alibi-detect Documentation

Release 0.6.1dev

Seldon Technologies Ltd

Apr 23, 2021

OVERVIEW

1 Installation 3

2 Features 5

3 Basic Usage 7

4 Algorithm Overview 9

5 Roadmap 11

6 Mahalanobis Distance 13

7 Isolation Forest 17

8 Variational Auto-Encoder 19

9 Auto-Encoder 23

10 Variational Auto-Encoding Gaussian Mixture Model 27

11 Auto-Encoding Gaussian Mixture Model 31

12 Likelihood Ratios for Outlier Detection 35

13 Prophet Detector 39

14 Spectral Residual 43

15 Sequence-to-Sequence (Seq2Seq) 45

16 Adversarial Auto-Encoder 49

17 Model distillation 55

18 Chi-Squared 59

19 Classifier 63

20 Kolmogorov-Smirnov 67

21 Maximum Mean Discrepancy 71

22 Mixed-type tabular data 75

23 Model Uncertainty 79

24 Outlier, adversarial and drift detection on CIFAR10 83

25 Mahalanobis outlier detection on KDD Cup '99 dataset 115

26 Isolation Forest outlier detection on KDD Cup '99 dataset 127

27 VAE outlier detection on KDD Cup '99 dataset 133

28 VAE outlier detection on CIFAR10 141

29 AE outlier detection on CIFAR10 153

30 AEGMM and VAEGMM outlier detection on KDD Cup '99 dataset 165

31 Likelihood Ratio Outlier Detection with PixelCNN++ 177

32 Likelihood Ratio Outlier Detection on Genomic Sequences 191

33 Time-series outlier detection using Prophet on weather data 199

34 Time series outlier detection with Spectral Residuals on synthetic data 207

35 Time series outlier detection with Seq2Seq models on synthetic data 213

36 Seq2Seq time series outlier detection on ECG data 221

37 Adversarial AE detection and correction on CIFAR-10 229

38 Model Distillation drift detector on CIFAR-10 251

39 Categorical and mixed type data drift detection on income prediction 265

40 Kolmogorov-Smirnov data drift detector on CIFAR-10 273

41 Maximum Mean Discrepancy drift detector on CIFAR-10 291

42 Text drift detection on IMDB movie reviews 301

43 Classifier drift detector on CIFAR-10 321

44 Model uncertainty based drift detection on CIFAR-10 and Wine-Quality datasets 329

45 Overview 341

46 Overview 343

47 alibi_detect 345

48 Indices and tables 405

Python Module Index 407

Index 409

Alibi Detect is an open source Python library focused on **outlier**, **adversarial** and **drift** detection. The package aims to cover both online and offline detectors for tabular data, text, images and time series. Both **TensorFlow** and **PyTorch** backends are supported for drift detection.

For more background on the importance of monitoring outliers and distributions in a production setting, check out this talk from the *Challenges in Deploying and Monitoring Machine Learning Systems* ICML 2020 workshop, based on the paper Monitoring and explainability of models in production and referencing Alibi Detect.

CHAPTER

ONE

INSTALLATION

alibi-detect can be installed from PyPI:

```
pip install alibi-detect
```

CHAPTER

TWO

FEATURES

Alibi Detect is an open source Python library focused on **outlier**, **adversarial** and **drift** detection. The package aims to cover both online and offline detectors for tabular data, text, images and time series. Both **TensorFlow** and **PyTorch** backends are supported for drift detection. Alibi Detect does however not install PyTorch for you. Check the PyTorch docs how to do this.

To get a list of respectively the latest outlier, adversarial and drift detection algorithms, you can type:

```
import alibi_detect
alibi_detect.od.__all__
```

```
['OutlierAEGMM',
 'IForest',
 'Mahalanobis',
 'OutlierAE',
 'OutlierVAE',
 'OutlierVAEGMM',
 'OutlierProphet',  # requires prophet: pip install alibi-detect[prophet]
 'OutlierSeq2Seq',
 'SpectralResidual',
 'LLR']
```

```
alibi_detect.ad.__all__
```

```
['AdversarialAE',
'ModelDistillation']
```

```
alibi_detect.cd.__all__
```

```
['ChiSquareDrift',
 'ClassifierDrift',
 'ClassifierUncertaintyDrift',
 'KSDrift',
 'MMDDrift',
 'RegressorUncertaintyDrift',
 'TabularDrift']
```

Summary tables highlighting the practical use cases for all the algorithms can be found *here*.

For detailed information on the **outlier detectors**:

- *Isolation Forest*
- *Mahalanobis Distance*

- *Auto-Encoder (AE)*
- *Variational Auto-Encoder (VAE)*
- *Auto-Encoding Gaussian Mixture Model (AEGMM)*
- *Variational Auto-Encoding Gaussian Mixture Model (VAEGMM)*
- *Likelihood Ratios*
- *Prophet Detector*
- *Spectral Residual Detector*
- *Sequence-to-Sequence (Seq2Seq) Detector*

Similar for **adversarial detection**:

- *Adversarial AE Detector*
- *Model Distillation Detector*

And **data drift**:

- *Kolmogorov-Smirnov Drift Detector*
- *Maximum Mean Discrepancy Drift Detector*
- *Chi-Squared Drift Detector*
- *Mixed-type Tabular Data Drift Detector*
- *Classifier Drift Detector*
- *Classifier and Regressor Drift Detectors*

CHAPTER

THREE

BASIC USAGE

We will use the *VAE outlier detector* to illustrate the usage of outlier and adversarial detectors in alibi-detect.

First, we import the detector:

```
from alibi_detect.od import OutlierVAE
```

Then we initialize it by passing it the necessary arguments:

```
od = OutlierVAE(
    threshold=0.1,
    encoder_net=encoder_net,
    decoder_net=decoder_net,
    latent_dim=1024
)
```

Some detectors require an additional `.fit` step using training data:

```
od.fit(X_train)
```

The detectors can be saved or loaded as follows:

```
from alibi_detect.utils.saving import save_detector, load_detector

filepath = './my_detector/'
save_detector(od, filepath)
od = load_detector(filepath)
```

Finally, we can make predictions on test data and detect outliers or adversarial examples.

```
preds = od.predict(X_test)
```

The predictions are returned in a dictionary with as keys `meta` and `data`. `meta` contains the detector's metadata while `data` is in itself a dictionary with the actual predictions. It has either `is_outlier`, `is_adversarial` or `is_drift` (filled with 0's and 1's) as well as optional `instance_score`, `feature_score` or `p_value` as keys with numpy arrays as values.

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-detect.

CHAPTER

FOUR

ALGORITHM OVERVIEW

The following tables summarize the advised use cases for the current algorithms. Please consult the method specific pages for a more detailed breakdown of each method. The column *Feature Level* indicates whether the outlier scoring and detection can be done and returned at the feature level, e.g. per pixel for an image.

4.1 Outlier Detection

Detector	Tabular	Image	Time Series	Text	Categorical Features	Online	Feature Level
Isolation Forest	✓				✓		
Mahalanobis Distance	✓				✓	✓	
AE	✓	✓					✓
VAE	✓	✓					✓
AEGMM	✓	✓					
VAEGMM	✓	✓					
Likelihood Ratios	✓	✓	✓		✓		✓
Prophet			✓				
Spectral Residual			✓			✓	✓
Seq2Seq			✓				✓

4.2 Adversarial Detection

Detector	Tabular	Image	Time Series	Text	Categorical Features	Online	Feature Level
Adversarial AE	✓	✓					
Model distillation	✓	✓	✓	✓	✓		

4.3 Drift Detection

Detector	Tabular	Image	Time Series	Text	Categorical Features	Online	Feature Level
Kolmogorov-Smirnov	✓	✓		✓	✓		✓
Maximum Mean Discrepancy	✓	✓		✓	✓		
Chi-Squared	✓				✓		✓
Mixed-type tabular	✓				✓		✓
Classifier	✓	✓	✓	✓	✓		
Classifier Uncertainty	✓	✓	✓	✓	✓		
Regressor Uncertainty	✓	✓	✓	✓	✓		

All drift detectors and built-in preprocessing methods support both **PyTorch** and **TensorFlow** backends. The preprocessing steps include randomly initialized encoders, pretrained text embeddings to detect drift on using the transformers library and extraction of hidden layers from machine learning models. The preprocessing steps allow to detect different types of drift such as covariate and predicted distribution shift.

CHAPTER

FIVE

ROADMAP

Alibi Detect aims to be the go-to library for **outlier**, **adversarial** and **drift** detection in Python using both the **TensorFlow** and **PyTorch** backends.

This means that the algorithms in the library need to handle:

- **Online** detection with often stateful detectors.
- **Offline** detection, where the detector is trained on a batch of unsupervised or semi-supervised data. This assumption resembles a lot of real-world settings where labels are hard to come by.

The algorithms will cover the following data types:

- **Tabular**, including both numerical and categorical data.
- **Images**
- **Time series**, both univariate and multivariate.
- **Text**

It will also be possible to combine different algorithms in ensemble detectors.

The library **currently** covers both online and offline outlier detection algorithms for tabular data, images and time series as well as offline adversarial detectors for tabular data and images. Current drift detection capabilities cover mixed type tabular data, text and images.

The **near term** focus will be on adding online and text drift detectors, extending the PyTorch support, and adding outlier detectors for text and mixed data types.

In the **medium term**, we intend to leverage labels in a semi-supervised setting for the detectors and incorporate drift detection for time series.

source

CHAPTER

SIX

MAHALANOBIS DISTANCE

6.1 Overview

The Mahalanobis online outlier detector aims to predict anomalies in tabular data. The algorithm calculates an outlier score, which is a measure of distance from the center of the features distribution (Mahalanobis distance). If this outlier score is higher than a user-defined threshold, the observation is flagged as an outlier. The algorithm is online, which means that it starts without knowledge about the distribution of the features and learns as requests arrive. Consequently you should expect the output to be bad at the start and to improve over time. The algorithm is suitable for low to medium dimensional tabular data.

The algorithm is also able to include categorical variables. The `fit` step first computes pairwise distances between the categories of each categorical variable. The pairwise distances are based on either the model predictions (*MVDM method*) or the context provided by the other variables in the dataset (*ABDM method*). 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.

6.2 Usage

6.2.1 Initialize

Parameters:

- `threshold`: Mahalanobis distance threshold above which the instance is flagged as an outlier.
- `n_components`: number of principal components used.
- `std_clip`: feature-wise standard deviation used to clip the observations before updating the mean and covariance matrix.
- `start_clip`: number of observations before clipping is applied.
- `max_n`: algorithm behaves as if it has seen at most `max_n` points.
- `cat_vars`: dictionary with as keys the categorical columns and as values the number of categories per categorical variable. Only needed if categorical variables are present.
- `ohe`: boolean whether the categorical variables are one-hot encoded (OHE) or not. If not OHE, they are assumed to have ordinal encodings.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'* or *'image'*.

Initialized outlier detector example:

```python
from alibi_detect.od import Mahalanobis

od = Mahalanobis(
    threshold=10.,
    n_components=2,
    std_clip=3,
    start_clip=100
)
```

6.2.2 Fit

We only need to fit the outlier detector if there are categorical variables present in the data. The following parameters can be specified:

- `X`: training batch as a numpy array.
- `y`: model class predictions or ground truth labels for `X`. Used for *'mvdm'* and *'abdm-mvdm'* pairwise distance metrics. Not needed for *'abdm'*.
- `d_type`: 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`: weight on *'abdm'* (between 0. and 1.) distance if `d_type` equals *'abdm-mvdm'*.
- `disc_perc`: list with percentiles used in binning of numerical features used for the *'abdm'* and *'abdm-mvdm'* pairwise distance measures.
- `standardize_cat_vars`: standardize numerical values of categorical variables if True.
- `feature_range`: tuple with min and max ranges to allow for numerical values of categorical variables. Min and max ranges can be floats or numpy arrays with dimension *(1, number of features)* for feature-wise ranges.
- `smooth`: smoothing exponent between 0 and 1 for the distances. Lower values will smooth the difference in distance metric between different features.
- `center`: 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.

```python
od.fit(
    X_train,
    d_type='abdm',
    disc_perc=[25, 50, 75]
)
```

It is often hard to find a good threshold value. If we have a batch of normal and outlier data and we know approximately the percentage of normal data in the batch, we can infer a suitable threshold:

```python
od.infer_threshold(
    X,
    threshold_perc=95
)
```

Beware though that the outlier detector is stateful and every call to the `score` function will update the mean and covariance matrix, even when inferring the threshold.

6.2.3 Detect

We detect outliers by simply calling `predict` on a batch of instances `X` to compute the instance level Mahalanobis distances. We can also return the instance level outlier score by setting `return_instance_score` to True.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_outlier`: boolean whether instances are above the threshold and therefore outlier instances. The array is of shape *(batch size,)*.
- `instance_score`: contains instance level scores if `return_instance_score` equals True.

```
preds = od.predict(
    X,
    return_instance_score=True
)
```

6.3 Examples

6.3.1 Tabular

Outlier detection on KDD Cup 99

source

CHAPTER

SEVEN

ISOLATION FOREST

7.1 Overview

Isolation forests (IF) are tree based models specifically used for outlier detection. The IF isolates observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. The number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node. This path length, averaged over a forest of random trees, is a measure of normality and is used to define an anomaly score. Outliers can typically be isolated quicker, leading to shorter paths. The algorithm is suitable for low to medium dimensional tabular data.

7.2 Usage

7.2.1 Initialize

Parameters:

- `threshold`: threshold value for the outlier score above which the instance is flagged as an outlier.
- `n_estimators`: number of base estimators in the ensemble. Defaults to 100.
- `max_samples`: number of samples to draw from the training data to train each base estimator. If *int*, draw `max_samples` samples. If *float*, draw `max_samples` *times number of features* samples. If *'auto'*, `max_samples` = min(256, number of samples).
- `max_features`: number of features to draw from the training data to train each base estimator. If *int*, draw `max_features` features. If float, draw `max_features` *times number of features* features.
- `bootstrap`: whether to fit individual trees on random subsets of the training data, sampled with replacement.
- `n_jobs`: number of jobs to run in parallel for `fit` and `predict`.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'* or *'image'*.

Initialized outlier detector example:

```python
from alibi_detect.od import IForest

od = IForest(
    threshold=0.,
    n_estimators=100
)
```

7.2.2 Fit

We then need to train the outlier detector. The following parameters can be specified:

- X: training batch as a numpy array.
- sample_weight: array with shape *(batch size,)* used to assign different weights to each instance during training. Defaults to *None*.

```
od.fit(
    X_train
)
```

It is often hard to find a good threshold value. If we have a batch of normal and outlier data and we know approximately the percentage of normal data in the batch, we can infer a suitable threshold:

```
od.infer_threshold(
    X,
    threshold_perc=95
)
```

7.2.3 Detect

We detect outliers by simply calling predict on a batch of instances X to compute the instance level outlier scores. We can also return the instance level outlier score by setting return_instance_score to True.

The prediction takes the form of a dictionary with meta and data keys. meta contains the detector's metadata while data is also a dictionary which contains the actual predictions stored in the following keys:

- is_outlier: boolean whether instances are above the threshold and therefore outlier instances. The array is of shape *(batch size,)*.
- instance_score: contains instance level scores if return_instance_score equals True.

```
preds = od.predict(
    X,
    return_instance_score=True
)
```

7.3 Examples

7.3.1 Tabular

Outlier detection on KDD Cup 99

source

CHAPTER

EIGHT

VARIATIONAL AUTO-ENCODER

8.1 Overview

The Variational Auto-Encoder (VAE) outlier detector is first trained on a batch of unlabeled, but normal (*inlier*) data. Unsupervised or semi-supervised training is desirable since labeled data is often scarce. The VAE detector tries to reconstruct the input it receives. If the input data cannot be reconstructed well, the reconstruction error is high and the data can be flagged as an outlier. The reconstruction error is either measured as the mean squared error (MSE) between the input and the reconstructed instance or as the probability that both the input and the reconstructed instance are generated by the same process. The algorithm is suitable for tabular and image data.

8.2 Usage

8.2.1 Initialize

Parameters:

- `threshold`: threshold value above which the instance is flagged as an outlier.
- `score_type`: scoring method used to detect outliers. Currently only the default *'mse'* supported.
- `latent_dim`: latent dimension of the VAE.
- `encoder_net`: `tf.keras.Sequential` instance containing the encoder network. Example:

```
encoder_net = tf.keras.Sequential(
  [
      InputLayer(input_shape=(32, 32, 3)),
      Conv2D(64, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(128, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(512, 4, strides=2, padding='same', activation=tf.nn.relu)
  ])
```

- `decoder_net`: `tf.keras.Sequential` instance containing the decoder network. Example:

```
decoder_net = tf.keras.Sequential(
  [
      InputLayer(input_shape=(latent_dim,)),
      Dense(4*4*128),
      Reshape(target_shape=(4, 4, 128)),
      Conv2DTranspose(256, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2DTranspose(64, 4, strides=2, padding='same', activation=tf.nn.relu),
```

(continues on next page)

(continued from previous page)

```
        Conv2DTranspose(3, 4, strides=2, padding='same', activation='sigmoid')
    ])
```

- `vae`: instead of using a separate encoder and decoder, the VAE can also be passed as a `tf.keras.Model`.
- `samples`: number of samples drawn during detection for each instance to detect.
- `beta`: weight on the KL-divergence loss term following the β-VAE framework. Default equals 1.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'* or *'image'*.

Initialized outlier detector example:

```
from alibi_detect.od import OutlierVAE

od = OutlierVAE(
    threshold=0.1,
    encoder_net=encoder_net,
    decoder_net=decoder_net,
    latent_dim=1024,
    samples=10
)
```

8.2.2 Fit

We then need to train the outlier detector. The following parameters can be specified:

- `X`: training batch as a numpy array of preferably normal data.
- `loss_fn`: loss function used for training. Defaults to the elbo loss.
- `optimizer`: optimizer used for training. Defaults to Adam with learning rate 1e-3.
- `cov_elbo`: dictionary with covariance matrix options in case the elbo loss function is used. Either use the full covariance matrix inferred from X (*dict(cov_full=None)*), only the variance (*dict(cov_diag=None)*) or a float representing the same standard deviation for each feature (e.g. *dict(sim=.05)*) which is the default.
- `epochs`: number of training epochs.
- `batch_size`: batch size used during training.
- `verbose`: boolean whether to print training progress.
- `log_metric`: additional metrics whose progress will be displayed if verbose equals True.

```
od.fit(
    X_train,
    epochs=50
)
```

It is often hard to find a good threshold value. If we have a batch of normal and outlier data and we know approximately the percentage of normal data in the batch, we can infer a suitable threshold:

```
od.infer_threshold(
    X,
    threshold_perc=95
)
```

8.2.3 Detect

We detect outliers by simply calling `predict` on a batch of instances `X`. Detection can be customized via the following parameters:

- `outlier_type`: either *'instance'* or *'feature'*. If the outlier type equals *'instance'*, the outlier score at the instance level will be used to classify the instance as an outlier or not. If *'feature'* is selected, outlier detection happens at the feature level (e.g. by pixel in images).
- `outlier_perc`: percentage of the sorted (descending) feature level outlier scores. We might for instance want to flag an image as an outlier if at least 20% of the pixel values are on average above the threshold. In this case, we set `outlier_perc` to 20. The default value is 100 (using all the features).
- `return_feature_score`: boolean whether to return the feature level outlier scores.
- `return_instance_score`: boolean whether to return the instance level outlier scores.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_outlier`: boolean whether instances or features are above the threshold and therefore outliers. If `outlier_type` equals *'instance'*, then the array is of shape *(batch size,)*. If it equals *'feature'*, then the array is of shape *(batch size, instance shape)*.
- `feature_score`: contains feature level scores if `return_feature_score` equals True.
- `instance_score`: contains instance level scores if `return_instance_score` equals True.

```
preds = od.predict(
    X,
    outlier_type='instance',
    outlier_perc=75,
    return_feature_score=True,
    return_instance_score=True
)
```

8.3 Examples

8.3.1 Image

Outlier detection on CIFAR10

8.3.2 Tabular

Outlier detection on KDD Cup 99

source

CHAPTER

NINE

AUTO-ENCODER

9.1 Overview

The Auto-Encoder (AE) outlier detector is first trained on a batch of unlabeled, but normal (inlier) data. Unsupervised training is desireable since labeled data is often scarce. The AE detector tries to reconstruct the input it receives. If the input data cannot be reconstructed well, the reconstruction error is high and the data can be flagged as an outlier. The reconstruction error is measured as the mean squared error (MSE) between the input and the reconstructed instance.

9.2 Usage

9.2.1 Initialize

Parameters:

- `threshold`: threshold value above which the instance is flagged as an outlier.
- `encoder_net`: `tf.keras.Sequential` instance containing the encoder network. Example:

```
encoder_net = tf.keras.Sequential(
  [
      InputLayer(input_shape=(32, 32, 3)),
      Conv2D(64, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(128, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(512, 4, strides=2, padding='same', activation=tf.nn.relu)
  ])
```

- `decoder_net`: `tf.keras.Sequential` instance containing the decoder network. Example:

```
decoder_net = tf.keras.Sequential(
  [
      InputLayer(input_shape=(1024,)),
      Dense(4*4*128),
      Reshape(target_shape=(4, 4, 128)),
      Conv2DTranspose(256, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2DTranspose(64, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2DTranspose(3, 4, strides=2, padding='same', activation='sigmoid')
  ])
```

- `ae`: instead of using a separate encoder and decoder, the AE can also be passed as a `tf.keras.Model`.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'* or *'image'*.

Initialized outlier detector example:

```
from alibi_detect.od import OutlierAE

od = OutlierAE(threshold=0.1,
               encoder_net=encoder_net,
               decoder_net=decoder_net)
```

9.2.2 Fit

We then need to train the outlier detector. The following parameters can be specified:

- `X`: training batch as a numpy array of preferably normal data.
- `loss_fn`: loss function used for training. Defaults to the Mean Squared Error loss.
- `optimizer`: optimizer used for training. Defaults to Adam with learning rate 1e-3.
- `epochs`: number of training epochs.
- `batch_size`: batch size used during training.
- `verbose`: boolean whether to print training progress.
- `log_metric`: additional metrics whose progress will be displayed if verbose equals True.

```
od.fit(X_train, epochs=50)
```

It is often hard to find a good threshold value. If we have a batch of normal and outlier data and we know approximately the percentage of normal data in the batch, we can infer a suitable threshold:

```
od.infer_threshold(X, threshold_perc=95)
```

9.2.3 Detect

We detect outliers by simply calling `predict` on a batch of instances `X`. Detection can be customized via the following parameters:

- `outlier_type`: either *'instance'* or *'feature'*. If the outlier type equals *'instance'*, the outlier score at the instance level will be used to classify the instance as an outlier or not. If *'feature'* is selected, outlier detection happens at the feature level (e.g. by pixel in images).
- `outlier_perc`: percentage of the sorted (descending) feature level outlier scores. We might for instance want to flag an image as an outlier if at least 20% of the pixel values are on average above the threshold. In this case, we set `outlier_perc` to 20. The default value is 100 (using all the features).
- `return_feature_score`: boolean whether to return the feature level outlier scores.
- `return_instance_score`: boolean whether to return the instance level outlier scores.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_outlier`: boolean whether instances or features are above the threshold and therefore outliers. If `outlier_type` equals *'instance'*, then the array is of shape *(batch size,)*. If it equals *'feature'*, then the array is of shape *(batch size, instance shape)*.
- `feature_score`: contains feature level scores if `return_feature_score` equals True.
- `instance_score`: contains instance level scores if `return_instance_score` equals True.

```
preds = od.predict(X,
                   outlier_type='instance',
                   outlier_perc=75,
                   return_feature_score=True,
                   return_instance_score=True)
```

9.3 Examples

9.3.1 Image

Outlier detection on CIFAR10

source

CHAPTER

TEN

VARIATIONAL AUTO-ENCODING GAUSSIAN MIXTURE MODEL

10.1 Overview

The Variational Auto-Encoding Gaussian Mixture Model (VAEGMM) Outlier Detector follows the Deep Autoencoding Gaussian Mixture Model for Unsupervised Anomaly Detection paper but with a VAE instead of a regular Auto-Encoder. The encoder compresses the data while the reconstructed instances generated by the decoder are used to create additional features based on the reconstruction error between the input and the reconstructions. These features are combined with encodings and fed into a Gaussian Mixture Model (GMM). The VAEGMM outlier detector is first trained on a batch of unlabeled, but normal (*inlier*) data. Unsupervised or semi-supervised training is desirable since labeled data is often scarce. The sample energy of the GMM can then be used to determine whether an instance is an outlier (high sample energy) or not (low sample energy). The algorithm is suitable for tabular and image data.

