanation.AnchorExplanation method, 407
oscopes
obby.explainers.anchor_image.AnchorImage method, 409
oscopes
obby.explainers.anchor_tabular.AnchorTabular method, 412
oscopes
obby.explainers.anchor_tabular.TabularSampler method, 415
oscopes
obby.explainers.anchor_text.AnchorText method, 418
oscopes
obby.explainers.anchor_text.Neighbors method, 422
oscopes
obby.explainers.cem.CEM method, 423
oscopes
obby.explainers.cfproto.CounterFactualProto method, 425
oscopes
obby.explainers.counterfactual.CounterFactual method, 429
oscopes
obby.explainers.integrated_gradients.IntegratedGradients method, 430
oscopes
obby.explainers.shap_wrappers.KernelExplainerWrapper method, 432
oscopes
obby.explainers.shap_wrappers.KernelShap method, 432
oscopes
obby.explainers.shap_wrappers.TreeShap method, 435
oscopes
obby.tests.utils.MockPredictor method, 439
oscopes
obby.utils.discretizer.Discretizer method, 441
oscopes
obby.utils.distributed.ActorPool method, 443
oscopes
obby.utils.distributed.DistributedExplainer method, 446
oscopes
obby.utils.distributed.PoolCollection method, 448

A

obby.utils.distance.abdm() (in module alibi.utils.distance), 42
obby.utils.distributed.ActorPool (class in alibi.utils.distributed), 443
obby.utils.distributed.DistributedExplainer (property), 447
build_explanation() (alibi.explainers.integrated_gradients.IntegratedGradients method), 431
build_explanation() (alibi.explainers.IntegratedGradients method), 399
build_explanation() (alibi.explainers.KernelShap method), 393
build_explanation() (alibi.explainers.shap_wrappers.KernelShap method), 433
build_explanation() (alibi.explainers.shap_wrappers.TreeShap method), 436
build_explanation() (alibi.explainers.TreeShap method), 436
build_explanation() (alibi.explainers.TreeShap method), 396
build_lookups() (alibi.explainers.anchor_tabular.RemoteSampler method), 414
build_lookups() (alibi.explainers.anchor_tabular.TabularSampler method), 416
Bunch (class in alibi.utils.data), 441

C

CEM (class in alibi.explainers), 385
CEM (class in alibi.explainers), 423
check_ray() (in module alibi.utils.distributed), 449
cityblock_batch() (in module alibi.utils.distance), 442
compare_labels() (alibi.explainers.anchor_image.AnchorImage method), 409
compare_labels() (alibi.explainers.anchor_tabular.TabularSampler method), 416
compare_labels() (alibi.explainers.anchor_text.AnchorText method), 419
compare_labels() (alibi.explainers.AnchorImage method), 383
compare_labels() (alibi.explainers.AnchorText method), 379
compute_beta() (alibi.explainers.anchor_base.AnchorBaseBeam static method), 404
concatenate_minibatches() (in module alibi.utils.distributed), 449
CounterFactual (class in alibi.explainers), 387
CounterFactual (class in alibi.explainers), 429
CounterFactualProto (class in alibi.explainers.cfpproto), 425
coverage() (alibi.explainers.anchor_explanation.AnchorExplanation method), 408
create_explainer_handles() (alibi.utils.distributed.PoolCollection static method), 448
create_parallel_pool() (alibi.utils.distributed.DistributedExplainer method), 447