10.2 Usage

10.2.1 Initialize

Parameters:

- `threshold`: threshold value for the sample energy above which the instance is flagged as an outlier.
- `latent_dim`: latent dimension of the VAE.
- `n_gmm`: number of components in the GMM.
- `encoder_net`: `tf.keras.Sequential` instance containing the encoder network. Example:

```
encoder_net = tf.keras.Sequential(
[
    InputLayer(input_shape=(n_features,)),
    Dense(60, activation=tf.nn.tanh),
    Dense(30, activation=tf.nn.tanh),
    Dense(10, activation=tf.nn.tanh),
    Dense(latent_dim, activation=None)
])
```

- `decoder_net`: `tf.keras.Sequential` instance containing the decoder network. Example:

```
decoder_net = tf.keras.Sequential(
[
    InputLayer(input_shape=(latent_dim,)),
    Dense(10, activation=tf.nn.tanh),
    Dense(30, activation=tf.nn.tanh),
```

(continues on next page)

(continued from previous page)

```
    Dense(60, activation=tf.nn.tanh),
    Dense(n_features, activation=None)
])
```

- `gmm_density_net`: layers for the GMM network wrapped in a `tf.keras.Sequential` class. Example:

```
gmm_density_net = tf.keras.Sequential(
[
    InputLayer(input_shape=(latent_dim + 2,)),
    Dense(10, activation=tf.nn.tanh),
    Dense(n_gmm, activation=tf.nn.softmax)
])
```

- `vaegmm`: instead of using a separate encoder, decoder and GMM density net, the VAEGMM can also be passed as a `tf.keras.Model`.
- `samples`: number of samples drawn during detection for each instance to detect.
- `beta`: weight on the KL-divergence loss term following the β-VAE framework. Default equals 1.
- `recon_features`: function to extract features from the reconstructed instance by the decoder. Defaults to a combination of the mean squared reconstruction error and the cosine similarity between the original and reconstructed instances by the VAE.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'* or *'image'*.

Initialized outlier detector example:

```
from alibi_detect.od import OutlierVAEGMM

od = OutlierVAEGMM(
    threshold=7.5,
    encoder_net=encoder_net,
    decoder_net=decoder_net,
    gmm_density_net=gmm_density_net,
    latent_dim=4,
    n_gmm=2,
    samples=10
)
```

10.2.2 Fit

We then need to train the outlier detector. The following parameters can be specified:

- `X`: training batch as a numpy array of preferably normal data.
- `loss_fn`: loss function used for training. Defaults to the custom VAEGMM loss which is a combination of the elbo loss, sample energy of the GMM and a loss term penalizing small values on the diagonals of the covariance matrices in the GMM to avoid trivial solutions. It is important to balance the loss weights below so no single loss term dominates during the optimization.
- `w_recon`: weight on elbo loss term. Defaults to 1e-7.
- `w_energy`: weight on sample energy loss term. Defaults to 0.1.
- `w_cov_diag`: weight on covariance diagonals. Defaults to 0.005.
- `optimizer`: optimizer used for training. Defaults to Adam with learning rate 1e-4.

- `cov_elbo`: dictionary with covariance matrix options in case the elbo loss function is used. Either use the full covariance matrix inferred from X (*dict(cov_full=None)*), only the variance (*dict(cov_diag=None)*) or a float representing the same standard deviation for each feature (e.g. *dict(sim=.05)*) which is the default.
- `epochs`: number of training epochs.
- `batch_size`: batch size used during training.
- `verbose`: boolean whether to print training progress.
- `log_metric`: additional metrics whose progress will be displayed if verbose equals True.

```
od.fit(
    X_train,
    epochs=10,
    batch_size=1024
)
```

It is often hard to find a good threshold value. If we have a batch of normal and outlier data and we know approximately the percentage of normal data in the batch, we can infer a suitable threshold:

```
od.infer_threshold(
    X,
    threshold_perc=95
)
```

10.2.3 Detect

We detect outliers by simply calling `predict` on a batch of instances `X` to compute the instance level sample energies. We can also return the instance level outlier score by setting `return_instance_score` to True.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_outlier`: boolean whether instances are above the threshold and therefore outlier instances. The array is of shape *(batch size,)*.
- `instance_score`: contains instance level scores if `return_instance_score` equals True.

```
preds = od.predict(
    X,
    return_instance_score=True
)
```

10.3 Examples

10.3.1 Tabular

Outlier detection on KDD Cup 99

source

CHAPTER

ELEVEN

AUTO-ENCODING GAUSSIAN MIXTURE MODEL

11.1 Overview

The Auto-Encoding Gaussian Mixture Model (AEGMM) Outlier Detector follows the Deep Autoencoding Gaussian Mixture Model for Unsupervised Anomaly Detection paper. The encoder compresses the data while the reconstructed instances generated by the decoder are used to create additional features based on the reconstruction error between the input and the reconstructions. These features are combined with encodings and fed into a Gaussian Mixture Model (GMM). The AEGMM outlier detector is first trained on a batch of unlabeled, but normal (*inlier*) data. Unsupervised or semi-supervised training is desirable since labeled data is often scarce. The sample energy of the GMM can then be used to determine whether an instance is an outlier (high sample energy) or not (low sample energy). The algorithm is suitable for tabular and image data.

11.2 Usage

11.2.1 Initialize

Parameters:

- `threshold`: threshold value for the sample energy above which the instance is flagged as an outlier.
- `n_gmm`: number of components in the GMM.
- `encoder_net`: `tf.keras.Sequential` instance containing the encoder network. Example:

```
encoder_net = tf.keras.Sequential(
[
    InputLayer(input_shape=(n_features,)),
    Dense(60, activation=tf.nn.tanh),
    Dense(30, activation=tf.nn.tanh),
    Dense(10, activation=tf.nn.tanh),
    Dense(latent_dim, activation=None)
])
```

- `decoder_net`: `tf.keras.Sequential` instance containing the decoder network. Example:

```
decoder_net = tf.keras.Sequential(
[
    InputLayer(input_shape=(latent_dim,)),
    Dense(10, activation=tf.nn.tanh),
    Dense(30, activation=tf.nn.tanh),
    Dense(60, activation=tf.nn.tanh),
```

(continues on next page)

(continued from previous page)

```
    Dense(n_features, activation=None)
])
```

- `gmm_density_net`: layers for the GMM network wrapped in a `tf.keras.Sequential` class. Example:

```
gmm_density_net = tf.keras.Sequential(
[
    InputLayer(input_shape=(latent_dim + 2,)),
    Dense(10, activation=tf.nn.tanh),
    Dense(n_gmm, activation=tf.nn.softmax)
])
```

- `aegmm`: instead of using a separate encoder, decoder and GMM density net, the AEGMM can also be passed as a `tf.keras.Model`.
- `recon_features`: function to extract features from the reconstructed instance by the decoder. Defaults to a combination of the mean squared reconstruction error and the cosine similarity between the original and reconstructed instances by the AE.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'* or *'image'*.

Initialized outlier detector example:

```
from alibi_detect.od import OutlierAEGMM

od = OutlierAEGMM(
    threshold=7.5,
    encoder_net=encoder_net,
    decoder_net=decoder_net,
    gmm_density_net=gmm_density_net,
    n_gmm=2
)
```

11.2.2 Fit

We then need to train the outlier detector. The following parameters can be specified:

- `X`: training batch as a numpy array of preferably normal data.
- `loss_fn`: loss function used for training. Defaults to the custom AEGMM loss which is a combination of the mean squared reconstruction error, the sample energy of the GMM and a loss term penalizing small values on the diagonals of the covariance matrices in the GMM to avoid trivial solutions. It is important to balance the loss weights below so no single loss term dominates during the optimization.
- `w_energy`: weight on sample energy loss term. Defaults to 0.1.
- `w_cov_diag`: weight on covariance diagonals. Defaults to 0.005.
- `optimizer`: optimizer used for training. Defaults to Adam with learning rate 1e-4.
- `epochs`: number of training epochs.
- `batch_size`: batch size used during training.
- `verbose`: boolean whether to print training progress.
- `log_metric`: additional metrics whose progress will be displayed if verbose equals True.

```
od.fit(
    X_train,
    epochs=10,
    batch_size=1024
)
```

It is often hard to find a good threshold value. If we have a batch of normal and outlier data and we know approximately the percentage of normal data in the batch, we can infer a suitable threshold:

```
od.infer_threshold(
    X,
    threshold_perc=95
)
```

11.2.3 Detect

We detect outliers by simply calling `predict` on a batch of instances `X` to compute the instance level sample energies. We can also return the instance level outlier score by setting `return_instance_score` to True.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_outlier`: boolean whether instances are above the threshold and therefore outlier instances. The array is of shape *(batch size,)*.
- `instance_score`: contains instance level scores if `return_instance_score` equals True.

```
preds = od.predict(
    X,
    return_instance_score=True
)
```

11.3 Examples

11.3.1 Tabular

Outlier detection on KDD Cup 99

source

CHAPTER

TWELVE

LIKELIHOOD RATIOS FOR OUTLIER DETECTION

12.1 Overview

The outlier detector described by Ren et al. (2019) in Likelihood Ratios for Out-of-Distribution Detection uses the likelihood ratio (LLR) between 2 generative models as the outlier score. One model is trained on the original data while the other is trained on a perturbed version of the dataset. This is based on the observation that the log likelihood for an instance under a generative model can be heavily affected by population level background statistics. The second generative model is therefore trained to capture the background statistics still present in the perturbed data while the semantic features have been erased by the perturbations.

The perturbations are added using an independent and identical Bernoulli distribution with rate μ which substitutes a feature with one of the other possible feature values with equal probability. For images, this means for instance changing a pixel with a different pixel value randomly sampled within the 0 to 255 pixel range. The package also contains a PixelCNN++ implementation adapted from the official TensorFlow Probability version, and available as a standalone model in `alibi_detect.models.pixelcnn`.

12.2 Usage

12.2.1 Initialize

Parameters:

- `threshold`: outlier threshold value used for the negative likelihood ratio. Scores above the threshold are flagged as outliers.
- `model`: a generative model, either as a `tf.keras.Model`, TensorFlow Probability distribution or built-in PixelCNN++ model.
- `model_background`: optional separate model fit on the perturbed background data. If this is not specified, a copy of `model` will be used.
- `log_prob`: if the model does not have a `log_prob` function like e.g. a TensorFlow Probability distribution, a function needs to be passed that evaluates the log likelihood.
- `sequential`: flag whether the data is sequential or not. Used to create targets during training. Defaults to *False*.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'* or *'image'*.

Initialized outlier detector example:

```
from alibi_detect.od import LLR
from alibi_detect.models import PixelCNN

image_shape = (28, 28, 1)
model = PixelCNN(image_shape)
od = LLR(threshold=-100, model=model)
```

12.2.2 Fit

We then need to train the 2 generative models in sequence. The following parameters can be specified:

- `X`: training batch as a numpy array of preferably normal data.
- `mutate_fn`: function used to create the perturbations. Defaults to an independent and identical Bernoulli distribution with rate μ
- `mutate_fn_kwargs`: kwargs for `mutate_fn`. For the default function, the mutation rate and feature range needs to be specified, e.g. *dict(rate=.2, feature_range=(0,255))*.
- `loss_fn`: loss function used for the generative models.
- `loss_fn_kwargs`: kwargs for the loss function.
- `optimizer`: optimizer used for training. Defaults to Adam with learning rate 1e-3.
- `epochs`: number of training epochs.
- `batch_size`: batch size used during training.
- `log_metric`: additional metrics whose progress will be displayed if verbose equals True.

```
od.fit(X_train, epochs=10, batch_size=32)
```

It is often hard to find a good threshold value. If we have a batch of normal and outlier data and we know approximately the percentage of normal data in the batch, we can infer a suitable threshold:

```
od.infer_threshold(X, threshold_perc=95, batch_size=32)
```

12.2.3 Detect

We detect outliers by simply calling `predict` on a batch of instances `X`. Detection can be customized via the following parameters:

- `outlier_type`: either *'instance'* or *'feature'*. If the outlier type equals *'instance'*, the outlier score at the instance level will be used to classify the instance as an outlier or not. If *'feature'* is selected, outlier detection happens at the feature level (e.g. by pixel in images).
- `batch_size`: batch size used for model prediction calls.
- `return_feature_score`: boolean whether to return the feature level outlier scores.
- `return_instance_score`: boolean whether to return the instance level outlier scores.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_outlier`: boolean whether instances or features are above the threshold and therefore outliers. If `outlier_type` equals *'instance'*, then the array is of shape *(batch size,)*. If it equals *'feature'*, then the array is of shape *(batch size, instance shape)*.

- `feature_score`: contains feature level scores if `return_feature_score` equals True.
- `instance_score`: contains instance level scores if `return_instance_score` equals True.

```
preds = od.predict(X, outlier_type='instance', batch_size=32)
```

12.3 Examples

12.3.1 Image

Likelihood Ratio Outlier Detection with PixelCNN++

12.3.2 Sequential Data

Likelihood Ratio Outlier Detection on Genomic Sequences

source

CHAPTER

THIRTEEN

PROPHET DETECTOR

13.1 Overview

The Prophet outlier detector uses the Prophet time series forecasting package explained in this excellent paper. The underlying Prophet model is a decomposable univariate time series model combining trend, seasonality and holiday effects. The model forecast also includes an uncertainty interval around the estimated trend component using the MAP estimate of the extrapolated model. Alternatively, full Bayesian inference can be done at the expense of increased compute. The upper and lower values of the uncertainty interval can then be used as outlier thresholds for each point in time. First, the distance from the observed value to the nearest uncertainty boundary (upper or lower) is computed. If the observation is within the boundaries, the outlier score equals the negative distance. As a result, the outlier score is the lowest when the observation equals the model prediction. If the observation is outside of the boundaries, the score equals the distance measure and the observation is flagged as an outlier. One of the main drawbacks of the method however is that you need to refit the model as new data comes in. This is undesirable for applications with high throughput and real-time detection.

To use this detector, first install Prophet by running `pip install alibi-detect[prophet]`.

13.2 Usage

13.2.1 Initialize

Parameters:

- `threshold`: width of the uncertainty intervals of the forecast, used as outlier threshold. Equivalent to `interval_width`. If the instance lies outside of the uncertainty intervals, it is flagged as an outlier. If `mcmc_samples` equals 0, it is the uncertainty in the trend using the MAP estimate of the extrapolated model. If `mcmc_samples` >0, then uncertainty over all parameters is used.
- `growth`: *'linear'* or *'logistic'* to specify a linear or logistic trend.
- `cap`: growth cap in case growth equals *'logistic'*.
- `holidays`: pandas DataFrame with columns *'holiday'* (string) and *'ds'* (dates) and optionally columns *'lower_window'* and *'upper_window'* which specify a range of days around the date to be included as holidays.
- `holidays_prior_scale`: parameter controlling the strength of the holiday components model. Higher values imply a more flexible trend, more prone to more overfitting.
- `country_holidays`: include country-specific holidays via country abbreviations. The holidays for each country are provided by the holidays package in Python. A list of available countries and the country name to use is available on: https://github.com/dr-prodigy/python-holidays. Additionally, Prophet includes holidays

for: Brazil (BR), Indonesia (ID), India (IN), Malaysia (MY), Vietnam (VN), Thailand (TH), Philippines (PH), Turkey (TU), Pakistan (PK), Bangladesh (BD), Egypt (EG), China (CN) and Russian (RU).

- `changepoint_prior_scale`: parameter controlling the flexibility of the automatic changepoint selection. Large values will allow many changepoints, potentially leading to overfitting.
- `changepoint_range`: proportion of history in which trend changepoints will be estimated. Higher values means more changepoints, potentially leading to overfitting.
- `seasonality_mode`: either *'additive'* or *'multiplicative'*.
- `daily_seasonality`: can be *'auto'*, True, False, or a number of Fourier terms to generate.
- `weekly_seasonality`: can be *'auto'*, True, False, or a number of Fourier terms to generate.
- `yearly_seasonality`: can be *'auto'*, True, False, or a number of Fourier terms to generate.
- `add_seasonality`: manually add one or more seasonality components. Pass a list of dicts containing the keys *'name'*, *'period'*, *'fourier_order'* (obligatory), *'prior_scale'* and *'mode'* (optional).
- `seasonality_prior_scale`: parameter controlling the strength of the seasonality model. Larger values allow the model to fit larger seasonal fluctuations, potentially leading to overfitting.
- `uncertainty_samples`: number of simulated draws used to estimate uncertainty intervals.
- `mcmc_samples`: If > *0*, will do full Bayesian inference with the specified number of MCMC samples. If *0*, will do MAP estimation.

Initialized outlier detector example:

```python
from alibi_detect.od import OutlierProphet

od = OutlierProphet(
    threshold=0.9,
    growth='linear'
)
```

13.2.2 Fit

We then need to train the outlier detector. The `fit` method takes a pandas DataFrame *df* with as columns *'ds'* containing the dates or timestamps and *'y'* for the time series being investigated. The date format is ideally *YYYY-MM-DD* and timestamp format *YYYY-MM-DD HH:MM:SS*.

```python
od.fit(df)
```

13.2.3 Detect

We detect outliers by simply calling `predict` on a DataFrame *df*, again with columns *'ds'* and *'y'* to compute the instance level outlier scores. We can also return the instance level outlier score or the raw Prophet model forecast by setting respectively `return_instance_score` or `return_forecast` to True. **It is important that the dates or timestamps of the test data follow the training data**.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_outlier`: DataFrame with columns *'ds'* containing the dates or timestamps and *'is_outlier'* a boolean whether instances are above the threshold and therefore outlier instances.

- `instance_score`: DataFrame with *'ds'* and *'instance_score'* which contains instance level scores if `return_instance_score` equals True.
- `forecast`: DataFrame with the raw model predictions if `return_forecast` equals True. The DataFrame contains columns with the upper and lower boundaries (*'yhat_upper'* and *'yhat_lower'*), the model predictions (*'yhat'*), and the decomposition of the prediction in the different components (trend, seasonality, holiday).

```
preds = od.predict(
    df,
    return_instance_score=True,
    return_forecast=True
)
```

13.3 Examples

Time-series outlier detection using Prophet on weather data

source

CHAPTER

FOURTEEN

SPECTRAL RESIDUAL

14.1 Overview

The Spectral Residual outlier detector is based on the paper Time-Series Anomaly Detection Service at Microsoft and is suitable for **unsupervised online anomaly detection in univariate time series** data. The algorithm first computes the Fourier Transform of the original data. Then it computes the *spectral residual* of the log amplitude of the transformed signal before applying the Inverse Fourier Transform to map the sequence back from the frequency to the time domain. This sequence is called the *saliency map*. The anomaly score is then computed as the relative difference between the saliency map values and their moving averages. If the score is above a threshold, the value at a specific timestep is flagged as an outlier. For more details, please check out the paper.

14.2 Usage

14.2.1 Initialize

Parameters:

- `threshold`: threshold used to classify outliers. Relative saliency map distance from the moving average.
- `window_amp`: window used for the moving average in the *spectral residual* computation. The spectral residual is the difference between the log amplitude of the Fourier Transform and a convolution of the log amplitude over `window_amp`.
- `window_local`: window used for the moving average in the outlier score computation. The outlier score computes the relative difference between the saliency map and a moving average of the saliency map over `window_local` timesteps.
- `n_est_points`: number of estimated points padded to the end of the sequence.
- `n_grad_points`: number of points used for the gradient estimation of the additional points padded to the end of the sequence. The paper sets this value to 5.

Initialized outlier detector example:

```python
from alibi_detect.od import SpectralResidual

od = SpectralResidual(
    threshold=1.,
    window_amp=20,
    window_local=20,
    n_est_points=10,
    n_grad_points=5
)
```

It is often hard to find a good threshold value. If we have a time series containing both normal and outlier data and we know approximately the percentage of normal data in the time series, we can infer a suitable threshold:

```
od.infer_threshold(
    X,
    t=t,  # array with timesteps, assumes dt=1 between observations if omitted
    threshold_perc=95
)
```

14.2.2 Detect

We detect outliers by simply calling `predict` on a time series `X` to compute the outlier scores and flag the anomalies. We can also return the instance (timestep) level outlier score by setting `return_instance_score` to True.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_outlier`: boolean whether instances are above the threshold and therefore outlier instances. The array is of shape *(timesteps,)*.
- `instance_score`: contains instance level scores if `return_instance_score` equals True.

```
preds = od.predict(
    X,
    t=t,  # array with timesteps, assumes dt=1 between observations if omitted
    return_instance_score=True
)
```

14.3 Examples

Time series outlier detection with Spectral Residuals on synthetic data

source

CHAPTER

FIFTEEN

SEQUENCE-TO-SEQUENCE (SEQ2SEQ)

15.1 Overview

The Sequence-to-Sequence (Seq2Seq) outlier detector consists of 2 main building blocks: an encoder and a decoder. The encoder consists of a Bidirectional LSTM which processes the input sequence and initializes the decoder. The LSTM decoder then makes sequential predictions for the output sequence. In our case, the decoder aims to reconstruct the input sequence. If the input data cannot be reconstructed well, the reconstruction error is high and the data can be flagged as an outlier. The reconstruction error is measured as the mean squared error (MSE) between the input and the reconstructed instance.

Since even for normal data the reconstruction error can be state-dependent, we add an outlier threshold estimator network to the Seq2Seq model. This network takes in the hidden state of the decoder at each timestep and predicts the estimated reconstruction error for normal data. As a result, the outlier threshold is not static and becomes a function of the model state. This is similar to Park et al. (2017), but while they train the threshold estimator separately from the Seq2Seq model with a Support-Vector Regressor, we train a neural net regression network end-to-end with the Seq2Seq model.

The detector is first trained on a batch of unlabeled, but normal (*inlier*) data. Unsupervised training is desireable since labeled data is often scarce. The Seq2Seq outlier detector is suitable for both **univariate and multivariate time series**.

15.2 Usage

15.2.1 Initialize

Parameters:

- `n_features`: number of features in the time series.
- `seq_len`: sequence length fed into the Seq2Seq model.
- `threshold`: threshold used for outlier detection. Can be a float or feature-wise array.
- `seq2seq`: optionally pass an already defined or pretrained Seq2Seq model to the outlier detector as a `tf.keras.Model`.
- `threshold_net`: optionally pass the layers for the threshold estimation network wrapped in a `tf.keras.Sequential` instance. Example:

```
threshold_net = tf.keras.Sequential(
    [
        InputLayer(input_shape=(seq_len, latent_dim)),
```

(continues on next page)

(continued from previous page)

```
        Dense(64, activation=tf.nn.relu),
        Dense(64, activation=tf.nn.relu),
    ])
```

- `latent_dim`: latent dimension of the encoder and decoder.
- `output_activation`: activation used in the Dense output layer of the decoder.
- `beta`: weight on the threshold estimation mean-squared error (MSE) loss term.

Initialized outlier detector example:

```
from alibi_detect.od import OutlierSeq2Seq

n_features = 2
seq_len = 50

od = OutlierSeq2Seq(n_features,
                    seq_len,
                    threshold=None,
                    threshold_net=threshold_net,
                    latent_dim=100)
```

15.2.2 Fit

We then need to train the outlier detector. The following parameters can be specified:

- `X`: univariate or multivariate time series array with preferably normal data used for training. Shape equals *(batch, n_features)* or *(batch, seq_len, n_features)*.
- `loss_fn`: loss function used for training. Defaults to the MSE loss.
- `optimizer`: optimizer used for training. Defaults to Adam with learning rate 1e-3.
- `epochs`: number of training epochs.
- `batch_size`: batch size used during training.
- `verbose`: boolean whether to print training progress.
- `log_metric`: additional metrics whose progress will be displayed if verbose equals True.

```
od.fit(X_train, epochs=20)
```

It is often hard to find a good threshold value. If we have a batch of normal and outlier data and we know approximately the percentage of normal data in the batch, we can infer a suitable threshold. We can either set the threshold over both features combined or determine a feature-wise threshold. Here we opt for the feature-wise threshold. This is for instance useful when different features have different variance or sensitivity to outliers. The snippet assumes there are about 5% outliers in the first feature and 10% in the second:

```
od.infer_threshold(X, threshold_perc=np.array([95, 90]))
```

15.2.3 Detect

We detect outliers by simply calling `predict` on a batch of instances `X`. Detection can be customized via the following parameters:

- `outlier_type`: either *'instance'* or *'feature'*. If the outlier type equals *'instance'*, the outlier score at the instance level will be used to classify the instance as an outlier or not. If *'feature'* is selected, outlier detection happens at the feature level. It is **important to distinguish 2 use cases**:
- `X` has shape *(batch, n_features)*:
  - There are *batch* instances with *n_features* features per instance.
- `X` has shape *(batch, seq_len, n_features)*
  - Now there are *batch* instances with *seq_len x n_features* features per instance.
- `outlier_perc`: percentage of the sorted (descending) feature level outlier scores. We might for instance want to flag a multivariate time series as an outlier at a specific timestamp if at least 75% of the feature values are on average above the threshold. In this case, we set `outlier_perc` to 75. The default value is 100 (using all the features).
- `return_feature_score`: boolean whether to return the feature level outlier scores.
- `return_instance_score`: boolean whether to return the instance level outlier scores.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_outlier`: boolean whether instances or features are above the threshold and therefore outliers. If `outlier_type` equals *'instance'*, then the array is of shape *(batch,)*. If it equals *'feature'*, then the array is of shape *(batch, seq_len, n_features)* or *(batch, n_features)*, depending on the shape of `X`.
- `feature_score`: contains feature level scores if `return_feature_score` equals True.
- `instance_score`: contains instance level scores if `return_instance_score` equals True.

```
preds = od.predict(X,
                   outlier_type='instance',
                   outlier_perc=100,
                   return_feature_score=True,
                   return_instance_score=True)
```

15.3 Examples

Time series outlier detection with Seq2Seq models on synthetic data

Seq2Seq time series outlier detection on ECG data

source

CHAPTER

SIXTEEN

ADVERSARIAL AUTO-ENCODER

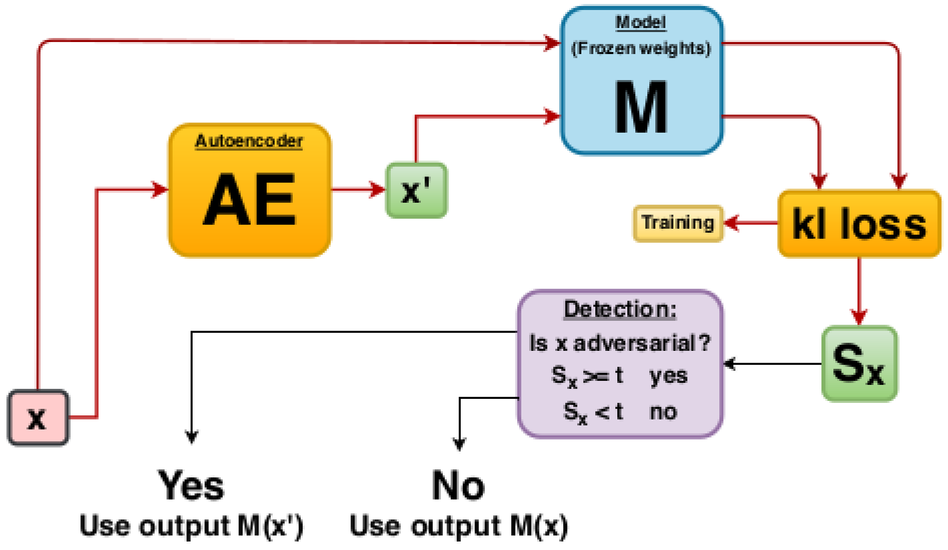
16.1 Overview

The adversarial detector follows the method explained in the Adversarial Detection and Correction by Matching Prediction Distributions paper. Usually, autoencoders are trained to find a transformation T that reconstructs the input instance x as accurately as possible with loss functions that are suited to capture the similarities between x and x' such as the mean squared reconstruction error. The novelty of the adversarial autoencoder (AE) detector relies on the use of a classification model-dependent loss function based on a distance metric in the output space of the model to train the autoencoder network. Given a classification model M we optimise the weights of the autoencoder such that the KL-divergence between the model predictions on x and on x' is minimised. Without the presence of a reconstruction loss term x' simply tries to make sure that the prediction probabilities $M(x')$ and $M(x)$ match without caring about the proximity of x' to x. As a result, x' is allowed to live in different areas of the input feature space than x with different decision boundary shapes with respect to the model M. The carefully crafted adversarial perturbation which is effective around x does not transfer to the new location of x' in the feature space, and the attack is therefore neutralised. Training of the autoencoder is unsupervised since we only need access to the model prediction probabilities and the normal training instances. We do not require any knowledge about the underlying adversarial attack and the classifier weights are frozen during training.

The detector can be used as follows:

- An adversarial score S is computed. S equals the K-L divergence between the model predictions on x and x'.
- If S is above a threshold (explicitly defined or inferred from training data), the instance is flagged as adversarial.
- For adversarial instances, the model M uses the reconstructed instance x' to make a prediction. If the adversarial score is below the threshold, the model makes a prediction on the original instance x.

This procedure is illustrated in the diagram below:



The method is very flexible and can also be used to detect common data corruptions and perturbations which negatively impact the model performance. The algorithm works well on tabular and image data.

16.2 Usage

16.2.1 Initialize

Parameters:

- `threshold`: threshold value above which the instance is flagged as an adversarial instance.
- `encoder_net`: `tf.keras.Sequential` instance containing the encoder network. Example:

```
encoder_net = tf.keras.Sequential(
    [
        InputLayer(input_shape=(32, 32, 3)),
        Conv2D(32, 4, strides=2, padding='same',
               activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
        Conv2D(64, 4, strides=2, padding='same',
               activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
        Conv2D(256, 4, strides=2, padding='same',
               activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
        Flatten(),
        Dense(40)
    ]
)
```

- `decoder_net`: `tf.keras.Sequential` instance containing the decoder network. Example:

```
decoder_net = tf.keras.Sequential(
[
        InputLayer(input_shape=(40,)),
        Dense(4 * 4 * 128, activation=tf.nn.relu),
        Reshape(target_shape=(4, 4, 128)),
```

(continues on next page)

(continued from previous page)

```
        Conv2DTranspose(256, 4, strides=2, padding='same',
                        activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
        Conv2DTranspose(64, 4, strides=2, padding='same',
                        activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
        Conv2DTranspose(3, 4, strides=2, padding='same',
                        activation=None, kernel_regularizer=l1(1e-5))
    ]
)
```

- `ae`: instead of using a separate encoder and decoder, the AE can also be passed as a `tf.keras.Model`.
- `model`: the classifier as a `tf.keras.Model`. Example:

```
inputs = tf.keras.Input(shape=(input_dim,))
outputs = tf.keras.layers.Dense(output_dim, activation=tf.nn.softmax)(inputs)
model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

- `hidden_layer_kld`: dictionary with as keys the number of the hidden layer(s) in the classification model which are extracted and used during training of the adversarial AE, and as values the output dimension for the hidden layer. Extending the training methodology to the hidden layers is optional and can further improve the adversarial correction mechanism.
- `model_hl`: instead of passing a dictionary to `hidden_layer_kld`, a list with tf.keras models for the hidden layer K-L divergence computation can be passed directly.
- `w_model_hl`: Weights assigned to the loss of each model in `model_hl`. Also used to weight the K-L divergence contribution for each model in `model_hl` when computing the adversarial score.
- `temperature`: Temperature used for model prediction scaling. Temperature <1 sharpens the prediction probability distribution which can be beneficial for prediction distributions with high entropy.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'* or *'image'*.

Initialized adversarial detector example:

```
from alibi_detect.ad import AdversarialAE

ad = AdversarialAE(
    encoder_net=encoder_net,
    decoder_net=decoder_net,
    model=model,
    temperature=0.5
)
```

16.2.2 Fit

We then need to train the adversarial detector. The following parameters can be specified:

- `X`: training batch as a numpy array.
- `loss_fn`: loss function used for training. Defaults to the custom adversarial loss.
- `w_model`: weight on the loss term minimizing the K-L divergence between model prediction probabilities on the original and reconstructed instance. Defaults to 1.
- `w_recon`: weight on the mean squared error reconstruction loss term. Defaults to 0.
- `optimizer`: optimizer used for training. Defaults to Adam with learning rate 1e-3.

- `epochs`: number of training epochs.
- `batch_size`: batch size used during training.
- `verbose`: boolean whether to print training progress.
- `log_metric`: additional metrics whose progress will be displayed if verbose equals True.
- `preprocess_fn`: optional data preprocessing function applied per batch during training.

```
ad.fit(X_train, epochs=50)
```

The threshold for the adversarial score can be set via `infer_threshold`. We need to pass a batch of instances X and specify what percentage of those we consider to be normal via `threshold_perc`. Even if we only have normal instances in the batch, it might be best to set the threshold value a bit lower (e.g. 95%) since the the model could have misclassified training instances leading to a higher score if the reconstruction picked up features from the correct class or some instances might look adversarial in the first place.

```
ad.infer_threshold(X_train, threshold_perc=95, batch_size=64)
```

16.2.3 Detect

We detect adversarial instances by simply calling `predict` on a batch of instances `X`. We can also return the instance level adversarial score by setting `return_instance_score` to True.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_adversarial`: boolean whether instances are above the threshold and therefore adversarial instances. The array is of shape *(batch size,)*.
- `instance_score`: contains instance level scores if `return_instance_score` equals True.

```
preds_detect = ad.predict(X, batch_size=64, return_instance_score=True)
```

16.2.4 Correct

We can immediately apply the procedure sketched out in the above diagram via `correct`. The method also returns a dictionary with `meta` and `data` keys. On top of the information returned by `detect`, 3 additional fields are returned under `data`:

- `corrected`: model predictions by following the adversarial detection and correction procedure.
- `no_defense`: model predictions without the adversarial correction.
- `defense`: model predictions where each instance is corrected by the defense, regardless of the adversarial score.

```
preds_correct = ad.correct(X, batch_size=64, return_instance_score=True)
```

16.3 Examples

16.3.1 Image

Adversarial detection on CIFAR10

source

CHAPTER

SEVENTEEN

MODEL DISTILLATION

17.1 Overview

Model distillation is a technique that is used to transfer knowledge from a large network to a smaller network. Typically, it consists of training a second model with a simplified architecture on soft targets (the output distributions or the logits) obtained from the original model.

Here, we apply model distillation to obtain harmfulness scores, by comparing the output distributions of the original model with the output distributions of the distilled model, in order to detect adversarial data, malicious data drift or data corruption. We use the following definition of harmful and harmless data points:

- Harmful data points are defined as inputs for which the model's predictions on the uncorrupted data are correct while the model's predictions on the corrupted data are wrong.
- Harmless data points are defined as inputs for which the model's predictions on the uncorrupted data are correct and the model's predictions on the corrupted data remain correct.

Analogously to the adversarial AE detector, which is also part of the library, the model distillation detector picks up drift that reduces the performance of the classification model.

The detector can be used as follows:

- Given an input x, an adversarial score $S(x)$ is computed. $S(x)$ equals the value loss function employed for distillation calculated between the original model's output and the distilled model's output on x.
- If $S(x)$ is above a threshold (explicitly defined or inferred from training data), the instance is flagged as adversarial.

17.2 Usage

17.2.1 Initialize

Parameters:

- `threshold`: threshold value above which the instance is flagged as an adversarial instance.
- `distilled_model`: `tf.keras.Sequential` instance containing the model used for distillation. Example:

```
distilled_model = tf.keras.Sequential(
    [
        tf.keras.InputLayer(input_shape=(input_dim,)),
        tf.keras.layers.Dense(output_dim, activation=tf.nn.softmax)
```

(continues on next page)

(continued from previous page)

```
    ]
)
```

- `model`: the classifier as a `tf.keras.Model`. Example:

```
inputs = tf.keras.Input(shape=(input_dim,))
hidden = tf.keras.layers.Dense(hidden_dim)(inputs)
outputs = tf.keras.layers.Dense(output_dim, activation=tf.nn.softmax)(hidden)
model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

- `loss_type`: type of loss used for distillation. Supported losses: 'kld', 'xent'.
- `temperature`: Temperature used for model prediction scaling. Temperature <1 sharpens the prediction probability distribution which can be beneficial for prediction distributions with high entropy.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'* or *'image'*.

Initialized detector example:

```
from alibi_detect.ad import ModelDistillation

ad = ModelDistillation(
    distilled_model=distilled_model,
    model=model,
    temperature=0.5
)
```

17.2.2 Fit

We then need to train the detector. The following parameters can be specified:

- `X`: training batch as a numpy array.
- `loss_fn`: loss function used for training. Defaults to the custom model distillation loss.
- `optimizer`: optimizer used for training. Defaults to Adam with learning rate 1e-3.
- `epochs`: number of training epochs.
- `batch_size`: batch size used during training.
- `verbose`: boolean whether to print training progress.
- `log_metric`: additional metrics whose progress will be displayed if verbose equals True.
- `preprocess_fn`: optional data preprocessing function applied per batch during training.

```
ad.fit(X_train, epochs=50)
```

The threshold for the adversarial / harmfulness score can be set via `infer_threshold`. We need to pass a batch of instances X and specify what percentage of those we consider to be normal via `threshold_perc`. Even if we only have normal instances in the batch, it might be best to set the threshold value a bit lower (e.g. 95%) since the model could have misclassified training instances.

```
ad.infer_threshold(X_train, threshold_perc=95, batch_size=64)
```

17.2.3 Detect

We detect adversarial / harmful instances by simply calling `predict` on a batch of instances `X`. We can also return the instance level score by setting `return_instance_score` to True.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_adversarial`: boolean whether instances are above the threshold and therefore adversarial instances. The array is of shape *(batch size,)*.
- `instance_score`: contains instance level scores if `return_instance_score` equals True.

```
preds_detect = ad.predict(X, batch_size=64, return_instance_score=True)
```

17.3 Examples

17.3.1 Image

Harmful drift detection through model distillation on CIFAR10

source

CHAPTER

EIGHTEEN

CHI-SQUARED

18.1 Overview

The drift detector applies feature-wise Chi-Squared tests for the categorical features. For multivariate data, the obtained p-values for each feature are aggregated either via the Bonferroni or the False Discovery Rate (FDR) correction. The Bonferroni correction is more conservative and controls for the probability of at least one false positive. The FDR correction on the other hand allows for an expected fraction of false positives to occur. Similarly to the other drift detectors, a preprocessing steps could be applied, but the output features need to be categorical.

18.2 Usage

18.2.1 Initialize

Arguments:

- `x_ref`: Data used as reference distribution.

Keyword arguments:

- `p_val`: p-value used for significance of the Chi-Squared test for. If the FDR correction method is used, this corresponds to the acceptable q-value.
- `categories_per_feature`: Optional dictionary with as keys the feature column index and as values the number of possible categorical values for that feature or a list with the possible values. If you know how many categories are present for a given feature you could pass this in the `categories_per_feature` dict in the *Dict[int, int]* format, e.g. *{0: 3, 3: 2}*. If you pass N categories this will assume the possible values for the feature are [0, . . . , N-1]. You can also explicitly pass the possible categories in the *Dict[int, List[int]]* format, e.g. *{0: [0, 1, 2], 3: [0, 55]}*. Note that the categories can be arbitrary *int* values. If it is not specified, `categories_per_feature` is inferred from `x_ref`.
- `preprocess_x_ref`: Whether to already count and store the number of instances for each possible category of each variable of the reference data `x_ref` when initializing the detector. If a preprocessing step is specified, the step will be applied first. Defaults to *True*. It is possible that it needs to be set to *False* if the preprocessing step requires statistics from both the reference and test data.
- `update_x_ref`: Reference data can optionally be updated to the last N instances seen by the detector or via reservoir sampling with size N. For the former, the parameter equals *{'last': N}* while for reservoir sampling *{'reservoir_sampling': N}* is passed.
- `preprocess_fn`: Function to preprocess the data before computing the data drift metrics. Typically a dimensionality reduction technique. Needs to return categorical features for the Chi-Squared detector.
- `correction`: Correction type for multivariate data. Either *'bonferroni'* or *'fdr'* (False Discovery Rate).

- `n_features`: Number of features used in the Chi-Squared test. No need to pass it if no preprocessing takes place. In case of a preprocessing step, this can also be inferred automatically but could be more expensive to compute.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'*.

Initialized drift detector example:

```
from alibi_detect.cd import ChiSquareDrift

cd = ChiSquareDrift(x_ref, p_val=0.05)
```

18.2.2 Detect Drift

We detect data drift by simply calling `predict` on a batch of instances `x`. We can return the feature-wise p-values before the multivariate correction by setting `return_p_val` to *True*. The drift can also be detected at the feature level by setting `drift_type` to *'feature'*. No multivariate correction will take place since we return the output of *n_features* univariate tests. For drift detection on all the features combined with the correction, use *'batch'*. `return_p_val` equal to *True* will also return the threshold used by the detector (either for the univariate case or after the multivariate correction).

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_drift`: 1 if the sample tested has drifted from the reference data and 0 otherwise.
- `p_val`: contains feature-level p-values if `return_p_val` equals *True*.
- `threshold`: for feature-level drift detection the threshold equals the p-value used for the significance of the Chi-Square test. Otherwise the threshold after the multivariate correction (either *bonferroni* or *fdr*) is returned.
- `distance`: feature-wise Chi-Square test statistics between the reference data and the new batch if `return_distance` equals *True*.

```
preds = cd.predict(x, drift_type='batch', return_p_val=True, return_distance=True)
```

18.2.3 Saving and loading

The drift detectors can be saved and loaded in the same way as other detectors:

```
from alibi_detect.utils import save_detector, load_detector

filepath = 'my_path'
save_detector(cd, filepath)
cd = load_detector(filepath)
```

18.3 Examples

Drift detection on income prediction

source

CHAPTER

NINETEEN

CLASSIFIER

19.1 Overview

The classifier-based drift detector Lopez-Paz and Oquab, 2017 simply tries to correctly distinguish instances from the reference set vs. the test set. The classifier is trained to output the probability that a given instance belongs to the test set. If the probabilities it assigns to unseen tests instances are significantly higher (as determined by a Kolmogorov-Smirnoff test) to those it assigns to unseen reference instances then the test set must differ from the reference set and drift is flagged. To leverage all the available reference and test data, stratified cross-validation can be applied and the out-of-fold predictions are used for the significance test. Note that a new classifier is trained for each test set or even each fold within the test set.

19.2 Usage

19.2.1 Initialize

Arguments:

- `x_ref`: Data used as reference distribution.
- `model`: Classification model used for drift detection. Both **TensorFlow** and **PyTorch** models are supported.

Keyword arguments:

- `backend`: Specify the backend (*tensorflow* or *pytorch*). This depends on the framework of the `model`. Defaults to *tensorflow*.
- `p_val`: p-value threshold used for the significance of the test.
- `preprocess_x_ref`: Whether to already apply the (optional) preprocessing step to the reference data at initialization and store the preprocessed data. Dependent on the preprocessing step, this can reduce the computation time for the predict step significantly, especially when the reference dataset is large. Defaults to *True*. It is possible that it needs to be set to *False* if the preprocessing step requires statistics from both the reference and test data, such as the mean or standard deviation.
- `update_x_ref`: Reference data can optionally be updated to the last N instances seen by the detector or via reservoir sampling with size N. For the former, the parameter equals *{'last': N}* while for reservoir sampling *{'reservoir_sampling': N}* is passed.
- `preprocess_fn`: Function to preprocess the data before computing the data drift metrics.
- `preds_type`: Whether the model outputs 'probs' or 'logits'.
- `binarize_preds`: Whether to test for discrepency on soft (e.g. probs/logits) model predictions directly with a K-S test or binarise to 0-1 prediction errors and apply a binomial test.

- `train_size`: Optional fraction (float between 0 and 1) of the dataset used to train the classifier. The drift is detected on *1 - train_size*. Cannot be used in combination with `n_folds`.
- `n_folds`: Optional number of stratified folds used for training. The model preds are then calculated on all the out-of-fold predictions. This allows to leverage all the reference and test data for drift detection at the expense of longer computation. If both `train_size` and `n_folds` are specified, `n_folds` is prioritized.
- `seed`: Optional random seed for fold selection.
- `optimizer`: Optimizer used during training of the classifier. From `torch.optim` for PyTorch and `tf.keras.optimizers` for TensorFlow.
- `learning_rate`: Learning rate for the optimizer.
- `batch_size`: Batch size used during training of the classifier.
- `epochs`: Number of training epochs for the classifier. Applies to each fold if `n_folds` is specified.
- `verbose`: Verbosity level during the training of the classifier. 0 is silent, 1 a progress bar and 2 prints the statistics after each epoch.
- `train_kwargs`: Optional additional kwargs for *model.fit()* when fitting the classifier for TensorFlow or for the built-in PyTorch trainer function (`from alibi_detect.models.pytorch import trainer`).
- `data_type`: Optionally specify the data type (e.g. tabular, image or time-series). Added to metadata.

Additional TensorFlow keyword arguments:

- `compile_kwargs`: Optional additional kwargs for *model.compile()* when compiling the classifier.

Additional PyTorch keyword arguments:

- `device`: *cuda* or *gpu* to use the GPU and *cpu* for the CPU. If the device is not specified, the detector will try to leverage the GPU if possible and otherwise fall back on CPU.

Initialized **TensorFlow** drift detector example:

```python
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, Dense, Flatten, Input
from alibi_detect.cd import ClassifierDrift

model = tf.keras.Sequential(
  [
      Input(shape=(32, 32, 3)),
      Conv2D(8, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(16, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(32, 4, strides=2, padding='same', activation=tf.nn.relu),
      Flatten(),
      Dense(2, activation='softmax')
  ]
)

cd = ClassifierDrift(x_ref, model, p_val=.05, preds_type='probs', n_folds=5, epochs=2)
```

A similar detector using **PyTorch**:

```python
import torch.nn as nn

model = nn.Sequential(
    nn.Conv2d(3, 8, 4, stride=2, padding=0),
    nn.ReLU(),
    nn.Conv2d(8, 16, 4, stride=2, padding=0),
```

(continues on next page)

(continued from previous page)

```
    nn.ReLU(),
    nn.Conv2d(16, 32, 4, stride=2, padding=0),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(128, 2)
)

cd = ClassifierDrift(x_ref, model, backend='pytorch', p_val=.05, preds_type='logits')
```

19.2.2 Detect Drift

We detect data drift by simply calling `predict` on a batch of instances `x`. `return_p_val` equal to *True* will also return the p-value of the test and `return_distance` equal to *True* will return a notion of strength of the drift.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_drift`: 1 if the sample tested has drifted from the reference data and 0 otherwise.
- `threshold`: the user-defined threshold defining the significance of the test
- `p_val`: the p-value of the test if `return_p_val` equals *True*.
- `distance`: a notion of strength of the drift if `return_distance` equals *True*. Equal to the K-S test statistic assuming `binarize_preds` equals *False* or the relative error reduction over the baseline error expected under the null if `binarize_preds` equals *True*.

```
preds = cd.predict(x)
```

19.2.3 Saving and loading

The drift detectors can be saved and loaded in the same way as other detectors:

```
from alibi_detect.utils.saving import save_detector, load_detector

filepath = 'my_path'
save_detector(cd, filepath)
cd = load_detector(filepath)
```

Currently on the **TensorFlow** backend is supported for `save_detector` and `load_detector`. Adding **PyTorch** support is a near term priority.

19.3 Examples

Drift detection on CIFAR10

source

CHAPTER

TWENTY

KOLMOGOROV-SMIRNOV

20.1 Overview

The drift detector applies feature-wise two-sample Kolmogorov-Smirnov (K-S) tests. For multivariate data, the obtained p-values for each feature are aggregated either via the Bonferroni or the False Discovery Rate (FDR) correction. The Bonferroni correction is more conservative and controls for the probability of at least one false positive. The FDR correction on the other hand allows for an expected fraction of false positives to occur.

For high-dimensional data, we typically want to reduce the dimensionality before computing the feature-wise univariate K-S tests and aggregating those via the chosen correction method. Following suggestions in Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift, we incorporate Untrained AutoEncoders (UAE), black-box shift detection using the classifier's softmax outputs (BBSDs) and PCA as out-of-the box preprocessing methods. Preprocessing methods which do not rely on the classifier will usually pick up drift in the input data, while BBSDs focuses on label shift. The adversarial detector which is part of the library can also be transformed into a drift detector picking up drift that reduces the performance of the classification model. We can therefore combine different preprocessing techniques to figure out if there is drift which hurts the model performance, and whether this drift can be classified as input drift or label shift.