D
default() (alibi.api.interfaces.NumpyEncoder method), 369
DEFAULT_DATA_ALE (in module alibi.api.defaults), 367
DEFAULT_DATA_ANCHOR (in module alibi.api.defaults), 367
DEFAULT_DATA_ANCHOR_IMG (in module alibi.api.defaults), 367
DEFAULT_DATA_CEM (in module alibi.api.defaults), 367
DEFAULT_DATA_CF (in module alibi.api.defaults), 367
DEFAULT_DATA_CFP (in module alibi.api.defaults), 367
DEFAULT_DATA_INTGRAD (in module alibi.api.defaults), 367
DEFAULT_DATA_KERNEL_SHAP (in module alibi.api.defaults), 367
DEFAULT_DATA_TREE_SHAP (in module alibi.api.defaults), 367
default_meta() (in module alibi.api.interfaces), 369
DEFAULT_META_ALE (in module alibi.api.defaults), 367
DEFAULT_META_ANCHOR (in module alibi.api.defaults), 367
DEFAULT_META_CEM (in module alibi.api.defaults), 367
DEFAULT_META_CF (in module alibi.api.defaults), 368
DEFAULT_META_CFP (in module alibi.api.defaults), 368
DEFAULT_META_INTGRAD (in module alibi.api.defaults), 368
DEFAULT_META_KERNEL_SHAP (in module alibi.api.defaults), 368
DEFAULT_META_TREE_SHAP (in module alibi.api.defaults), 368
default_target_fcn() (in module alibi.utils.distributed), 450
deferred_init() (alibi.explainers.anchor_tabular.TabularSampler method), 416
discretize() (alibi.utils.discretizer.Discretizer method), 442
DistributedAnchorBaseBeam (class in alibi.explainers.anchor_base), 407
DistributedAnchorTabular (class in alibi.explainers.anchor_tabular), 378
DistributedAnchorTabular (class in alibi.explainers.anchor_tabular), 414
DistributedExplainer (class in alibi.utils.distributed), 446
dlow_bernoulli () (alibi.explainers.anchor_base.AnchorBaseBeam static method), 404
draw_samples () (alibi.explainers.anchor_base.AnchorBaseBeam method), 404
draw_samples () (alibi.explainers.anchor_base.DistributedAnchorBaseBeam method), 407
dup_bernoulli () (alibi.explainers.anchor_base.AnchorBaseBeam static method), 405
examples () (alibi.explainers.anchor_explanation.AnchorExplanation method), 408
explain () (alibi.api.interfaces.Explainer method), 368
explain () (alibi.explainers.ALE method), 376
explain () (alibi.explainers.ale.ALE method), 401
explain () (alibi.explainers.anchor_image.AnchorImage method), 409
explain () (alibi.explainers.anchor_tabular.AnchorTabular method), 412
explain () (alibi.explainers.anchor_tabular.DistributedAnchorTabular method), 414
explain () (alibi.explainers.anchor_text.AnchorText method), 419
explain () (alibi.explainers.AnchorImage method), 383
explain () (alibi.explainers.AnchorTabular method), 377
explain () (alibi.explainers.AnchorText method), 379
explain () (alibi.explainers.CEM method), 386
explain () (alibi.explainers.cem.CEM method), 424
explain () (alibi.explainers.cfproto.CounterFactualProto method), 427
explain () (alibi.explainers.CounterFactual method), 388
explain () (alibi.explainers.counterfactual.CounterFactual method), 430
explain () (alibi.explainers.CounterFactualProto method), 390
explain () (alibi.explainers.DistributedAnchorTabular method), 378
explain () (alibi.explainers.integrated_gradients.IntegratedGradients method), 431
explain () (alibi.explainers.IntegratedGradients method), 399
explain () (alibi.explainers.shap_wrappers.KernelShap method), 393
explain () (alibi.explainers.shap_wrappers.TreeShap method), 437
explain () (alibi.explainers.TreeShap method), 397
Explaner (class in alibi.api.interfaces), 368
Explanation (class in alibi.api.interfaces), 368
fit()  (alibi.explainers.cfproto.CounterFactualProto method), 427
fit()  (alibi.explainers.CounterFactual method), 388
fit()  (alibi.explainers.counterfactual.CounterFactual method), 430
fit()  (alibi.explainers.CounterFactualProto method), 391
fit()  (alibi.explainers.DistributedAnchorTabular method), 378
fit()  (alibi.explainers.KernelShap method), 394
fit()  (alibi.explainers.shap_wrappers.KernelShap method), 434
fit()  (alibi.explainers.shap_wrappers.TreeShap method), 437
fit()  (alibi.explainers.TreeShap method), 397
FitMixin  (class in alibi.api.interfaces), 369
from_json()  (alibi.api.interfaces.Explanation class method), 368