Detecting input data drift (covariate shift) $\Delta p(x)$ for text data requires a custom preprocessing step. We can pick up changes in the semantics of the input by extracting (contextual) embeddings and detect drift on those. Strictly speaking we are not detecting $\Delta p(x)$ anymore since the whole training procedure (objective function, training data etc) for the (pre)trained embeddings has an impact on the embeddings we extract. The library contains functionality to leverage pre-trained embeddings from HuggingFace's transformer package but also allows you to easily use your own embeddings of choice. Both options are illustrated with examples in the *Text drift detection on IMDB movie reviews* notebook.

20.2 Usage

20.2.1 Initialize

Arguments:

- `x_ref`: Data used as reference distribution.

Keyword arguments:

- `p_val`: p-value used for significance of the K-S test. If the FDR correction method is used, this corresponds to the acceptable q-value.
- `preprocess_x_ref`: Whether to already apply the (optional) preprocessing step to the reference data at initialization and store the preprocessed data. Dependent on the preprocessing step, this can reduce the computation time for the predict step significantly, especially when the reference dataset is large. Defaults to *True*. It

is possible that it needs to be set to *False* if the preprocessing step requires statistics from both the reference and test data, such as the mean or standard deviation.

- `update_x_ref`: Reference data can optionally be updated to the last N instances seen by the detector or via reservoir sampling with size N. For the former, the parameter equals *{'last': N}* while for reservoir sampling *{'reservoir_sampling': N}* is passed.
- `preprocess_fn`: Function to preprocess the data before computing the data drift metrics. Typically a dimensionality reduction technique.
- `correction`: Correction type for multivariate data. Either *'bonferroni'* or *'fdr'* (False Discovery Rate).
- `alternative`: Defines the alternative hypothesis. Options are *'two-sided'* (default), *'less'* or *'greater'*.
- `n_features`: Number of features used in the K-S test. No need to pass it if no preprocessing takes place. In case of a preprocessing step, this can also be inferred automatically but could be more expensive to compute.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'* or *'image'*.

Initialized drift detector example:

```
from alibi_detect.cd import KSDrift

cd = KSDrift(x_ref, p_val=0.05)
```

20.2.2 Detect Drift

We detect data drift by simply calling `predict` on a batch of instances `x`. We can return the feature-wise p-values before the multivariate correction by setting `return_p_val` to *True*. The drift can also be detected at the feature level by setting `drift_type` to *'feature'*. No multivariate correction will take place since we return the output of *n_features* univariate tests. For drift detection on all the features combined with the correction, use *'batch'*. `return_p_val` equal to *True* will also return the threshold used by the detector (either for the univariate case or after the multivariate correction).

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_drift`: 1 if the sample tested has drifted from the reference data and 0 otherwise.
- `p_val`: contains feature-level p-values if `return_p_val` equals *True*.
- `threshold`: for feature-level drift detection the threshold equals the p-value used for the significance of the K-S test. Otherwise the threshold after the multivariate correction (either *bonferroni* or *fdr*) is returned.
- `distance`: feature-wise K-S statistics between the reference data and the new batch if `return_distance` equals *True*.

```
preds = cd.predict(x, drift_type='batch', return_p_val=True, return_distance=True)
```

20.2.3 Saving and loading

The drift detectors can be saved and loaded in the same way as other detectors:

```
from alibi_detect.utils import save_detector, load_detector

filepath = 'my_path'
save_detector(cd, filepath)
cd = load_detector(filepath)
```

20.3 Examples

20.3.1 Image

Drift detection on CIFAR10

20.3.2 Text

Text drift detection on IMDB movie reviews

source

CHAPTER

TWENTYONE

MAXIMUM MEAN DISCREPANCY

21.1 Overview

The Maximum Mean Discrepancy (MMD) detector is a kernel-based method for multivariate 2 sample testing. The MMD is a distance-based measure between 2 distributions p and q based on the mean embeddings μ_p and μ_q in a reproducing kernel Hilbert space F:

$$MMD(F, p, q) = ||\mu_p - \mu_q||^2_F \tag{21.1}$$

(21.2)

We can compute unbiased estimates of MMD^2 from the samples of the 2 distributions after applying the kernel trick. We use by default a radial basis function kernel, but users are free to pass their own kernel of preference to the detector. We obtain a p-value via a permutation test on the values of MMD^2.

For high-dimensional data, we typically want to reduce the dimensionality before computing the permutation test. Following suggestions in Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift, we incorporate Untrained AutoEncoders (UAE), black-box shift detection using the classifier's softmax outputs (BBSDs) and PCA as out-of-the box preprocessing methods. Preprocessing methods which do not rely on the classifier will usually pick up drift in the input data, while BBSDs focuses on label shift.

Detecting input data drift (covariate shift) $\Delta p(x)$ for text data requires a custom preprocessing step. We can pick up changes in the semantics of the input by extracting (contextual) embeddings and detect drift on those. Strictly speaking we are not detecting $\Delta p(x)$ anymore since the whole training procedure (objective function, training data etc) for the (pre)trained embeddings has an impact on the embeddings we extract. The library contains functionality to leverage pre-trained embeddings from HuggingFace's transformer package but also allows you to easily use your own embeddings of choice. Both options are illustrated with examples in the *Text drift detection on IMDB movie reviews* notebook.

21.2 Usage

21.2.1 Initialize

Arguments:

- `x_ref`: Data used as reference distribution.

Keyword arguments:

- `backend`: Both **TensorFlow** and **PyTorch** implementations of the MMD detector as well as various preprocessing steps are available. Specify the backend (*tensorflow* or *pytorch*). Defaults to *tensorflow*.
- `p_val`: p-value used for significance of the permutation test.

- `preprocess_x_ref`: Whether to already apply the (optional) preprocessing step to the reference data at initialization and store the preprocessed data. Dependent on the preprocessing step, this can reduce the computation time for the predict step significantly, especially when the reference dataset is large. Defaults to *True*. It is possible that it needs to be set to *False* if the preprocessing step requires statistics from both the reference and test data, such as the mean or standard deviation.
- `update_x_ref`: Reference data can optionally be updated to the last N instances seen by the detector or via reservoir sampling with size N. For the former, the parameter equals *{'last': N}* while for reservoir sampling *{'reservoir_sampling': N}* is passed.
- `preprocess_fn`: Function to preprocess the data before computing the data drift metrics. Typically a dimensionality reduction technique.
- `kernel`: Kernel used when computing the MMD. Defaults to a Gaussian RBF kernel (`from alibi_detect.utils.pytorch import GaussianRBF` or `from alibi_detect.utils.tensorflow import GaussianRBF` dependent on the backend used).
- `sigma`: Optional bandwidth for the kernel as a `np.ndarray`. We can also average over a number of different bandwidths, e.g. `np.array([.5, 1., 1.5])`.
- `configure_kernel_from_x_ref`: If `sigma` is not specified, the detector can infer it via a heuristic and set `sigma` to the median pairwise distance between 2 samples. If `configure_kernel_from_x_ref` is *True*, we can already set `sigma` at initialization of the detector by inferring it from `x_ref`, speeding up the prediction step. If set to *False*, `sigma` is computed separately for each test batch at prediction time.
- `n_permutations`: Number of permutations used in the permutation test.
- `input_shape`: Optionally pass the shape of the input data.
- `data_type`: can specify data type added to the metadata. E.g. *'tabular'* or *'image'*.

Additional PyTorch keyword arguments:

- `device`: *cuda* or *gpu* to use the GPU and *cpu* for the CPU. If the device is not specified, the detector will try to leverage the GPU if possible and otherwise fall back on CPU.

Initialized drift detector example:

```
from alibi_detect.cd import MMDDrift

cd = MMDDrift(x_ref, backend='tensorflow', p_val=.05)
```

The same detector in PyTorch:

```
cd = MMDDrift(x_ref, backend='pytorch', p_val=.05)
```

We can also easily add preprocessing functions for both frameworks. The following example uses a randomly initialized image encoder in PyTorch:

```
from functools import partial
import torch
import torch.nn as nn
from alibi_detect.cd.pytorch import preprocess_drift

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# define encoder
encoder_net = nn.Sequential(
    nn.Conv2d(3, 64, 4, stride=2, padding=0),
    nn.ReLU(),
    nn.Conv2d(64, 128, 4, stride=2, padding=0),
```

(continues on next page)

(continued from previous page)

```
    nn.ReLU(),
    nn.Conv2d(128, 512, 4, stride=2, padding=0),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(2048, 32)
).to(device).eval()

# define preprocessing function
preprocess_fn = partial(preprocess_drift, model=encoder_net, device=device, batch_
↪size=512)

cd = MMDDrift(x_ref, backend='pytorch', p_val=.05, preprocess_fn=preprocess_fn)
```

The same functionality is supported in TensorFlow and the main difference is that you would import from `alibi_detect.cd.tensorflow import preprocess_drift`. Other preprocessing steps such as the output of hidden layers of a model or extracted text embeddings using transformer models can be used in a similar way in both frameworks. TensorFlow example for the hidden layer output:

```
from alibi_detect.cd.tensorflow import HiddenOutput, preprocess_drift

model = # TensorFlow model; tf.keras.Model or tf.keras.Sequential
preprocess_fn = partial(preprocess_drift, model=HiddenOutput(model, layer=-1), batch_
↪size=128)

cd = MMDDrift(x_ref, backend='tensorflow', p_val=.05, preprocess_fn=preprocess_fn)
```

Check out the *Drift detection on CIFAR10* example for more details.

Alibi Detect also includes custom text preprocessing steps in both TensorFlow and PyTorch based on Huggingface's transformers package:

```
import torch
import torch.nn as nn
from transformers import AutoTokenizer
from alibi_detect.cd.pytorch import preprocess_drift
from alibi_detect.models.pytorch import TransformerEmbedding

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model_name = 'bert-base-cased'
tokenizer = AutoTokenizer.from_pretrained(model_name)

embedding_type = 'hidden_state'
layers = [5, 6, 7]
embed = TransformerEmbedding(model_name, embedding_type, layers)
model = nn.Sequential(embed, nn.Linear(768, 256), nn.ReLU(), nn.Linear(256, enc_dim)).
↪to(device).eval()
preprocess_fn = partial(preprocess_drift, model=model, tokenizer=tokenizer, max_
↪len=512, batch_size=32)

# initialise drift detector
cd = MMDDrift(x_ref, backend='pytorch', p_val=.05, preprocess_fn=preprocess_fn)
```

Again the same functionality is supported in TensorFlow but with `from alibi_detect.cd.tensorflow import preprocess_drift` and `from alibi_detect.models.tensorflow import TransformerEmbedding` imports. Check out the *Text drift detection on IMDB movie reviews* example for more information.

21.2.2 Detect Drift

We detect data drift by simply calling `predict` on a batch of instances `x`. We can return the p-value and the threshold of the permutation test by setting `return_p_val` to *True* and the maximum mean discrepancy metric and threshold by setting `return_distance` to *True*.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_drift`: 1 if the sample tested has drifted from the reference data and 0 otherwise.
- `p_val`: contains the p-value if `return_p_val` equals *True*.
- `threshold`: p-value threshold if `return_p_val` equals *True*.
- `distance`: MMD^2 metric between the reference data and the new batch if `return_distance` equals *True*.
- `distance_threshold`: MMD^2 metric value from the permutation test which corresponds to the the p-value threshold.

```
preds = cd.predict(X, return_p_val=True, return_distance=True)
```

21.2.3 Saving and loading

The drift detectors can be saved and loaded in the same way as other detectors:

```
from alibi_detect.utils.saving import save_detector, load_detector

filepath = 'my_path'
save_detector(cd, filepath)
cd = load_detector(filepath)
```

Currently on the **TensorFlow** backend is supported for `save_detector` and `load_detector`. Adding **PyTorch** support is a near term priority.

21.3 Examples

21.3.1 Image

Drift detection on CIFAR10

21.3.2 Text

Text drift detection on IMDB movie reviews

source

CHAPTER

TWENTYTWO

MIXED-TYPE TABULAR DATA

22.1 Overview

The drift detector applies feature-wise two-sample Kolmogorov-Smirnov (K-S) tests for the continuous numerical features and Chi-Squared tests for the categorical features. For multivariate data, the obtained p-values for each feature are aggregated either via the Bonferroni or the False Discovery Rate (FDR) correction. The Bonferroni correction is more conservative and controls for the probability of at least one false positive. The FDR correction on the other hand allows for an expected fraction of false positives to occur. Similarly to the other drift detectors, a preprocessing steps could be applied, but the output features need to be categorical.

22.2 Usage

22.2.1 Initialize

Arguments:

- `x_ref`: Data used as reference distribution.

Keyword arguments:

- `p_val`: p-value used for significance of the K-S and Chi-Squared test across all features. If the FDR correction method is used, this corresponds to the acceptable q-value.
- `categories_per_feature`: Dictionary with as keys the column indices of the categorical features and optionally as values the number of possible categorical values for that feature or a list with the possible values. If you know which features are categorical and simply want to infer the possible values of the categorical feature from the reference data you can pass a *Dict[int, NoneType]* such as *{0: None, 3: None}* if features 0 and 3 are categorical. If you also know how many categories are present for a given feature you could pass this in the `categories_per_feature` dict in the *Dict[int, int]* format, e.g. *{0: 3, 3: 2}*. If you pass N categories this will assume the possible values for the feature are [0, . . . , N-1]. You can also explicitly pass the possible categories in the *Dict[int, List[int]]* format, e.g. *{0: [0, 1, 2], 3: [0, 55]}*. Note that the categories can be arbitrary *int* values.
- `preprocess_x_ref`: Whether to already count and store the number of instances for each possible category of each categorical variable of the reference data `x_ref` when initializing the detector. If a preprocessing step is specified, the step will be applied first. Defaults to *True*. It is possible that it needs to be set to *False* if the preprocessing step requires statistics from both the reference and test data.
- `update_x_ref`: Reference data can optionally be updated to the last N instances seen by the detector or via reservoir sampling with size N. For the former, the parameter equals *{'last': N}* while for reservoir sampling *{'reservoir_sampling': N}* is passed.

- `preprocess_fn`: Function to preprocess the data before computing the data drift metrics. Typically a dimensionality reduction technique.
- `correction`: Correction type for multivariate data. Either *'bonferroni'* or *'fdr'* (False Discovery Rate).
- `alternative`: Defines the alternative hypothesis for the K-S tests. Options are *'two-sided'* (default), *'less'* or *'greater'*. Make sure to use *'two-sided'* when mixing categorical and numerical features.
- `n_features`: Number of features used in the K-S and Chi-Squared tests. No need to pass it if no preprocessing takes place. In case of a preprocessing step, this can also be inferred automatically but could be more expensive to compute.
- `data_type`: can specify data type added to metadata. E.g. *'tabular'*.

Initialized drift detector example:

```
from alibi_detect.cd import TabularDrift

cd = TabularDrift(x_ref, p_val=0.05, categories_per_feature={0: None, 3: None})
```

22.2.2 Detect Drift

We detect data drift by simply calling `predict` on a batch of instances `x`. We can return the feature-wise p-values before the multivariate correction by setting `return_p_val` to *True*. The drift can also be detected at the feature level by setting `drift_type` to *'feature'*. No multivariate correction will take place since we return the output of *n_features* univariate tests. For drift detection on all the features combined with the correction, use *'batch'*. `return_p_val` equal to *True* will also return the threshold used by the detector (either for the univariate case or after the multivariate correction).

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_drift`: 1 if the sample tested has drifted from the reference data and 0 otherwise.
- `p_val`: contains feature-level p-values if `return_p_val` equals *True*.
- `threshold`: for feature-level drift detection the threshold equals the p-value used for the significance of the K-S and Chi-Squared tests. Otherwise the threshold after the multivariate correction (either *bonferroni* or *fdr*) is returned.
- `distance`: feature-wise K-S or Chi-Squared statistics between the reference data and the new batch if `return_distance` equals *True*.

```
preds = cd.predict(x, drift_type='batch', return_p_val=True, return_distance=True)
```

22.2.3 Saving and loading

The drift detectors can be saved and loaded in the same way as other detectors:

```
from alibi_detect.utils import save_detector, load_detector

filepath = 'my_path'
save_detector(cd, filepath)
cd = load_detector(filepath)
```

22.3 Examples

Drift detection on income prediction

source

CHAPTER

TWENTYTHREE

MODEL UNCERTAINTY

23.1 Overview

Model-uncertainty drift detectors aim to directly detect drift that's likely to effect the performance of a model of interest. The approach is to test for change in the number of instances falling into regions of the input space on which the model is uncertain in its predictions. For each instance in the reference set the detector obtains the model's prediction and some associated notion of uncertainty. For example for a classifier this may be the entropy of the predicted label probabilities or for a regressor with dropout layers dropout Monte Carlo can be used to provide a notion of uncertainty. The same is done for the test set and if significant differences in uncertainty are detected (via a Kolmogorov-Smirnoff test) then drift is flagged. The detector's reference set should be disjoint from the model's training set (on which the model's confidence may be higher).

`ClassifierUncertaintyDrift` should be used with classification models whereas `RegressorUncertaintyDrift` should be used with regression models. They are used in much the same way.

By default `ClassifierUncertaintyDrift` uses `uncertainty_type='entropy'` as the notion of uncertainty for classifier predictions and a Kolmogorov-Smirnov two-sample test is performed on these continuous values. However `uncertainty_type='margin'` can also be specified to deem the classifier's prediction uncertain if they fall within a margin (e.g. in [0.45,0.55] for binary classifier probabilities) (similar to Sethi and Kantardzic (2017)) and a Chi-Squared two-sample test is performed on these 0-1 flags of uncertainty.

By default `RegressorUncertaintyDrift` uses `uncertainty_type='mc_dropout'` and assumes a PyTorch or TensorFlow model with dropout layers as the regressor. This evaluates the model under multiple dropout configurations and uses the variation as the notion of uncertainty. Alternatively a model that outputs (for each instance) a vector of independent model predictions can be passed and `uncertainty_type='ensemble'` can be specified. Again the variation is taken as the notion of uncertainty and in both cases a Kolmogorov-Smirnov two-sample test is performed on the continuous notions of uncertainty.

23.2 Usage

23.2.1 Initialize

Arguments:

- `x_ref`: Data used as reference distribution. Should be disjoint from the model's training set
- `model`: The model of interest whose performance we'd like to remain constant.

Keyword arguments:

- `p_val`: p-value used for the significance of the test.

- `update_x_ref`: Reference data can optionally be updated to the last N instances seen by the detector or via reservoir sampling with size N. For the former, the parameter equals *{'last': N}* while for reservoir sampling *{'reservoir_sampling': N}* is passed.
- `data_type`: Optionally specify the data type (e.g. tabular, image or time-series). Added to metadata.

`ClassifierUncertaintyDrift`-specific keyword arguments:

- `preds_type`: Type of prediction output by the model. Options are 'probs' (in [0,1]) or 'logits' (in [-inf,inf]).
- `uncertainty_type`: Method for determining the model's uncertainty for a given instance. Options are 'entropy' or 'margin'.
- `margin_width`: Width of the margin if uncertainty_type = 'margin'. The model is considered uncertain on an instance if the highest two class probabilities it assigns to the instance differ by less than this.

`RegressorUncertaintyDrift`-specific keyword arguments:

- `uncertainty_type`: Method for determining the model's uncertainty for a given instance. Options are 'mc_dropout' or 'ensemble'. For the former the model should have dropout layers and output a scalar per instance. For the latter the model should output a vector of predictions per instance.
- `n_evals`: The number of times to evaluate the model under different dropout configurations. Only relavent when using the 'mc_dropout' uncertainty type.

Additional arguments if batch prediction required:

- `backend`: Framework that was used to define model. Options are 'tensorflow' or 'pytorch'.
- `batch_size`: Batch size to use to evaluate model. Defaults to 32.
- `device`: Device type to use. The default None tries to use the GPU and falls back on CPU if needed. Can be specified by passing either 'cuda', 'gpu' or 'cpu'. Only relevant for 'pytorch' backend.

Additional arguments for NLP models

- `tokenizer`: Tokenizer to use before passing data to model.
- `max_len`: Max length to be used by tokenizer.

23.2.2 Examples

Drift detector for a **TensorFlow** classifier outputting probabilities:

```
from alibi_detect.cd import ClassifierUncertaintyDrift

clf =  # tensorflow classifier model
cd = ClassifierUncertaintyDetector(x_ref, clf, backend='tensorflow', p_val=.05, preds_
↪type='probs')
```

Drift detector for a **PyTorch** regressor (with dropout layers) outputting scalars:

```
from alibi_detect.cd import RegressorUncertaintyDrift

reg =  # pytorch regression model with at least 1 dropout layer
cd = RegressorUncertaintyDrift(x_ref, reg, backend='pytorch', p_val=.05, uncertainty_
↪type='mc_dropout')
```

23.2.3 Detect Drift

We detect data drift by simply calling `predict` on a batch of instances `x`. `return_p_val` equal to *True* will also return the p-value of the test and `return_distance` equal to *True* will return the test-statistic.

The prediction takes the form of a dictionary with `meta` and `data` keys. `meta` contains the detector's metadata while `data` is also a dictionary which contains the actual predictions stored in the following keys:

- `is_drift`: 1 if the sample tested has drifted from the reference data and 0 otherwise.
- `threshold`: the user-defined threshold defining the significance of the test.
- `p_val`: the p-value of the test if `return_p_val` equals *True*.
- `distance`: the test-statistic if `return_distance` equals *True*.

```
preds = cd.predict(x)
```

23.3 Examples

Drift detection on CIFAR10 and Wine Quality Data Set

CHAPTER

TWENTYFOUR

OUTLIER, ADVERSARIAL AND DRIFT DETECTION ON CIFAR10

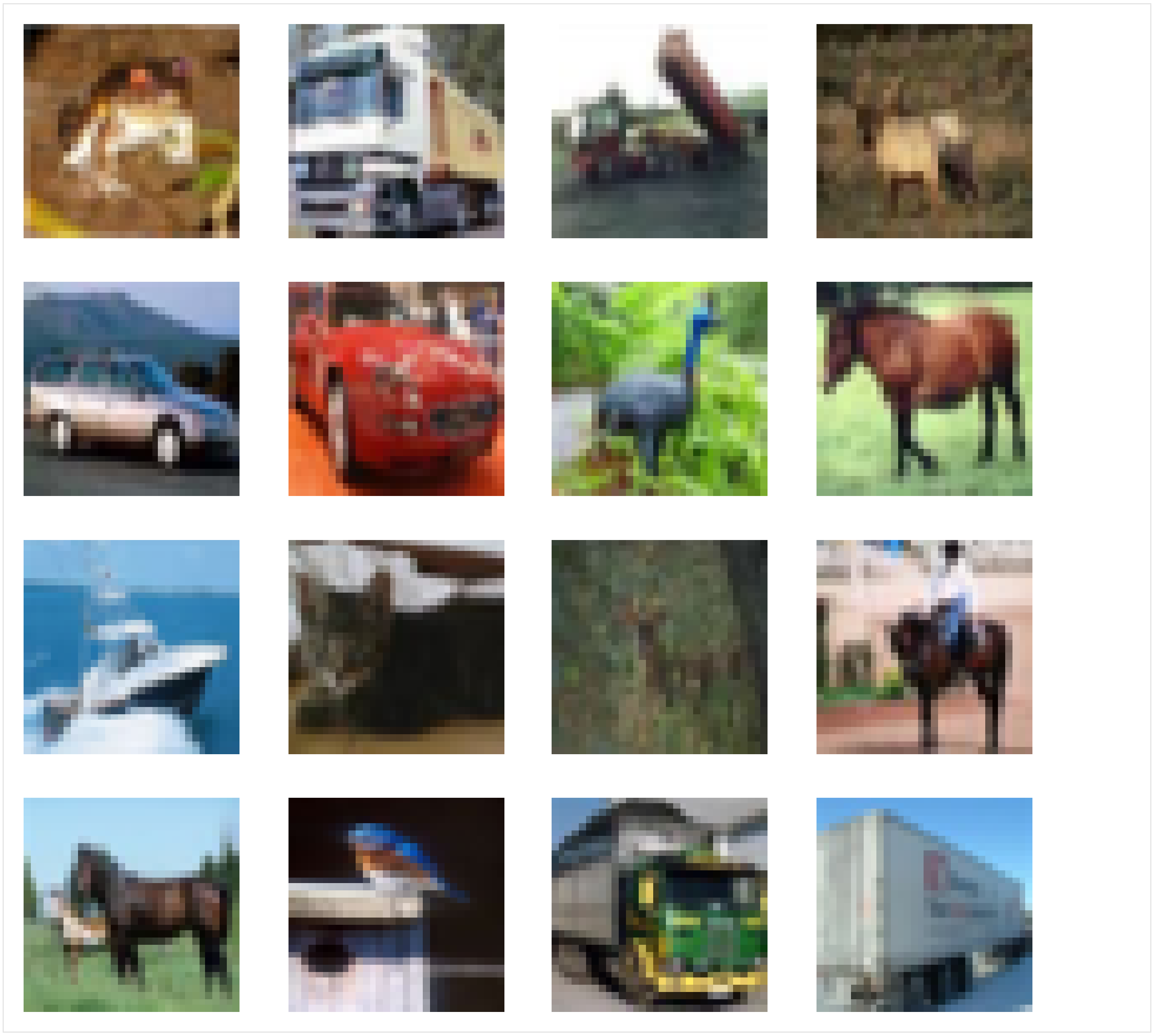
- *0. Dataset*
- *1. Outlier detection with a variational autoencoder (VAE)*
  - *Method*
  - *Load detector or train from scratch*
  - *Setting the threshold*
  - *Create and detect outliers*
  - *Deploy*
- *2. Adversarial detection by matching prediction probabilities*
  - *Method*
  - *Utility functions*
  - *Rescale data*
  - *Load pre-trained classifier*
  - *Adversarial attack*
  - *Load or train and evaluate the adversarial detector*
- *3. Drift detection with Kolmogorov-Smirnov*
  - *Method*
  - *Dataset*
  - *Detect drift*
    - *Untrained AutoEncoder*
    - *BBSDs*
  - *Leveraging the adversarial detector for malicious drift detection*
  - *Deploy*

24.1 0. Dataset

CIFAR10 consists of 60,000 32 by 32 RGB images equally distributed over 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck.

[1]:
```
# imports and plot examples
import matplotlib.pyplot as plt
%matplotlib inline
import tensorflow as tf

(X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
y_train = y_train.astype('int64').reshape(-1,)
y_test = y_test.astype('int64').reshape(-1,)
print('Train: ', X_train.shape, y_train.shape)
print('Test: ', X_test.shape, y_test.shape)

plt.figure(figsize=(10, 10))
n = 4
for i in range(n ** 2):
    plt.subplot(n, n, i + 1)
    plt.imshow(X_train[i])
    plt.axis('off')
plt.show();
```

```
Train:  (50000, 32, 32, 3) (50000,)
Test:  (10000, 32, 32, 3) (10000,)
```



24.2 1. Outlier detection with a variational autoencoder (VAE)

24.2.1 Method

In a nutshell:

- Train a VAE on *normal* data so it can reconstruct *inliers* well
- If the VAE cannot reconstruct the incoming requests well? Outlier!

More resources on VAE: paper, excellent blog post

|vae_lillog|

Image source: https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html

[2]:
```
# more imports
import logging
import numpy as np
import os

from tensorflow.keras.layers import Conv2D, Conv2DTranspose, Dense
from tensorflow.keras.layers import Flatten, Layer, Reshape, InputLayer
from tensorflow.keras.regularizers import l1

from alibi_detect.od import OutlierVAE
from alibi_detect.utils.fetching import fetch_detector
from alibi_detect.utils.perturbation import apply_mask
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.utils.visualize import plot_instance_score, plot_feature_outlier_
↪image

logger = tf.get_logger()
logger.setLevel(logging.ERROR)
```

24.2.2 Load detector or train from scratch

The pretrained outlier and adversarial detectors used in the notebook can be found here. You can use the built-in `fetch_detector` function which saves the pre-trained models in a local directory `filepath` and loads the detector. Alternatively, you can train a detector from scratch:

[3]:
```
load_pretrained = True
```

[4]:
```
filepath = os.path.join(os.getcwd(), 'outlier')

if load_pretrained:  # load pre-trained detector
    detector_type = 'outlier'
    dataset = 'cifar10'
    detector_name = 'OutlierVAE'
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
else:  # define model, initialize, train and save outlier detector

    # define encoder and decoder networks
    latent_dim = 1024
    encoder_net = tf.keras.Sequential(
      [
          InputLayer(input_shape=(32, 32, 3)),
          Conv2D(64, 4, strides=2, padding='same', activation=tf.nn.relu),
          Conv2D(128, 4, strides=2, padding='same', activation=tf.nn.relu),
          Conv2D(512, 4, strides=2, padding='same', activation=tf.nn.relu)
      ]
    )

    decoder_net = tf.keras.Sequential(
      [
          InputLayer(input_shape=(latent_dim,)),
          Dense(4*4*128),
          Reshape(target_shape=(4, 4, 128)),
          Conv2DTranspose(256, 4, strides=2, padding='same', activation=tf.nn.relu),
          Conv2DTranspose(64, 4, strides=2, padding='same', activation=tf.nn.relu),
```

(continues on next page)

(continued from previous page)

```
            Conv2DTranspose(3, 4, strides=2, padding='same', activation='sigmoid')
        ]
    )

    # initialize outlier detector
    od = OutlierVAE(
        threshold=.015,  # threshold for outlier score
        encoder_net=encoder_net,  # can also pass VAE model instead
        decoder_net=decoder_net,  # of separate encoder and decoder
        latent_dim=latent_dim
    )

    # train
    od.fit(X_train, epochs=50, verbose=False)

    # save the trained outlier detector
    save_detector(od, filepath)
```

Let's check whether the model manages to reconstruct the in-distribution training data:

[5]:
```
# plot original and reconstructed instance
idx = 8
X = X_train[idx].reshape(1, 32, 32, 3)
X_recon = od.vae(X)
plt.imshow(X.reshape(32, 32, 3)); plt.axis('off'); plt.show()
plt.imshow(X_recon.numpy().reshape(32, 32, 3)); plt.axis('off'); plt.show()
```

24.2.3 Setting the threshold

Finding good threshold values can be tricky since they are typically not easy to interpret. The `infer_threshold` method helps finding a sensible value. We need to pass a batch of instances `X` and specify what percentage of those we consider to be normal via `threshold_perc`.

[6]:
```
print('Current threshold: {}'.format(od.threshold))
od.infer_threshold(X_train, threshold_perc=99, batch_size=128)  # assume 1% of the␣
↪training data are outliers
print('New threshold: {}'.format(od.threshold))
```

```
Current threshold: 0.015
New threshold: 0.009693648591637621
```

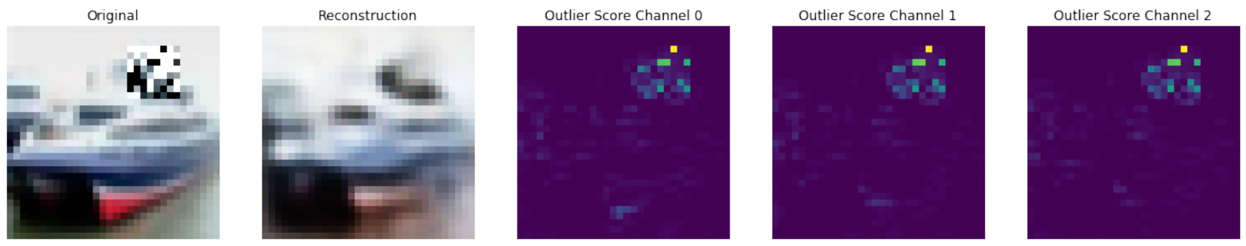
24.2.4 Create and detect outliers

We can create some outliers by applying a random noise mask to the original instances:

[7]:
```
np.random.seed(0)

i = 1

# create masked instance
x = X_test[i].reshape(1, 32, 32, 3)
x_mask, mask = apply_mask(
    x,
    mask_size=(8,8),
    n_masks=1,
    channels=[0,1,2],
    mask_type='normal',
    noise_distr=(0,1),
    clip_rng=(0,1)
)

# predict outliers and reconstructions
sample = np.concatenate([x_mask, x])
preds = od.predict(sample)
x_recon = od.vae(sample).numpy()
```

[8]:
```python
# check if outlier and visualize outlier scores
labels = ['No!', 'Yes!']
print(f"Is original outlier? {labels[preds['data']['is_outlier'][1]]}")
print(f"Is perturbed outlier? {labels[preds['data']['is_outlier'][0]]}")
plot_feature_outlier_image(preds, sample, x_recon, max_instances=1)
```

```
Is original outlier? No!
Is perturbed outlier? Yes!
```



24.2.5 Deploy the detector

For this example we use the open source deployment platform Seldon Core and eventing based project Knative which allows serverless components to be connected to event streams. The Seldon Core payload logger sends events containing model requests to the Knative broker which can farm these out to serverless components such as the outlier, drift or adversarial detection modules. Further eventing components can be added to feed off events produced by these components to send onwards to, for example, alerting or storage modules. This happens asynchronously.

|deploy_diagram|

We already configured a cluster on DigitalOcean with Seldon Core installed. The configuration steps to set everything up from scratch are detailed in this example notebook.

First we get the IP address of the Istio Ingress Gateway. This assumes Istio is installed with a LoadBalancer.

[9]:
```python
CLUSTER_IPS=!(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.
↪status.loadBalancer.ingress[0].ip}')
CLUSTER_IP=CLUSTER_IPS[0]
print(CLUSTER_IP)
```

```
188.166.139.197
```

[10]:
```python
SERVICE_HOSTNAMES=!(kubectl get ksvc vae-outlier -o jsonpath='{.status.url}' | cut -d
↪"/" -f 3)
SERVICE_HOSTNAME_VAEOD=SERVICE_HOSTNAMES[0]
print(SERVICE_HOSTNAME_VAEOD)
```

```
vae-outlier.default.example.com
```

We define some utility functions for the prediction of the deployed model.

[11]:
```python
import json
import requests
from typing import Union

classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')

def predict(x: np.ndarray) -> Union[str, list]:
```

(continues on next page)

(continued from previous page)

```
    """ Model prediction. """
    formData = {
    'instances': x.tolist()
    }
    headers = {}
    res = requests.post(
        'http://'+CLUSTER_IP+'/seldon/default/tfserving-cifar10/v1/models/resnet32/:
↪predict',
        json=formData,
        headers=headers
    )
    if res.status_code == 200:
        return classes[np.array(res.json()["predictions"])[0].argmax()]
    else:
        print("Failed with ",res.status_code)
        return []


def outlier(x: np.ndarray) -> Union[dict, list]:
    """ Outlier prediction. """
    formData = {
    'instances': x.tolist()
    }
    headers = {
        "Alibi-Detect-Return-Feature-Score": "true",
        "Alibi-Detect-Return-Instance-Score": "true"
    }
    headers["Host"] = SERVICE_HOSTNAME_VAEOD
    res = requests.post('http://'+CLUSTER_IP+'/', json=formData, headers=headers)
    if res.status_code == 200:
        od = res.json()
        od["data"]["feature_score"] = np.array(od["data"]["feature_score"])
        od["data"]["instance_score"] = np.array(od["data"]["instance_score"])
        return od
    else:
        print("Failed with ",res.status_code)
        return []


def show(x: np.ndarray) -> None:
    plt.imshow(x.reshape(32, 32, 3))
    plt.axis('off')
    plt.show()
```

Let’s make a prediction on the original instance:

[12]:
```
show(x)
predict(x)
```

[12]:
```
'ship'
```

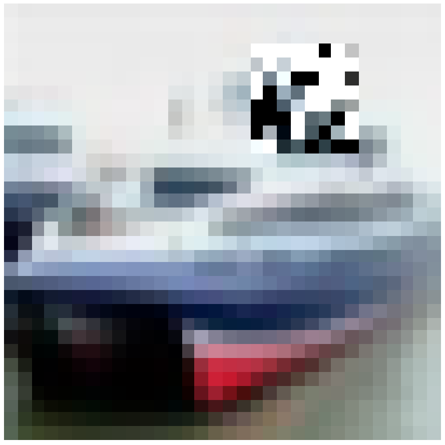
Let's check the message dumper for the output of the outlier detector:

[13]:
```
res=!kubectl logs $(kubectl get pod -l serving.knative.dev/configuration=message-
↪dumper -o jsonpath='{.items[0].metadata.name}') user-container
data = []
for i in range(0,len(res)):
    if res[i] == 'Data,':
        data.append(res[i+1])
j = json.loads(json.loads(data[0]))
print("Outlier?",labels[j["data"]["is_outlier"]==[1]])
```

```
Outlier? No!
```

We then make a prediction on the perturbed instance:

[14]:
```
show(x_mask)
predict(x_mask)
```



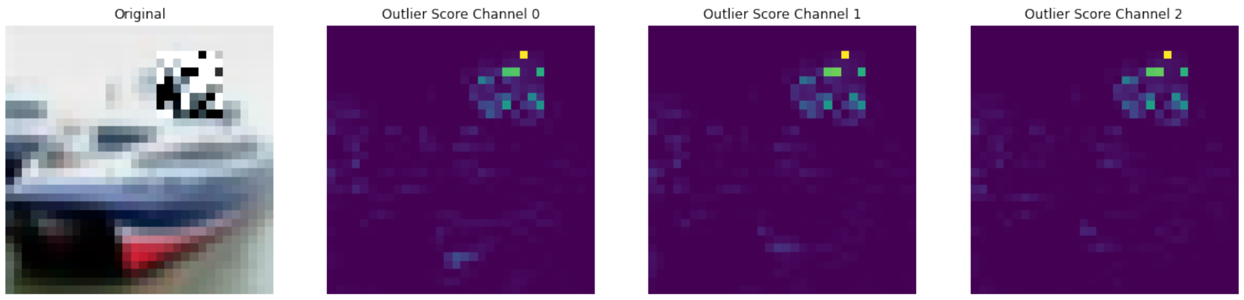
[14]:
```
'ship'
```

Although the prediction is still correct, the instance is clearly an outlier:

[15]:
```
res=!kubectl logs $(kubectl get pod -l serving.knative.dev/configuration=message-
↪dumper -o jsonpath='{.items[0].metadata.name}') user-container
data= []
for i in range(0,len(res)):
    if res[i] == 'Data,':
        data.append(res[i+1])
j = json.loads(json.loads(data[1]))
print("Outlier?",labels[j["data"]["is_outlier"]==[1]])
```

```
Outlier? Yes!
```

[16]:
```
preds = outlier(x_mask)
plot_feature_outlier_image(preds, x_mask, X_recon=None)
```



24.3 2. Adversarial detection by matching prediction probabilities

24.3.1 Method

The adversarial detector is based on Adversarial Detection and Correction by Matching Prediction Distributions. Usually, autoencoders are trained to find a transformation T that reconstructs the input instance x as accurately as possible with loss functions that are suited to capture the similarities between x and x' such as the mean squared reconstruction error. The novelty of the adversarial autoencoder (AE) detector relies on the use of a classification model-dependent loss function based on a distance metric in the output space of the model to train the autoencoder network. Given a classification model M we optimise the weights of the autoencoder such that the KL-divergence between the model predictions on x and on x' is minimised. Without the presence of a reconstruction loss term x' simply tries to make sure that the prediction probabilities $M(x')$ and $M(x)$ match without caring about the proximity of x' to x. As a result, x' is allowed to live in different areas of the input feature space than x with different decision boundary shapes with respect to the model M. The carefully crafted adversarial perturbation which is effective around x does not transfer to the new location of x' in the feature space, and the attack is therefore neutralised. Training of the autoencoder is unsupervised since we only need access to the model prediction probabilities and the normal training instances. We do not require any knowledge about the underlying adversarial attack and the classifier weights are frozen during training.

The detector can be used as follows:

- An adversarial score S is computed. S equals the K-L divergence between the model predictions on x and x'.
- If S is above a threshold (explicitly defined or inferred from training data), the instance is flagged as adversarial.
- For adversarial instances, the model M uses the reconstructed instance x' to make a prediction. If the adversarial score is below the threshold, the model makes a prediction on the original instance x.

This procedure is illustrated in the diagram below:

examples/image/adversarialae.png

The method is very flexible and can also be used to detect common data corruptions and perturbations which negatively impact the model performance.

```
[17]: # more imports
from sklearn.metrics import roc_curve, auc
from alibi_detect.ad import AdversarialAE
from alibi_detect.datasets import fetch_attack
from alibi_detect.utils.fetching import fetch_tf_model
from alibi_detect.utils.prediction import predict_batch
```

24.3.2 Utility functions

```
[18]: # instance scaling and plotting utility functions
def scale_by_instance(X: np.ndarray) -> np.ndarray:
    mean_ = X.mean(axis=(1, 2, 3)).reshape(-1, 1, 1, 1)
    std_ = X.std(axis=(1, 2, 3)).reshape(-1, 1, 1, 1)
    return (X - mean_) / std_, mean_, std_


def accuracy(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    return (y_true == y_pred).astype(int).sum() / y_true.shape[0]


def plot_adversarial(idx: list,
                     X: np.ndarray,
                     y: np.ndarray,
                     X_adv: np.ndarray,
                     y_adv: np.ndarray,
                     mean: np.ndarray,
                     std: np.ndarray,
                     score_x: np.ndarray = None,
                     score_x_adv: np.ndarray = None,
                     X_recon: np.ndarray = None,
                     y_recon: np.ndarray = None,
                     figsize: tuple = (10, 5)) -> None:

    # category map from class numbers to names
    cifar10_map = {0: 'airplane', 1: 'automobile', 2: 'bird', 3: 'cat', 4: 'deer', 5:
↪'dog',
                   6: 'frog', 7: 'horse', 8: 'ship', 9: 'truck'}

    nrows = len(idx)
    ncols = 3 if isinstance(X_recon, np.ndarray) else 2
    fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)

    n_subplot = 1
    for i in idx:

        # rescale images in [0, 1]
```

(continues on next page)

(continued from previous page)

```
        X_adj = (X[i] * std[i] + mean[i]) / 255
        X_adv_adj = (X_adv[i] * std[i] + mean[i]) / 255
        if isinstance(X_recon, np.ndarray):
            X_recon_adj = (X_recon[i] * std[i] + mean[i]) / 255

        # original image
        plt.subplot(nrows, ncols, n_subplot)
        plt.axis('off')
        if i == idx[0]:
            if isinstance(score_x, np.ndarray):
                plt.title('CIFAR-10 Image \n{}: {:.3f}'.format(cifar10_map[y[i]],␣
↪score_x[i]))
            else:
                plt.title('CIFAR-10 Image \n{}'.format(cifar10_map[y[i]]))
        else:
            if isinstance(score_x, np.ndarray):
                plt.title('{}: {:.3f}'.format(cifar10_map[y[i]], score_x[i]))
            else:
                plt.title('{}'.format(cifar10_map[y[i]]))
        plt.imshow(X_adj)
        n_subplot += 1

        # adversarial image
        plt.subplot(nrows, ncols, n_subplot)
        plt.axis('off')
        if i == idx[0]:
            if isinstance(score_x_adv, np.ndarray):
                plt.title('Adversarial \n{}: {:.3f}'.format(cifar10_map[y_adv[i]],␣
↪score_x_adv[i]))
            else:
                plt.title('Adversarial \n{}'.format(cifar10_map[y_adv[i]]))
        else:
            if isinstance(score_x_adv, np.ndarray):
                plt.title('{}: {:.3f}'.format(cifar10_map[y_adv[i]], score_x_adv[i]))
            else:
                plt.title('{}'.format(cifar10_map[y_adv[i]]))
        plt.imshow(X_adv_adj)
        n_subplot += 1

        # reconstructed image
        if isinstance(X_recon, np.ndarray):
            plt.subplot(nrows, ncols, n_subplot)
            plt.axis('off')
            if i == idx[0]:
                plt.title('AE Reconstruction \n{}'.format(cifar10_map[y_recon[i]]))
            else:
                plt.title('{}'.format(cifar10_map[y_recon[i]]))
            plt.imshow(X_recon_adj)
            n_subplot += 1

    plt.show()


def plot_roc(roc_data: dict, figsize: tuple = (10,5)):
    plot_labels = []
    scores_attacks = []
    labels_attacks = []
```

(continues on next page)

(continued from previous page)

```
    for k, v in roc_data.items():
        if 'original' in k:
            continue
        score_x = roc_data[v['normal']]['scores']
        y_pred = roc_data[v['normal']]['predictions']
        score_v = v['scores']
        y_pred_v = v['predictions']
        labels_v = np.ones(score_x.shape[0])
        idx_remove = np.where(y_pred == y_pred_v)[0]
        labels_v = np.delete(labels_v, idx_remove)
        score_v = np.delete(score_v, idx_remove)
        scores = np.concatenate([score_x, score_v])
        labels = np.concatenate([np.zeros(y_pred.shape[0]), labels_v]).astype(int)
        scores_attacks.append(scores)
        labels_attacks.append(labels)
        plot_labels.append(k)

    for sc_att, la_att, plt_la in zip(scores_attacks, labels_attacks, plot_labels):
        fpr, tpr, thresholds = roc_curve(la_att, sc_att)
        roc_auc = auc(fpr, tpr)
        label = str('{}: AUC = {:.2f}'.format(plt_la, roc_auc))
        plt.plot(fpr, tpr, lw=1, label='{}: AUC={:.4f}'.format(plt_la, roc_auc))

    plt.plot([0, 1], [0, 1], color='black', lw=1, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('{}'.format('ROC curve'))
    plt.legend(loc="lower right", ncol=1)
    plt.grid()
    plt.show()
```

24.3.3 Rescale data

The ResNet classification model is trained on data standardized by instance:

```
[19]: # rescale data
X_train, mean_train, std_train = scale_by_instance(X_train * 255.)
X_test, mean_test, std_test = scale_by_instance(X_test * 255.)
scale = (mean_train, std_train), (mean_test, std_test)
```

24.3.4 Load pre-trained classifier

```
[20]: dataset = 'cifar10'
model = 'resnet56'
clf = fetch_tf_model(dataset, model)
```

```
/home/avl/anaconda3/envs/detect/lib/python3.7/site-packages/tensorflow_core/python/
↪keras/layers/core.py:901: UserWarning:

official.vision.image_classification.resnet_cifar_model is not loaded, but a Lambda␣
↪layer uses it. It may cause errors.
```

Check the predictions on the test:

[21]:
```
y_pred = predict_batch(clf, X_test, batch_size=32, return_class=True)
acc_y_pred = accuracy(y_test, y_pred)
print('Accuracy: {:.4f}'.format(acc_y_pred))
```

```
Accuracy: 0.9315
```

24.3.5 Adversarial attack

We investigate both Carlini-Wagner (C&W) and SLIDE attacks. You can simply load previously found adversarial instances on the pretrained ResNet-56 model. The attacks are generated by using Foolbox:

[22]:
```
# C&W attack
data_cw = fetch_attack(dataset, model, 'cw')
X_train_cw, X_test_cw = data_cw['data_train'], data_cw['data_test']
meta_cw = data_cw['meta'] # metadata with hyperparameters of the attack
# SLIDE attack
data_slide = fetch_attack(dataset, model, 'slide')
X_train_slide, X_test_slide = data_slide['data_train'], data_slide['data_test']
meta_slide = data_slide['meta']
```

We can verify that the accuracy of the classifier drops to almost 0%:

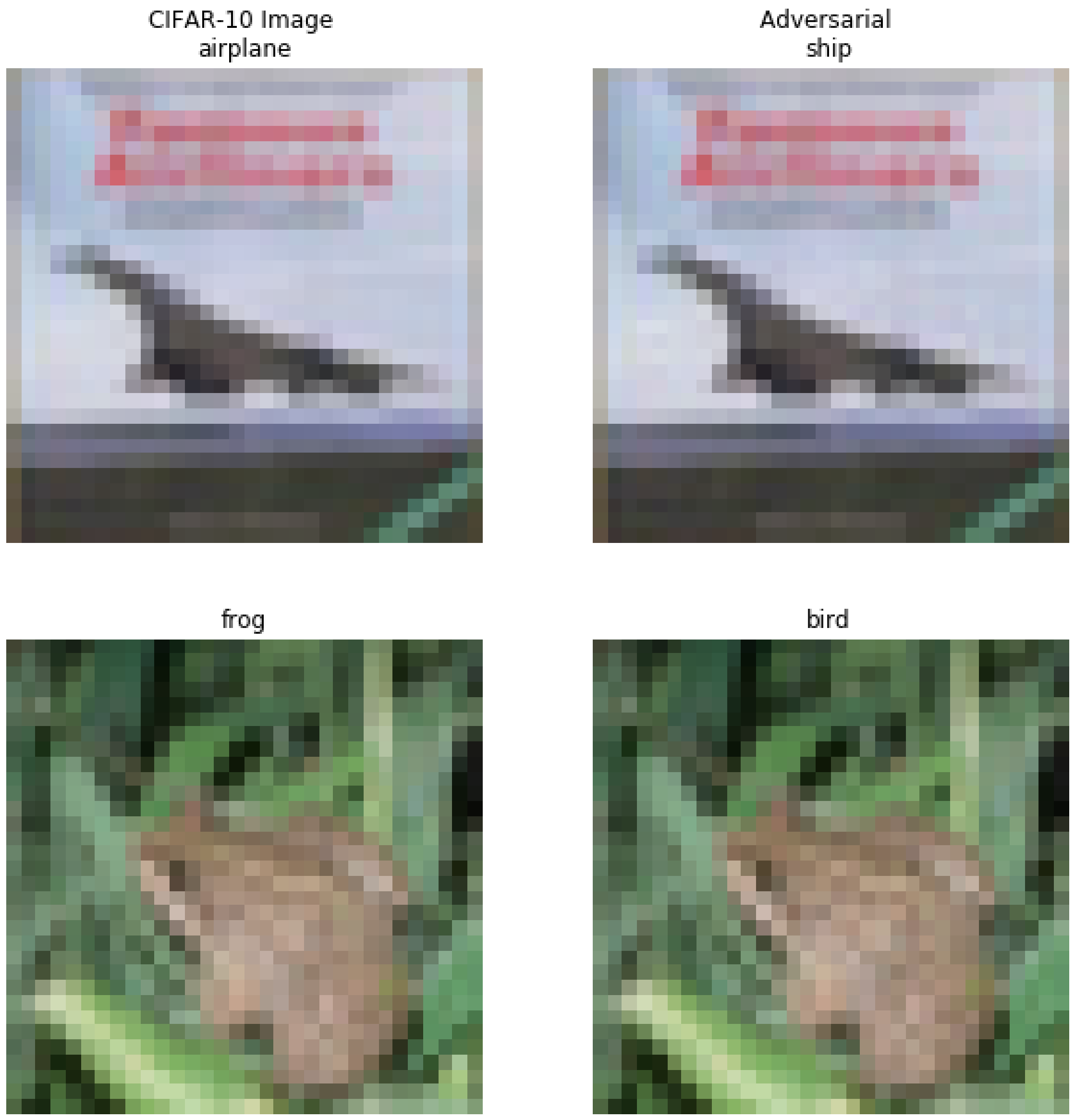
[23]:
```
y_pred_cw = predict_batch(clf, X_test_cw, batch_size=32, return_class=True)
y_pred_slide = predict_batch(clf, X_test_slide, batch_size=32, return_class=True)
acc_y_pred_cw = accuracy(y_test, y_pred_cw)
acc_y_pred_slide = accuracy(y_test, y_pred_slide)
print('Accuracy: cw {:.4f} -- SLIDE {:.4f}'.format(acc_y_pred_cw, acc_y_pred_slide))
```

```
Accuracy: cw 0.0000 -- SLIDE 0.0001
```

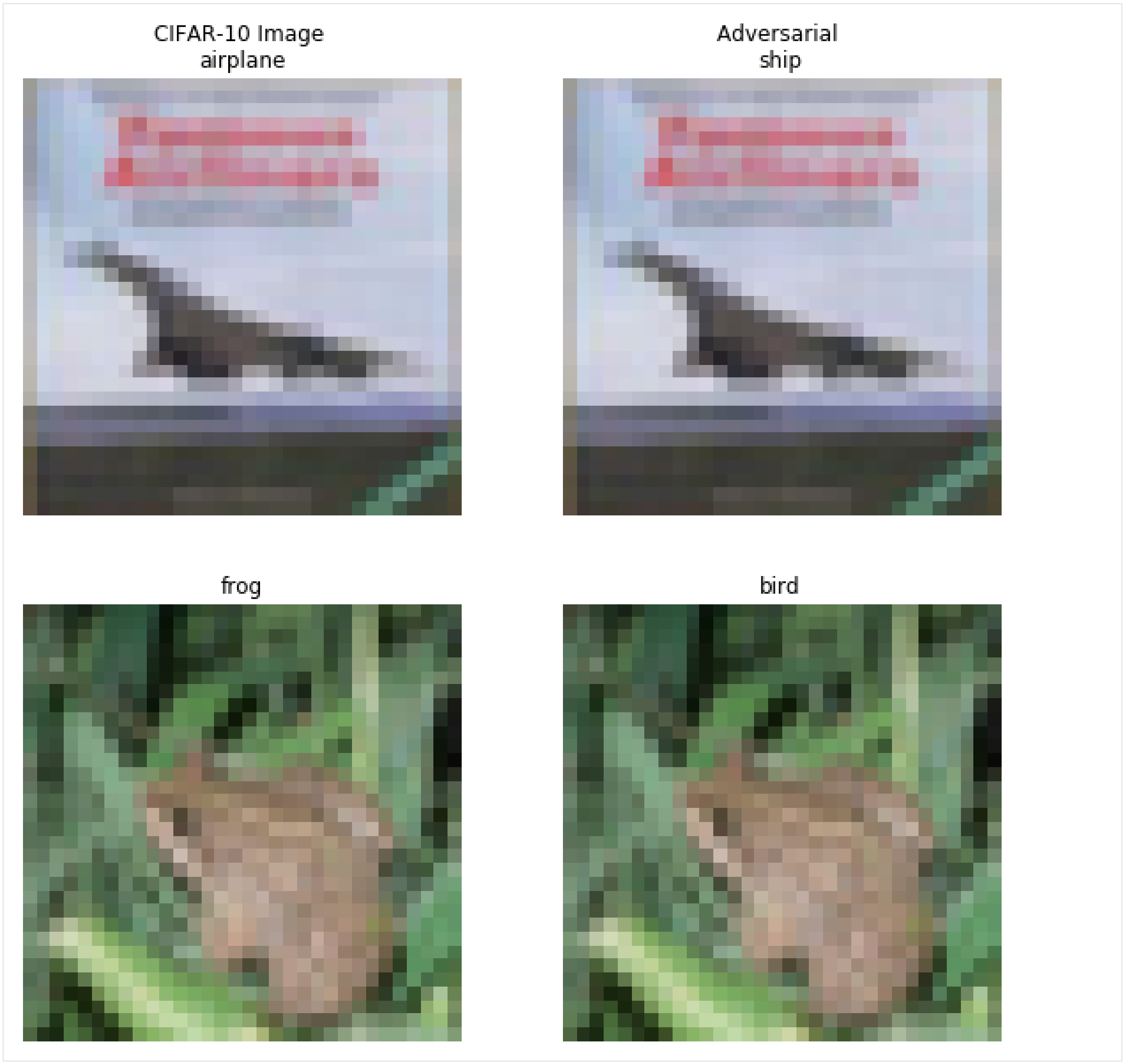
Let's visualise some adversarial instances:

[24]:
```
# plot attacked instances
idx = [3, 4]
print('C&W attack...')
plot_adversarial(idx, X_test, y_pred, X_test_cw, y_pred_cw,
                 mean_test, std_test, figsize=(10, 10))
print('SLIDE attack...')
plot_adversarial(idx, X_test, y_pred, X_test_slide, y_pred_slide,
                 mean_test, std_test, figsize=(10, 10))
```

```
C&W attack...
```



```
SLIDE attack...
```



24.3.6 Load or train and evaluate the adversarial detector

We can again either fetch the pretrained detector from a Google Cloud Bucket or train one from scratch:

[25]:
```
load_pretrained = True
```

[26]:
```
filepath = os.path.join(os.getcwd(), 'adversarial')

if load_pretrained:
    detector_type = 'adversarial'
    detector_name = 'base'
    ad = fetch_detector(filepath, detector_type, dataset, detector_name, model=model)
    filepath = os.path.join(filepath, detector_name)
else:  # train detector from scratch
```

(continues on next page)

(continued from previous page)

```
    # define encoder and decoder networks
    encoder_net = tf.keras.Sequential(
            [
                InputLayer(input_shape=(32, 32, 3)),
                Conv2D(32, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2D(64, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2D(256, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Flatten(),
                Dense(40)
            ]
        )

    decoder_net = tf.keras.Sequential(
        [
                InputLayer(input_shape=(40,)),
                Dense(4 * 4 * 128, activation=tf.nn.relu),
                Reshape(target_shape=(4, 4, 128)),
                Conv2DTranspose(256, 4, strides=2, padding='same',
                                activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2DTranspose(64, 4, strides=2, padding='same',
                                activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2DTranspose(3, 4, strides=2, padding='same',
                                activation=None, kernel_regularizer=l1(1e-5))
            ]
        )

    # initialise and train detector
    ad = AdversarialAE(
        encoder_net=encoder_net,
        decoder_net=decoder_net,
        model=clf
    )
    ad.fit(X_train, epochs=40, batch_size=64, verbose=True)

    # save the trained adversarial detector
    save_detector(ad, filepath)
```

```
WARNING:alibi_detect.ad.adversarialae:No threshold level set. Need to infer threshold␣
↪using `infer_threshold`.
```

The detector first reconstructs the input instances which can be adversarial. The reconstructed input is then fed to the classifier to compute the adversarial score. If the score is above a threshold, the instance is classified as adversarial and the detector tries to correct the attack. Let's investigate what happens when we reconstruct attacked instances and make predictions on them:

[27]:
```
X_recon_cw = predict_batch(ad.ae, X_test_cw, batch_size=32)
X_recon_slide = predict_batch(ad.ae, X_test_slide, batch_size=32)
```

[28]:
```
y_recon_cw = predict_batch(clf, X_recon_cw, batch_size=32, return_class=True)
y_recon_slide = predict_batch(clf, X_recon_slide, batch_size=32, return_class=True)
```

Accuracy on attacked vs. reconstructed instances:

[29]:
```
acc_y_recon_cw = accuracy(y_test, y_recon_cw)
acc_y_recon_slide = accuracy(y_test, y_recon_slide)
print('Accuracy after C&W attack {:.4f} -- reconstruction {:.4f}'.format(acc_y_pred_
↪cw, acc_y_recon_cw))
print('Accuracy after SLIDE attack {:.4f} -- reconstruction {:.4f}'.format(acc_y_pred_
↪slide, acc_y_recon_slide))
```

```
Accuracy after C&W attack 0.0000 -- reconstruction 0.8048
Accuracy after SLIDE attack 0.0001 -- reconstruction 0.8159
```

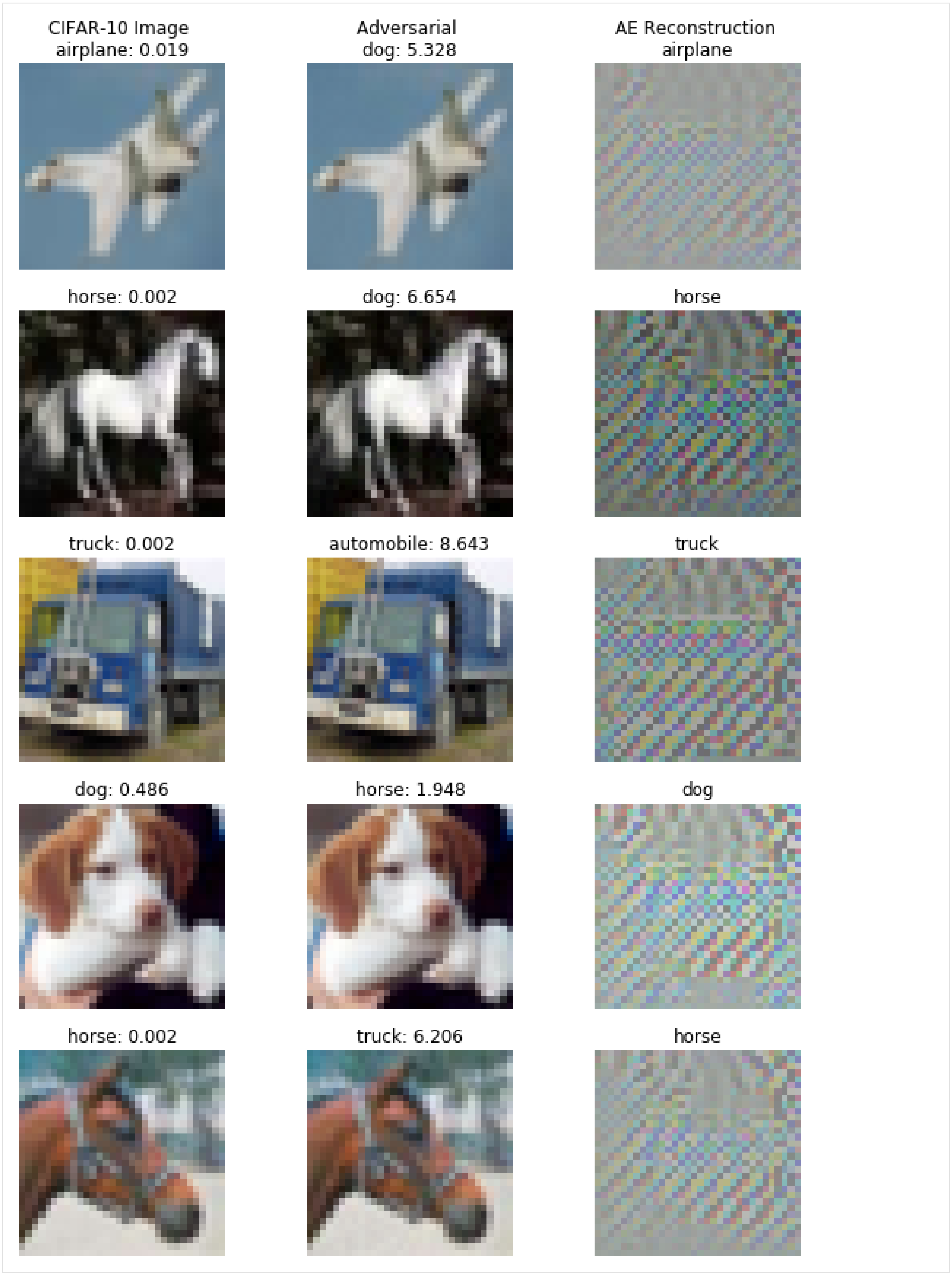
The detector restores the accuracy after the attacks from almost 0% to well over 80%! We can compute the adversarial scores and inspect some of the reconstructed instances:

[30]:
```
score_x = ad.score(X_test, batch_size=32)
score_cw = ad.score(X_test_cw, batch_size=32)
score_slide = ad.score(X_test_slide, batch_size=32)
```

[31]:
```
# visualize original, attacked and reconstructed instances with adversarial scores
print('C&W attack...')
idx = [10, 13, 14, 16, 17]
plot_adversarial(idx, X_test, y_pred, X_test_cw, y_pred_cw, mean_test, std_test,
                 score_x=score_x, score_x_adv=score_cw, X_recon=X_recon_cw,
                 y_recon=y_recon_cw, figsize=(10, 15))
print('SLIDE attack...')
idx = [23, 25, 27, 29, 34]
plot_adversarial(idx, X_test, y_pred, X_test_slide, y_pred_slide, mean_test, std_test,
                 score_x=score_x, score_x_adv=score_slide, X_recon=X_recon_slide,
                 y_recon=y_recon_slide, figsize=(10, 15))
```

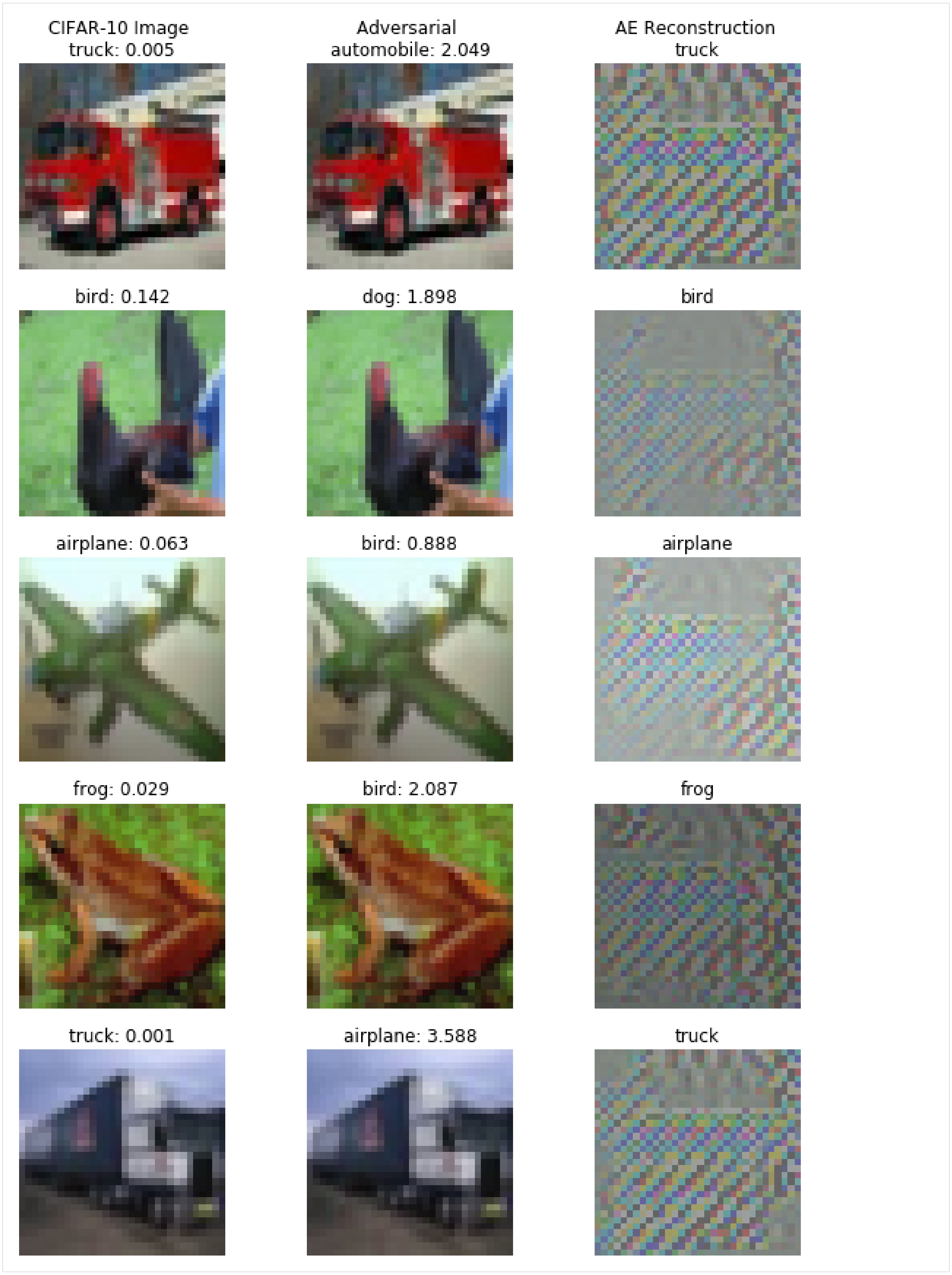
```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB␣
↪data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB␣
↪data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB␣
↪data ([0..1] for floats or [0..255] for integers).
```

```
C&W attack...
```


CIFAR-10 Image
airplane: 0.019
Adversarial
dog: 5.328
AE Reconstruction
airplane
horse: 0.002
dog: 6.654
horse
truck: 0.002
automobile: 8.643
truck
dog: 0.486
horse: 1.948
dog
horse: 0.002
truck: 6.206
horse

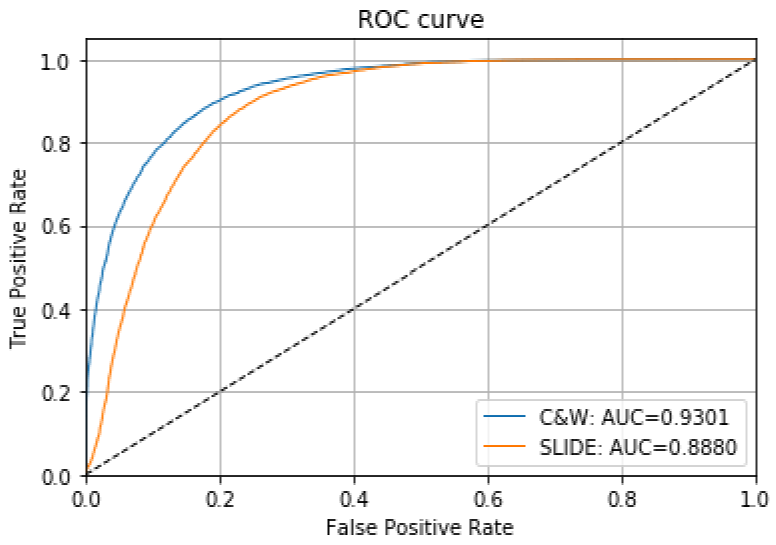
```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB␣
↪data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB␣
↪data ([0..1] for floats or [0..255] for integers).
```

```
SLIDE attack...
```


CIFAR-10 Image
truck: 0.005
Adversarial
automobile: 2.049
AE Reconstruction
truck
bird: 0.142
dog: 1.898
bird
airplane: 0.063
bird: 0.888
airplane
frog: 0.029
bird: 2.087
frog
truck: 0.001
airplane: 3.588
truck

The ROC curves and AUC values show the effectiveness of the adversarial score to detect adversarial instances:

```
[32]: # plot roc curve
roc_data = {
    'original': {'scores': score_x, 'predictions': y_pred},
    'C&W': {'scores': score_cw, 'predictions': y_pred_cw, 'normal': 'original'},
    'SLIDE': {'scores': score_slide, 'predictions': y_pred_slide, 'normal': 'original
↪'}
}

plot_roc(roc_data)
```



The threshold for the adversarial score can be set via `infer_threshold`. We need to pass a batch of instances X and specify what percentage of those we consider to be normal via `threshold_perc`. Assume we have only normal instances some of which the model has misclassified leading to a higher score if the reconstruction picked up features from the correct class or some might look adversarial in the first place. As a result, we set our threshold at 95%:

```
[33]: ad.infer_threshold(X_test, threshold_perc=95, margin=0., batch_size=32)
print('Adversarial threshold: {:.4f}'.format(ad.threshold))
```

```
Adversarial threshold: 2.6722
```

The `correct` method of the detector executes the diagram in Figure 1. First the adversarial scores is computed. For instances where the score is above the threshold, the classifier prediction on the reconstructed instance is returned. Otherwise the original prediction is kept. The method returns a dictionary containing the metadata of the detector, whether the instances in the batch are adversarial (above the threshold) or not, the classifier predictions using the correction mechanism and both the original and reconstructed predictions. Let's illustrate this on a batch containing some adversarial (C&W) and original test set instances:

```
[34]: n_test = X_test.shape[0]
np.random.seed(0)
idx_normal = np.random.choice(n_test, size=1600, replace=False)
idx_cw = np.random.choice(n_test, size=400, replace=False)

X_mix = np.concatenate([X_test[idx_normal], X_test_cw[idx_cw]])
y_mix = np.concatenate([y_test[idx_normal], y_test[idx_cw]])
print(X_mix.shape, y_mix.shape)
```

```
(2000, 32, 32, 3) (2000,)
```

Let's check the model performance:

[35]:
```
y_pred_mix = predict_batch(clf, X_mix, batch_size=32, return_class=True)
acc_y_pred_mix = accuracy(y_mix, y_pred_mix)
print('Accuracy {:.4f}'.format(acc_y_pred_mix))
```

```
Accuracy 0.7380
```

This can be improved with the correction mechanism:

[36]:
```
preds = ad.correct(X_mix, batch_size=32)
acc_y_corr_mix = accuracy(y_mix, preds['data']['corrected'])
print('Accuracy {:.4f}'.format(acc_y_corr_mix))
```

```
Accuracy 0.8205
```

There are a few other tricks highlighted in the paper (**temperature scaling** and **hidden layer K-L divergence**) and implemented in Alibi Detect which can further boost the adversarial detector's performance. Check this example notebook for more details.