gauss_legendre_builders() (in module alibi.utils.approximation_methods), 440
gen_category_map() (in module alibi.utils.data), 441
generate_superpixels()  (alibi.explainers.anchor_image.AnchorImage method), 410
generate_superpixels()  (alibi.explainers.AnchorImage method), 384
get_anchor_metadata()  (alibi.explainers.anchor_base.AnchorBaseBeam method), 405
get_explanation()  (alibi.explainers.shap_wrappers.KernelExplainerWrapper method), 432
get_explanation()  (alibi.utils.distributed.DistributedExplainer method), 447
get_explanation()  (alibi.utils.distributed.PoolCollection method), 448
get_features_index()  (alibi.explainers.anchor_tabular.TabularSampler method), 417
get_gradients()  (alibi.explainers.CEM method), 386
get_gradients()  (alibi.explainers.CEM method), 424
get_gradients()  (alibi.explainers.cfproto.CounterFactualProto method), 428
get_gradients()  (alibi.explainers.CounterFactualProto method), 392
get_init_stats()  (alibi.explainers.anchor_base.AnchorBaseBeam method), 405
get_next()  (alibi.utils.distributed.ActorPool method), 444
get_next_unordered()  (alibi.utils.distributed.ActorPool method), 444
get_percentiles()  (alibi.utils.discretizer.Discretizer static method), 442
get_quantiles()  (in module alibi.explainers.ale), 402

H
handle_unk_features()  (alibi.explainers.anchor_tabular.TabularSampler method), 417
has_next()  (alibi.utils.distributed.ActorPool method), 444

I
IntegratedGradients  (class in alibi.explainers), 398
IntegratedGradients  (class in alibi.explainers.integrated_gradients), 430
invert_permutation()  (in module alibi.utils.distributed), 450
issorted()  (in module alibi.tests.utils), 439

K
KERNEL_SHAP_PARAMS (in module alibi.api.defaults), 368
KernelExplainerWrapper  (class in alibi.explainers.shap_wrappers), 432
KernelShap  (class in alibi.explainers), 392
KernelShap  (class in alibi.explainers.shap_wrappers), 432
kl_bernoulli()  (in module alibi.utils.distributions), 451
klucb()  (alibi.explainers.anchor_base.AnchorBaseBeam method), 405

L
left  (alibi.utils.approximation_methods.Riemann attribute), 440
linearity_measure()  (in module alibi.confidence), 369
linearity_measure()  (in module alibi.confidence.model_linearity), 373
LinearityMeasure  (class in alibi.confidence), 370
LinearityMeasure  (class in alibi.confidence.model_linearity), 372
loss_fn()  (alibi.explainers.CEM method), 387
loss_fn()  (alibi.explainers.cem.CEM method), 424
loss_fn() (alibi.explainers.cfproto.CounterFactualProto method), 428
loss_fn() (alibi.explainers.CounterFactualProto method), 392

prefix function

M
map() (alibi.utils.distributed.ActorPool method), 444
map_unordered() (alibi.utils.distributed.ActorPool method), 445
methdispatch() (in module alibi.utils.wrappers), 453
middle (alibi.utils.approximation_methods.Riemann attribute), 440
minimum_satisfied() (in module alibi.explainers.ale), 402
MockPredictor (class in alibi.tests.utils), 439
multidim_scaling() (in module alibi.utils.distance), 442
mvdm() (in module alibi.utils.distance), 443

N
names() (alibi.explainers.anchor_explanation.AnchorExplanation method), 408
Neighbors (class in alibi.explainers.anchor_text), 422
neighbors() (alibi.explainers.anchor_text.Neighbors method), 422
not_raises() (in module alibi.tests.utils), 439
num_grad_batch() (in module alibi.utils.gradients), 451
num_to_ord() (in module alibi.utils.mapping), 452
NumpyEncoder (class in alibi.api.interfaces), 369

O
ohe_to_ord() (in module alibi.utils.mapping), 452
ohe_to_ord_shape() (in module alibi.utils.mapping), 452
one_hot_grad() (in module alibi.utils.tf), 453
ord_to_num() (in module alibi.utils.mapping), 452
ord_to_ohe() (in module alibi.utils.mapping), 452
order_result() (in module alibi.utils.distributed), 450
overlay_mask() (alibi.explainers.anchor_image.AnchorImage method), 410
overlay_mask() (alibi.explainers.AnchorImage method), 384