24.4 3. Drift detection with Kolmogorov-Smirnov

24.4.1 Method

The drift detector applies feature-wise two-sample Kolmogorov-Smirnov (K-S) tests. For multivariate data, the obtained p-values for each feature are aggregated either via the Bonferroni or the False Discovery Rate (FDR) correction. The Bonferroni correction is more conservative and controls for the probability of at least one false positive. The FDR correction on the other hand allows for an expected fraction of false positives to occur.

For high-dimensional data, we typically want to reduce the dimensionality before computing the feature-wise univariate K-S tests and aggregating those via the chosen correction method. Following suggestions in Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift, we incorporate Untrained AutoEncoders (UAE), black-box shift detection using the classifier's softmax outputs (BBSDs) and PCA as out-of-the box preprocessing methods. Preprocessing methods which do not rely on the classifier will usually pick up drift in the input data, while BBSDs focuses on label shift. The adversarial detector which is part of the library can also be transformed into a drift detector picking up drift that reduces the performance of the classification model. We can therefore combine different preprocessing techniques to figure out if there is drift which hurts the model performance, and whether this drift can be classified as input drift or label shift.

Note that the library also has a drift detector based on the Maximum Mean Discrepancy and contains drift on text functionality as well.

24.4.2 Dataset

We will use the CIFAR-10-C dataset (Hendrycks & Dietterich, 2019) to evaluate the drift detector. The instances in CIFAR-10-C come from the test set in CIFAR-10 but have been corrupted and perturbed by various types of noise, blur, brightness etc. at different levels of severity, leading to a gradual decline in the classification model performance. We also check for drift against the original test set with class imbalances.

[37]:
```
# yet again import stuff
from alibi_detect.cd import KSDrift
```

(continues on next page)

(continued from previous page)

```
from alibi_detect.cd.preprocess import UAE, HiddenOutput
from alibi_detect.datasets import fetch_cifar10c, corruption_types_cifar10c
```

We can select from the following corruption types at 5 severity levels:

[38]:
```
corruptions = corruption_types_cifar10c()
print(corruptions)
```

```
['brightness', 'contrast', 'defocus_blur', 'elastic_transform', 'fog', 'frost',
↪'gaussian_blur', 'gaussian_noise', 'glass_blur', 'impulse_noise', 'jpeg_compression
↪', 'motion_blur', 'pixelate', 'saturate', 'shot_noise', 'snow', 'spatter', 'speckle_
↪noise', 'zoom_blur']
```

Let's pick a subset of the corruptions at corruption level 5. Each corruption type consists of perturbations on all of the original test set images.

[39]:
```
corruption = ['gaussian_noise', 'motion_blur', 'brightness', 'pixelate']
X_corr, y_corr = fetch_cifar10c(corruption=corruption, severity=5, return_X_y=True)
X_corr = X_corr.astype('float32') / 255
```

We split the original test set in a reference dataset and a dataset which should not be rejected under the *H0* of the K-S test. We also split the corrupted data by corruption type:

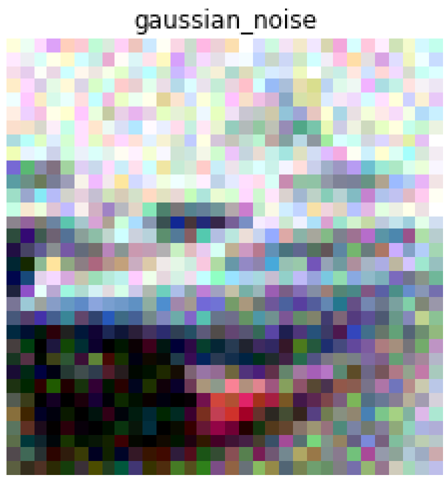
[40]:
```
np.random.seed(0)
n_test = X_test.shape[0]
idx = np.random.choice(n_test, size=n_test // 2, replace=False)
idx_h0 = np.delete(np.arange(n_test), idx, axis=0)
X_ref,y_ref = X_test[idx], y_test[idx]
X_h0, y_h0 = X_test[idx_h0], y_test[idx_h0]
print(X_ref.shape, X_h0.shape)

X_c = []
n_corr = len(corruption)
for i in range(n_corr):
    X_c.append(scale_by_instance(X_corr[i * n_test:(i + 1) * n_test])[0])
```

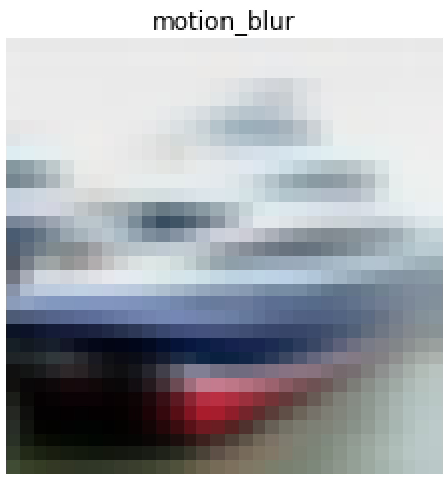
```
(5000, 32, 32, 3) (5000, 32, 32, 3)
```

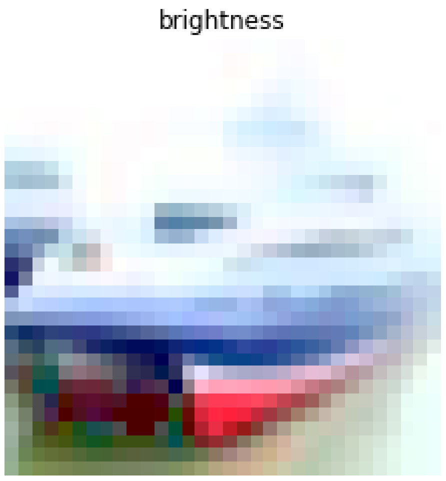
We can visualise the same instance for each corruption type:

[41]:
```
# plot original and corrupted images
i = 1

n_test = X_test.shape[0]
plt.title('Original')
plt.axis('off')
plt.imshow((X_test[i] * std_test[i] + mean_test[i]) / 255.)
plt.show()
for _ in range(len(corruption)):
    plt.title(corruption[_])
    plt.axis('off')
    plt.imshow(X_corr[n_test * _+ i])
    plt.show()
```

Original


gaussian_noise


motion_blur



We can also verify that the performance of a ResNet-32 classification model on CIFAR-10 drops significantly on this perturbed dataset:

[42]:
```python
dataset = 'cifar10'
model = 'resnet32'
clf = fetch_tf_model(dataset, model)
acc = clf.evaluate(X_test, y_test, batch_size=128, verbose=0)[1]
print('Test set accuracy:')
print('Original {:.4f}'.format(acc))
clf_accuracy = {'original': acc}
for _ in range(len(corruption)):
    acc = clf.evaluate(X_c[_], y_test, batch_size=128, verbose=0)[1]
    clf_accuracy[corruption[_]] = acc
    print('{} {:.4f}'.format(corruption[_], acc))
```

```
Test set accuracy:
Original 0.9278
gaussian_noise 0.2208
motion_blur 0.6339
brightness 0.8913
pixelate 0.3666
```

Given the drop in performance, it is important that we detect the harmful data drift!

24.4.3 Detect drift

We are trying to detect data drift on high-dimensional (*32x32x3*) data using an aggregation of univariate K-S tests. It therefore makes sense to apply dimensionality reduction first. Some dimensionality reduction methods also used in Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift are readily available: **UAE** (Untrained AutoEncoder), **BBSDs** (black-box shift detection using the classifier's softmax outputs) and **PCA**.

Untrained AutoEncoder

First we try UAE:

[43]:
```
tf.random.set_seed(0)

# define encoder
encoding_dim = 32
encoder_net = tf.keras.Sequential(
  [
      InputLayer(input_shape=(32, 32, 3)),
      Conv2D(64, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(128, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(512, 4, strides=2, padding='same', activation=tf.nn.relu),
      Flatten(),
      Dense(encoding_dim,)
  ]
)
uae = UAE(encoder_net=encoder_net)
preprocess_kwargs = {'model': uae, 'batch_size': 128}

# initialise drift detector
p_val = .05
cd = KSDrift(
    p_val=p_val,        # p-value for K-S test
    X_ref=X_ref,       # test against original test set
    preprocess_kwargs=preprocess_kwargs
)
```

Let's check whether the detector thinks drift occurred within the original test set:

[44]:
```
preds_h0 = cd.predict(X_h0, return_p_val=True)
print('Drift? {}'.format(labels[preds_h0['data']['is_drift']]))
```

```
Drift? No!
```

As expected, no drift occurred. We can also inspect the feature-wise K-S statistics, threshold value and p-values for each univariate K-S test by (encoded) feature before the multivariate correction. Most of them are well above the 0.05 threshold:

[45]:
```
# print stats for H0
print('K-S statistics:')
print(preds_h0['data']['distance'])
print(f"\nK-S statistic threshold: {preds_h0['data']['threshold']}")
print('\np-values:')
print(preds_h0['data']['p_val'])
```

```
K-S statistics:
[0.017  0.013  0.0124 0.0234 0.0156 0.0144 0.025  0.0144 0.0216 0.0112
 0.0122 0.0146 0.012  0.0154 0.015  0.0148 0.012  0.0158 0.0104 0.0144
```

(continues on next page)

(continued from previous page)

```
 0.0116 0.0148 0.0194 0.0112 0.0176 0.0154 0.0152 0.0158 0.0166 0.0202
 0.0208 0.0196]

K-S statistic threshold: 0.0015625

p-values:
[0.46531922 0.79201305 0.8367454  0.12939005 0.5769981  0.677735
 0.08786641 0.677735   0.19387017 0.912423   0.850771   0.66088647
 0.8642828  0.5936282  0.62716705 0.6440195  0.8642828  0.5604951
 0.94969434 0.677735   0.88960564 0.6440195  0.30355498 0.912423
 0.420929   0.5936282  0.61036026 0.5604951  0.496191   0.25943416
 0.22956407 0.2920585 ]
```

Let's now check the predictions on the perturbed data:

[46]:
```python
# print stats for corrupted data
for x, c in zip(X_c, corruption):
    preds = cd.predict(x, return_p_val=True)
    print(f'Corruption type: {c}')
    print('Drift? {}'.format(labels[preds['data']['is_drift']]))
    print('Feature-wise p-values:')
    print(preds['data']['p_val'])
    print('')
```

```
Corruption type: gaussian_noise
Drift? Yes!
Feature-wise p-values:
[1.3873853e-04 2.6511249e-01 9.4417885e-02 3.8390055e-01 7.1598392e-04
 5.9501763e-04 1.7882723e-02 6.8460542e-01 2.9989053e-02 3.7781857e-02
 6.6516680e-01 3.5206401e-07 4.4194090e-05 7.9351515e-03 6.5543061e-01
 1.0220607e-04 1.9904150e-01 8.7863362e-01 1.1572546e-02 8.7863362e-01
 2.7950103e-03 1.1150700e-02 3.6837953e-01 4.4130555e-01 7.8937606e-05
 5.1594800e-03 6.2871156e-03 1.3455943e-03 9.7143359e-02 1.4105450e-07
 1.7258449e-02 1.4648294e-01]

Corruption type: motion_blur
Drift? Yes!
Feature-wise p-values:
[3.75149103e-13 1.34077845e-02 2.96938539e-01 1.38920277e-01
 5.03090501e-01 1.71172902e-01 6.25065863e-02 5.15948003e-03
 4.85032678e-01 2.16893386e-03 7.48323351e-02 1.28741434e-03
 1.24597345e-02 2.57197134e-05 1.32995670e-09 7.83811294e-10
 7.93515146e-03 3.10395747e-01 2.09074125e-01 5.97174764e-01
 5.95017627e-04 9.71433595e-02 3.38559955e-01 3.31362933e-01
 7.15983915e-04 3.45860541e-01 5.68405211e-01 2.43121485e-06
 5.30714750e-01 2.57197134e-05 1.80098459e-01 8.85808766e-01]

Corruption type: brightness
Drift? Yes!
Feature-wise p-values:
[4.8499879e-07 1.3873853e-04 1.1572546e-02 1.6691783e-10 2.7127105e-01
 2.5326056e-02 1.6036824e-03 4.6618203e-05 2.1745481e-04 1.2474350e-01
 6.2343346e-14 3.1785589e-08 8.9277834e-01 3.8390055e-01 8.2942984e-12
 8.9514272e-09 2.4149875e-01 3.5845602e-03 2.2287663e-29 5.2143931e-01
 2.2484422e-01 2.7950277e-16 1.9522957e-05 1.0870906e-01 7.6357084e-03
 3.2130507e-04 3.4248088e-02 4.7411124e-14 2.5942018e-18 1.5351619e-03
 4.6728885e-01 2.2888689e-06]
```

(continues on next page)

(continued from previous page)

```
Corruption type: pixelate
Drift? Yes!
Feature-wise p-values:
[3.0361506e-01 8.5970649e-04 1.6684313e-01 7.0512637e-02 5.0309050e-01
 7.0512637e-02 2.5905645e-01 2.2484422e-01 8.9953583e-01 3.5326433e-01
 5.4005289e-01 3.3136293e-01 2.1316923e-02 3.9982069e-01 1.4057782e-06
 7.9374194e-02 3.7114436e-04 7.7923602e-01 2.7127105e-01 2.5905645e-01
 8.9953583e-01 1.6260068e-01 2.8040027e-02 9.1756582e-02 1.9192154e-02
 1.7117290e-01 7.1357483e-01 2.9989053e-02 3.6077085e-01 1.2459734e-02
 4.2442977e-01 2.6461019e-04]
```

BBSDs

For **BBSDs**, we use the classifier's softmax outputs for black-box shift detection. This method is based on Detecting and Correcting for Label Shift with Black Box Predictors.

Here we use the output of the softmax layer to detect the drift, but other hidden layers can be extracted as well by setting *'layer'* to the index of the desired hidden layer in the model:

[47]:
```
# use output softmax layer
preprocess_kwargs = {'model': HiddenOutput(model=clf, layer=-1), 'batch_size': 128}

cd = KSDrift(
    p_val=p_val,
    X_ref=X_ref,
    preprocess_kwargs=preprocess_kwargs
)
```

There is again no drift on the original held out test set:

[48]:
```
preds_h0 = cd.predict(X_h0)
print('Drift? {}'.format(labels[preds_h0['data']['is_drift']]))
print('\np-values:')
print(preds_h0['data']['p_val'])
```

```
Drift? No!

p-values:
[0.11774229 0.52796143 0.19387017 0.20236294 0.496191   0.72781175
 0.12345381 0.420929   0.8367454  0.7604178 ]
```

We compare this with the perturbed data:

[49]:
```
for x, c in zip(X_c, corruption):
    preds = cd.predict(x)
    print(f'Corruption type: {c}')
    print('Drift? {}'.format(labels[preds['data']['is_drift']]))
    print('Feature-wise p-values:')
    print(preds['data']['p_val'])
    print('')
```

```
Corruption type: gaussian_noise
Drift? Yes!
Feature-wise p-values:
```

(continues on next page)

(continued from previous page)

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

Corruption type: motion_blur
Drift? Yes!
Feature-wise p-values:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

Corruption type: brightness
Drift? Yes!
Feature-wise p-values:
[0.0000000e+00 4.2024049e-15 2.8963613e-33 4.8499879e-07 2.3718185e-15
 1.2473309e-05 2.9714003e-30 1.0611427e-09 4.6048109e-12 4.1857830e-17]

Corruption type: pixelate
Drift? Yes!
Feature-wise p-values:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

For more functionality and examples, such as updating the reference data with reservoir sampling or picking another multivariate correction mechanism, check out this example notebook.

24.4.4 Leveraging the adversarial detector for malicious drift detection

While monitoring covariate and predicted label shift is all very interesting and exciting, at the end of the day we are mainly interested in whether the drift actually hurt the model performance significantly. To this end, we can leverage the adversarial detector and measure univiariate drift on the adversarial scores!

[50]:
```
np.random.seed(0)
idx = np.random.choice(n_test, size=n_test // 2, replace=False)
X_ref = scale_by_instance(X_test[idx])[0]

cd = KSDrift(
    p_val=.05,
    X_ref=X_ref,
    preprocess_fn=ad.score,  # adversarial score fn = preprocess step
    preprocess_kwargs={'batch_size': 128}
)
```

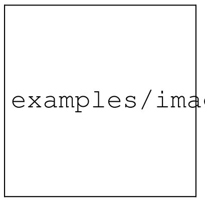
Make drift predictions on the original test set and corrupted data:

[51]:
```
# evaluate classifier on different datasets
clf_accuracy['h0'] = clf.evaluate(X_h0, y_h0, batch_size=128, verbose=0)[1]
preds_h0 = cd.predict(X_h0)
print('H0: Accuracy {:.4f} -- Drift? {}'.format(
    clf_accuracy['h0'], labels[preds_h0['data']['is_drift']]))
for x, c in zip(X_c, corruption):
    preds = cd.predict(x)
    print('{}: Accuracy {:.4f} -- Drift? {}'.format(
        c, clf_accuracy[c],labels[preds['data']['is_drift']]))
```

```
H0: Accuracy 0.9286 -- Drift? No!
gaussian_noise: Accuracy 0.2208 -- Drift? Yes!
motion_blur: Accuracy 0.6339 -- Drift? Yes!
brightness: Accuracy 0.8913 -- Drift? Yes!
pixelate: Accuracy 0.3666 -- Drift? Yes!
```

We can therefore **use the scores of the detector itself to quantify the harmfulness of the drift**! We can generalise this to all the corruptions at each severity level in CIFAR-10-C.

On the plot below we show the mean values and standard deviations of the adversarial scores per severity level. The plot shows the mean adversarial scores (lhs) and ResNet-32 accuracies (rhs) for increasing data corruption severity levels. Level 0 corresponds to the original test set. Harmful scores are scores from instances which have been flipped from the correct to an incorrect prediction because of the corruption. Not harmful means that the prediction was unchanged after the corruption. The chart can be reproduced in this notebook.



examples/image/adversarialscores.png

24.4.5 Deploy

We can deploy the drift detector in a similar fashion as the *outlier detector*. For a more detailed step-by-step overview of the deployment process, check this notebook.

[52]:
```
SERVICE_HOSTNAMES=!(kubectl get ksvc drift-detector -o jsonpath='{.status.url}' | cut␣
↪-d "/" -f 3)
SERVICE_HOSTNAME_CD=SERVICE_HOSTNAMES[0]
print(SERVICE_HOSTNAME_CD)
```

```
drift-detector.default.example.com
```

The deployed drift detector accumulates requests until a predefined `drift_batch_size` is reached, in our case 5000 which is defined in the *yaml* for the deployment and set in the drift detector wrapper. After 5000 instances, the batch is cleared and fills up again.

[54]:
```
from tqdm.notebook import tqdm

drift_batch_size = 5000

# accumulate batches
for i in tqdm(range(0, drift_batch_size, 100)):
    x = X_h0[i:i+100]
    predict(x)

# check message dumper
res=!kubectl logs $(kubectl get pod -l serving.knative.dev/configuration=message-
↪dumper-drift -o jsonpath='{.items[0].metadata.name}') user-container
data= []
for i in range(0,len(res)):
    if res[i] == 'Data,':
        data.append(res[i+1])
j = json.loads(json.loads(data[0]))
print("Drift?", labels[j["data"]["is_drift"]==1])
```

```
HBox(children=(FloatProgress(value=0.0, max=50.0), HTML(value='')))


Drift? No!
```

We now run the same test on some corrupted data:

```
[55]: c = 0

print(f'Corruption: {corruption[c]}')

# accumulate batches
for i in tqdm(range(0, drift_batch_size, 100)):
    x = X_c[c][i:i+100]
    predict(x)

# check message dumper
res=!kubectl logs $(kubectl get pod -l serving.knative.dev/configuration=message-
↪dumper-drift -o jsonpath='{.items[0].metadata.name}') user-container
data= []
for i in range(0,len(res)):
    if res[i] == 'Data,':
        data.append(res[i+1])
j = json.loads(json.loads(data[1]))
print("Drift?", labels[j["data"]["is_drift"]==1])
```

```
Corruption: gaussian_noise
```

```
HBox(children=(FloatProgress(value=0.0, max=50.0), HTML(value='')))
```

```

Drift? Yes!
```

CHAPTER

TWENTYFIVE

MAHALANOBIS OUTLIER DETECTION ON KDD CUP '99 DATASET

25.1 Method

The Mahalanobis online outlier detector aims to predict anomalies in tabular data. The algorithm calculates an outlier score, which is a measure of distance from the center of the features distribution (Mahalanobis distance). If this outlier score is higher than a user-defined threshold, the observation is flagged as an outlier. The algorithm is online, which means that it starts without knowledge about the distribution of the features and learns as requests arrive. Consequently you should expect the output to be bad at the start and to improve over time.

25.2 Dataset

The outlier detector needs to detect computer network intrusions using TCP dump data for a local-area network (LAN) simulating a typical U.S. Air Force LAN. A connection is a sequence of TCP packets starting and ending at some well defined times, between which data flows to and from a source IP address to a target IP address under some well defined protocol. Each connection is labeled as either normal, or as an attack.

There are 4 types of attacks in the dataset:

- DOS: denial-of-service, e.g. syn flood;
- R2L: unauthorized access from a remote machine, e.g. guessing password;
- U2R: unauthorized access to local superuser (root) privileges;
- probing: surveillance and other probing, e.g., port scanning.

The dataset contains about 5 million connection records.

There are 3 types of features:

- basic features of individual connections, e.g. duration of connection
- content features within a connection, e.g. number of failed log in attempts
- traffic features within a 2 second window, e.g. number of connections to the same host as the current connection

[1]:
```
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import seaborn as sns
from sklearn.metrics import confusion_matrix, f1_score
```

(continues on next page)

(continued from previous page)

```
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder

from alibi_detect.od import Mahalanobis
from alibi_detect.datasets import fetch_kdd
from alibi_detect.utils.data import create_outlier_batch
from alibi_detect.utils.fetching import fetch_detector
from alibi_detect.utils.mapping import ord2ohe
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.utils.visualize import plot_instance_score, plot_roc
```

25.3 Load dataset

We only keep a number of continuous (18 out of 41) features.

```
[2]: kddcup = fetch_kdd(percent10=True)  # only load 10% of the dataset
print(kddcup.data.shape, kddcup.target.shape)
```

```
(494021, 18) (494021,)
```

Assume that a machine learning model is trained on *normal* instances of the dataset (not outliers) and standardization is applied:

```
[3]: np.random.seed(0)
normal_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=100000,␣
↪perc_outlier=0)
X_train, y_train = normal_batch.data.astype('float'), normal_batch.target
print(X_train.shape, y_train.shape)
print('{}% outliers'.format(100 * y_train.mean()))
```

```
(100000, 18) (100000,)
0.0% outliers
```

```
[4]: mean, stdev = X_train.mean(axis=0), X_train.std(axis=0)
```

25.4 Load or define outlier detector

The pretrained outlier and adversarial detectors used in the example notebooks can be found here. You can use the built-in `fetch_detector` function which saves the pre-trained models in a local directory `filepath` and loads the detector. Alternatively, you can initialise a detector from scratch.