P
perturb() (alibi.explainers.CEM method), 387
perturb() (alibi.explainers.cem.CEM method), 425
perturb() (in module alibi.utils.gradients), 451
perturb_sentence() (alibi.explainers.AnchorText method), 420
perturbation() (alibi.explainers.anchor_image.AnchorImage method), 411
perturbation() (alibi.explainers.anchor_tabular.TabularSampler method), 417
perturbation() (alibi.explainers.AnchorImage method), 384
plot_ale() (in module alibi.explainers), 398
PoolCollection (class in alibi.utils.distributed), 448
precision() (alibi.explainers.anchor_explanation.AnchorExplanation method), 408
predict() (alibi.tests.utils.MockPredictor method), 439
Predictor (class in alibi.utils.wrappers), 453
propose_anchors() (alibi.explainers.anchor_base.AnchorBaseBeam method), 406

R
rank_by_importance() (in module alibi.explainers.shap_wrappers), 438
remote_explainer_index() (alibi.utils.distributed.PoolCollection property), 449
RemoteSampler (class in alibi.explainers.anchor_tabular), 414
replace_features() (alibi.explainers.anchor_tabular.TabularSampler method), 417
ResourceError, 449
return_attribute() (alibi.explainers.shap_wrappers.KernelExplainerWrapper method), 432
return_attribute() (alibi.utils.distributed.DistributedExplainer method), 448
Riemann (class in alibi.utils.approximation_methods), 440
riemann_builders() (in module alibi.utils.approximation_methods), 440
right (alibi.utils.approximation_methods.Riemann attribute), 440
round_grad() (in module alibi.utils.tf), 453

S
sampler() (alibi.explainers.anchor_image.AnchorImage method), 411
sampler() (alibi.explainers.anchor_text.AnchorText method), 421

466 Index
sampler() (alibi.explainers.AnchorImage method), 384
sampler() (alibi.explainers.AnchorText method), 381
score() (alibi.confidence.LinearityMeasure method), 370
score() (alibi.confidence.model_linearity.LinearityMeasure method), 372
score() (alibi.confidence.TrustScore method), 371
score() (alibi.explainers.CounterFactualProto method), 392
select_critical_arms() (alibi.explainers.anchor_base.AnchorBaseBeam method), 406
set_actor_index() (alibi.utils.distributed.DistributedExplainer method), 448
set_data_type() (alibi.explainers.anchor_text.AnchorText method), 421
set_data_type() (alibi.explainers.AnchorText method), 381
set_instance_label() (alibi.explainers.anchor_tabular.RemoteSampler method), 414
set_instance_label() (alibi.explainers.anchor_tabular.TabularSampler method), 418
set_n_covered() (alibi.explainers.anchor_tabular.RemoteSampler method), 414
set_n_covered() (alibi.explainers.anchor_tabular.TabularSampler method), 418
set_sampler_perturbation() (alibi.explainers.anchor_text.AnchorText method), 421
set_sampler_perturbation() (alibi.explainers.AnchorText method), 381
set_words_and_pos() (alibi.explainers.AnchorText method), 422
set_words_and_pos() (alibi.explainers.shap_wrappers), 438
spacy_model() (alibi.utils.download), 451
submit() (alibi.utils.distributed.ActorPool method), 445
sum_categories() (in module alibi.explainers.shap_wrappers), 438

T
TabularSampler (class in alibi.explainers.anchor_tabular), 415
to_json() (alibi.api.interfaces.Explanation method), 368
to_sample() (alibi.explainers.anchor_base.AnchorBaseBeam static method), 406
trapezoid(alibi.utils.approximation_methods.Riemann attribute), 440
TREE_SHAP_PARAMS (in module alibi.api.defaults), 368
TreeShap (class in alibi.explainers), 395
TreeShap (class in alibi.explainers.shap_wrappers), 435
TrustScore (class in alibi.confidence), 370
TrustScore (class in alibi.confidence.trustscore), 373

U
UNK (alibi.explainers.anchor_text.AnchorText attribute), 418
UNK (alibi.explainers.AnchorText attribute), 378
update_state() (alibi.explainers.anchor_base.AnchorBaseBeam method), 407