Be aware that `Mahalanobis` is an online, stateful outlier detector. Saving or loading a Mahalanobis detector therefore also saves and loads the state of the detector. This allows the user to *warm up* the detector before deploying it into production.

```
[5]: load_outlier_detector = False
```

```
[6]: filepath = 'my_path'  # change to directory where model is downloaded
if load_outlier_detector:  # load initialized outlier detector
    detector_type = 'outlier'
    dataset = 'kddcup'
    detector_name = 'Mahalanobis'
```

(continues on next page)

(continued from previous page)

```
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
else:  # initialize and save outlier detector
    threshold = None  # scores above threshold are classified as outliers
    n_components = 2  # nb of components used in PCA
    std_clip = 3  # clip values used to compute mean and cov above "std_clip"␣
↪standard deviations
    start_clip = 20  # start clipping values after "start_clip" instances

    od = Mahalanobis(threshold,
                     n_components=n_components,
                     std_clip=std_clip,
                     start_clip=start_clip)

    save_detector(od, filepath)  # save outlier detector
```

```
WARNING:alibi_detect.od.mahalanobis:No threshold level set. Need to infer threshold␣
↪using `infer_threshold`.
```

If `load_outlier_detector` equals False, the warning tells us we still need to set the outlier threshold. This can be done with the `infer_threshold` method. We need to pass a batch of instances and specify what percentage of those we consider to be normal via `threshold_perc`. Let's assume we have some data which we know contains around 5% outliers. The percentage of outliers can be set with `perc_outlier` in the `create_outlier_batch` function.

```
[7]: np.random.seed(0)
     perc_outlier = 5
     threshold_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=1000,␣
     ↪perc_outlier=perc_outlier)
     X_threshold, y_threshold = threshold_batch.data.astype('float'), threshold_batch.
     ↪target
     X_threshold = (X_threshold - mean) / stdev
     print('{}% outliers'.format(100 * y_threshold.mean()))
```

```
5.0% outliers
```

```
[8]: od.infer_threshold(X_threshold, threshold_perc=100-perc_outlier)
     print('New threshold: {}'.format(od.threshold))
     threshold = od.threshold
```

```
New threshold: 10.670934046719518
```

25.5 Detect outliers

We now generate a batch of data with 10% outliers, standardize those with the `mean` and `stdev` values obtained from the normal data (*inliers*) and detect the outliers in the batch.

```
[9]: np.random.seed(1)
     outlier_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=1000, perc_
     ↪outlier=10)
     X_outlier, y_outlier = outlier_batch.data.astype('float'), outlier_batch.target
     X_outlier = (X_outlier - mean) / stdev
     print(X_outlier.shape, y_outlier.shape)
     print('{}% outliers'.format(100 * y_outlier.mean()))
```

```
(1000, 18) (1000,)
10.0% outliers
```

Predict outliers:

```
[10]: od_preds = od.predict(X_outlier, return_instance_score=True)
```

We can now save the *warmed up* outlier detector:

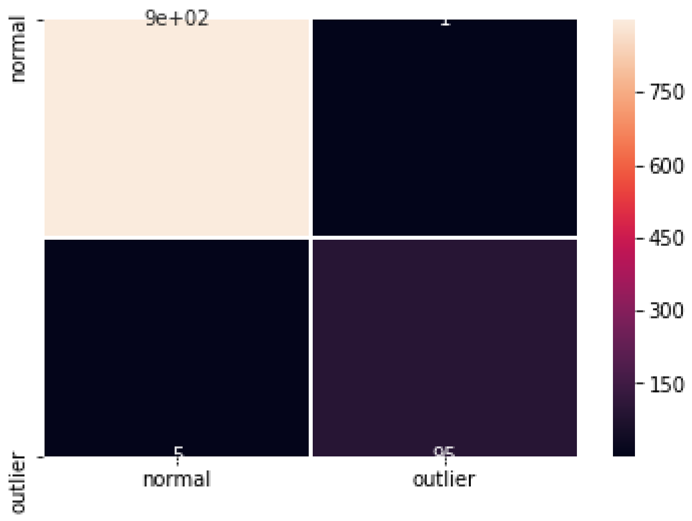
```
[11]: save_detector(od, filepath)
```

25.6 Display results

F1 score and confusion matrix:

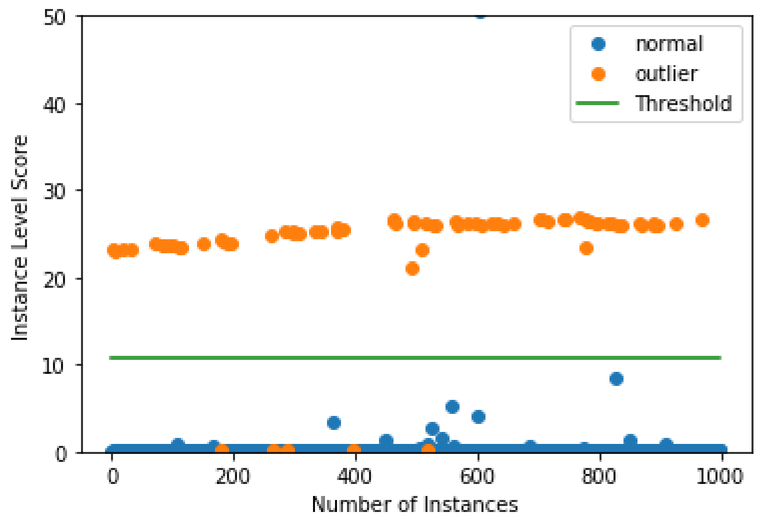
```
[12]: labels = outlier_batch.target_names
y_pred = od_preds['data']['is_outlier']
f1 = f1_score(y_outlier, y_pred)
print('F1 score: {}'.format(f1))
cm = confusion_matrix(y_outlier, y_pred)
df_cm = pd.DataFrame(cm, index=labels, columns=labels)
sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
plt.show()
```

```
F1 score: 0.9693877551020408
```



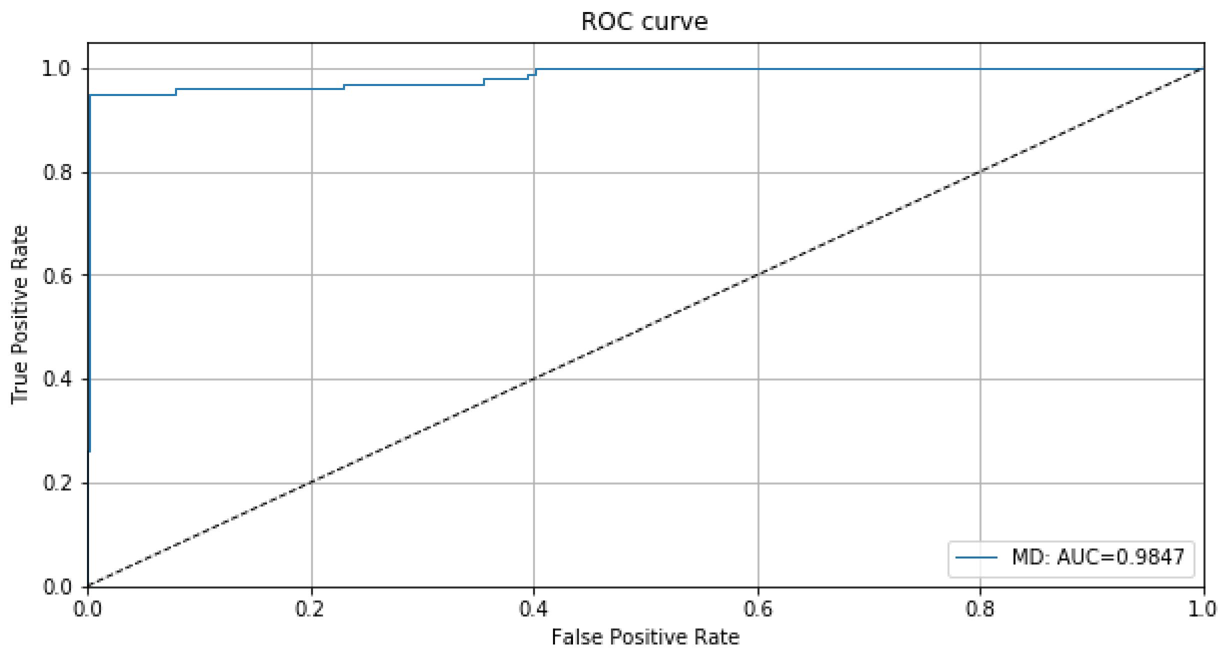
Plot instance level outlier scores vs. the outlier threshold:

```
[13]: plot_instance_score(od_preds, y_outlier, labels, od.threshold, ylim=(0,50))
```



We can also plot the ROC curve for the outlier scores of the detector:

```
[14]: roc_data = {'MD': {'scores': od_preds['data']['instance_score'], 'labels': y_outlier}}
plot_roc(roc_data)
```



25.7 Include categorical variables

So far we only tracked continuous variables. We can however also include categorical variables. The `fit` step first computes pairwise distances between the categories of each categorical variable. The pairwise distances are based on either the model predictions (*MVDM method*) or the context provided by the other variables in the dataset (*ABDM method*). 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.

25.7.1 Load and transform data

```
[15]: cat_cols = ['protocol_type', 'service', 'flag']
      num_cols = ['srv_count', 'serror_rate', 'srv_serror_rate',
                  'rerror_rate', 'srv_rerror_rate', 'same_srv_rate',
                  'diff_srv_rate', 'srv_diff_host_rate', 'dst_host_count',
                  'dst_host_srv_count', 'dst_host_same_srv_rate',
                  'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate',
                  'dst_host_srv_diff_host_rate', 'dst_host_serror_rate',
                  'dst_host_srv_serror_rate', 'dst_host_rerror_rate',
                  'dst_host_srv_rerror_rate']
      cols = cat_cols + num_cols
```

```
[16]: np.random.seed(0)
      kddcup = fetch_kdd(keep_cols=cols, percent10=True)
      print(kddcup.data.shape, kddcup.target.shape)

      (494021, 21) (494021,)
```

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 `fit` step of the outlier detector.

```
[17]: cat_vars_ord = {}
      n_categories = len(cat_cols)
      for i in range(n_categories):
          cat_vars_ord[i] = len(np.unique(kddcup.data[:, i]))
      print(cat_vars_ord)

      {0: 3, 1: 66, 2: 11}
```

Fit an ordinal encoder on the categorical data:

```
[18]: enc = OrdinalEncoder()
      enc.fit(kddcup.data[:, :n_categories])

[18]: OrdinalEncoder(categories='auto', dtype=<class 'numpy.float64'>)
```

Combine scaled numerical and ordinal features. `X_fit` will be used to infer distances between categorical features later. To make it easy, we will already transform the whole dataset, including the outliers that need to be detected later. This is for illustrative purposes:

```
[19]: X_num = (kddcup.data[:, n_categories:] - mean) / stdev  # standardize numerical␣
      ↪features
      X_ord = enc.transform(kddcup.data[:, :n_categories])  # apply ordinal encoding to␣
      ↪categorical features
```

(continues on next page)

(continued from previous page)

```
X_fit = np.c_[X_ord, X_num].astype(np.float32, copy=False)  # combine numerical and␣
↪categorical features
print(X_fit.shape)
```

```
(494021, 21)
```

25.7.2 Initialize and fit outlier detector

We use the same threshold as for the continuous data. This will likely not result in optimal performance. Alternatively, you can infer the threshold again.

```
[20]: n_components = 2
std_clip = 3
start_clip = 20

od = Mahalanobis(threshold,
                 n_components=n_components,
                 std_clip=std_clip,
                 start_clip=start_clip,
                 cat_vars=cat_vars_ord,
                 ohe=False)  # True if one-hot encoding (OHE) is used
```

Set `fit` parameters:

```
[21]: d_type = 'abdm'  # pairwise distance type, 'abdm' infers context from other variables
disc_perc = [25, 50, 75]  # percentiles used to bin numerical values; used in 'abdm'␣
↪calculations
standardize_cat_vars = True  # standardize numerical values of categorical variables
```

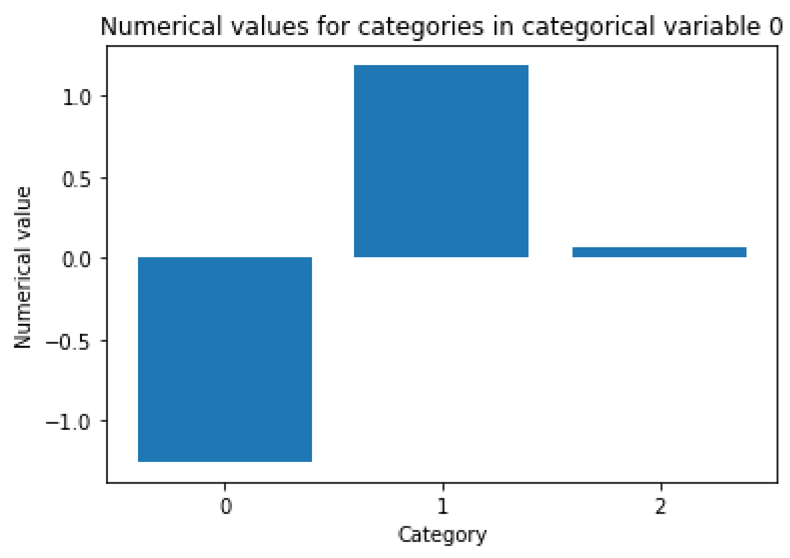
Apply `fit` method to find numerical values for categorical variables:

```
[22]: od.fit(X_fit,
       d_type=d_type,
       disc_perc=disc_perc,
       standardize_cat_vars=standardize_cat_vars)
```

The numerical values for the categorical features are stored in the attribute `od.d_abs`. This is a dictionary with as keys the columns for the categorical features and as values the numerical equivalent of the category:

```
[23]: cat = 0  # categorical variable to plot numerical values for
```

```
[24]: plt.bar(np.arange(len(od.d_abs[cat])), od.d_abs[cat])
plt.xticks(np.arange(len(od.d_abs[cat])))
plt.title('Numerical values for categories in categorical variable {}'.format(cat))
plt.xlabel('Category')
plt.ylabel('Numerical value')
plt.show()
```



Another option would be to set `d_type` to `'mvdm'` and `y` to `kddcup.target` to infer the numerical values for categorical variables from the model labels (or alternatively the predictions).

25.7.3 Run outlier detector and display results

Generate batch of data with 10% outliers:

```
[25]: np.random.seed(1)
outlier_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=1000, perc_
↪outlier=10)
data, y_outlier = outlier_batch.data, outlier_batch.target
print(data.shape, y_outlier.shape)
print('{}% outliers'.format(100 * y_outlier.mean()))
```

```
(1000, 21) (1000,)
10.0% outliers
```

Preprocess the outlier batch:

```
[26]: X_num = (data[:, n_categories:] - mean) / stdev
X_ord = enc.transform(data[:, :n_categories])
X_outlier = np.c_[X_ord, X_num].astype(np.float32, copy=False)
print(X_outlier.shape)
```

```
(1000, 21)
```

Predict outliers:

```
[27]: od_preds = od.predict(X_outlier, return_instance_score=True)
```

F1 score and confusion matrix:

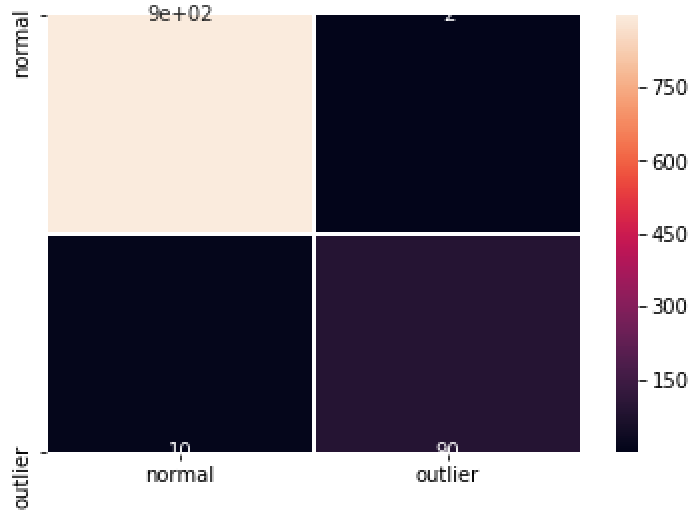
```
[28]: y_pred = od_preds['data']['is_outlier']
f1 = f1_score(y_outlier, y_pred)
print('F1 score: {}'.format(f1))
cm = confusion_matrix(y_outlier, y_pred)
df_cm = pd.DataFrame(cm, index=labels, columns=labels)
```

(continues on next page)

(continued from previous page)

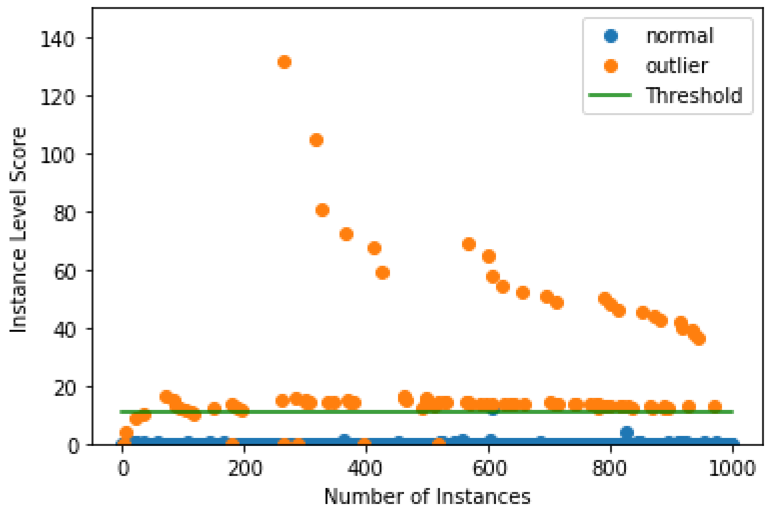
```
sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
plt.show()
```

```
F1 score: 0.9375
```



Plot instance level outlier scores vs. the outlier threshold:

```
[29]: plot_instance_score(od_preds, y_outlier, labels, od.threshold, ylim=(0, 150))
```



25.8 Use OHE instead of ordinal encoding for the categorical variables

Since we will apply one-hot encoding (OHE) on the categorical variables, we convert `cat_vars_ord` from the ordinal to OHE format. `alibi_detect.utils.mapping` contains utility functions to do this. The keys in `cat_vars_ohe` now represent the first column index for each one-hot encoded categorical variable. This dictionary will later be used in the counterfactual explanation.

[30]:
```
cat_vars_ohe = ord2ohe(X_fit, cat_vars_ord)[1]
print(cat_vars_ohe)
```

```
{0: 3, 3: 66, 69: 11}
```

Fit a one-hot encoder on the categorical data:

[31]:
```
enc = OneHotEncoder(categories='auto')
enc.fit(X_fit[:, :n_categories])
```

[31]:
```
OneHotEncoder(categorical_features=None, categories='auto', drop=None,
              dtype=<class 'numpy.float64'>, handle_unknown='error',
              n_values=None, sparse=True)
```

Transform `X_fit` to OHE:

[32]:
```
X_ohe = enc.transform(X_fit[:, :n_categories])
X_fit = np.array(np.c_[X_ohe.todense(), X_fit[:, n_categories:]].astype(np.float32,␣
↪copy=False))
print(X_fit.shape)
```

```
(494021, 98)
```

25.8.1 Initialize and fit outlier detector

Initialize:

[33]:
```
od = Mahalanobis(threshold,
                 n_components=n_components,
                 std_clip=std_clip,
                 start_clip=start_clip,
                 cat_vars=cat_vars_ohe,
                 ohe=True)
```

Apply fit method:

[34]:
```
od.fit(X_fit,
       d_type=d_type,
       disc_perc=disc_perc,
       standardize_cat_vars=standardize_cat_vars)
```

25.8.2 Run outlier detector and display results

Transform outlier batch to OHE:

```
[35]: X_ohe = enc.transform(X_ord)
X_outlier = np.array(np.c_[X_ohe.todense(), X_num].astype(np.float32, copy=False))
print(X_outlier.shape)
```

```
(1000, 98)
```

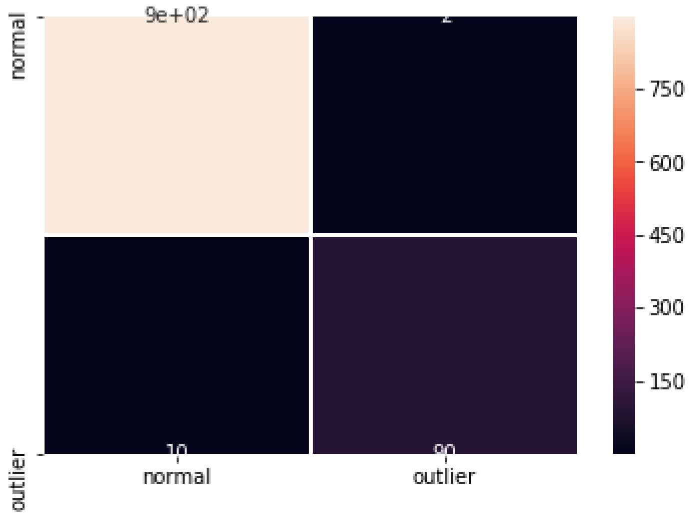
Predict outliers:

```
[36]: od_preds = od.predict(X_outlier, return_instance_score=True)
```

F1 score and confusion matrix:

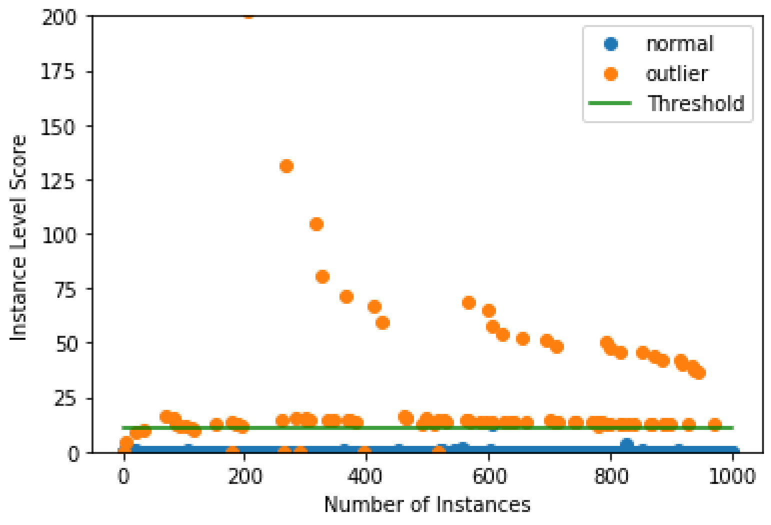
```
[37]: y_pred = od_preds['data']['is_outlier']
f1 = f1_score(y_outlier, y_pred)
print('F1 score: {}'.format(f1))
cm = confusion_matrix(y_outlier, y_pred)
df_cm = pd.DataFrame(cm, index=labels, columns=labels)
sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
plt.show()
```

```
F1 score: 0.9375
```



Plot instance level outlier scores vs. the outlier threshold:

```
[38]: plot_instance_score(od_preds, y_outlier, labels, od.threshold, ylim=(0,200))
```


normal
outlier
Threshold
Instance Level Score
Number of Instances
200
175
150
125
100
75
50
25
0
0
200
400
600
800
1000

CHAPTER

TWENTYSIX

ISOLATION FOREST OUTLIER DETECTION ON KDD CUP ‘99 DATASET

26.1 Method

Isolation forests (IF) are tree based models specifically used for outlier detection. The IF isolates observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. The number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node. This path length, averaged over a forest of random trees, is a measure of normality and is used to define an anomaly score. Outliers can typically be isolated quicker, leading to shorter paths.

26.2 Dataset

The outlier detector needs to detect computer network intrusions using TCP dump data for a local-area network (LAN) simulating a typical U.S. Air Force LAN. A connection is a sequence of TCP packets starting and ending at some well defined times, between which data flows to and from a source IP address to a target IP address under some well defined protocol. Each connection is labeled as either normal, or as an attack.

There are 4 types of attacks in the dataset:

- DOS: denial-of-service, e.g. syn flood;
- R2L: unauthorized access from a remote machine, e.g. guessing password;
- U2R: unauthorized access to local superuser (root) privileges;
- probing: surveillance and other probing, e.g., port scanning.

The dataset contains about 5 million connection records.

There are 3 types of features:

- basic features of individual connections, e.g. duration of connection
- content features within a connection, e.g. number of failed log in attempts
- traffic features within a 2 second window, e.g. number of connections to the same host as the current connection

```
[1]: import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import seaborn as sns
```

(continues on next page)

(continued from previous page)

```
from sklearn.metrics import confusion_matrix, f1_score

from alibi_detect.od import IForest
from alibi_detect.datasets import fetch_kdd
from alibi_detect.utils.data import create_outlier_batch
from alibi_detect.utils.fetching import fetch_detector
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.utils.visualize import plot_instance_score, plot_roc
```

26.3 Load dataset

We only keep a number of continuous (18 out of 41) features.

[2]:
```
kddcup = fetch_kdd(percent10=True)  # only load 10% of the dataset
print(kddcup.data.shape, kddcup.target.shape)
```

```
(494021, 18) (494021,)
```

Assume that a model is trained on *normal* instances of the dataset (not outliers) and standardization is applied:

[3]:
```
np.random.seed(0)
normal_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=400000,␣
↪perc_outlier=0)
X_train, y_train = normal_batch.data.astype('float'), normal_batch.target
print(X_train.shape, y_train.shape)
print('{}% outliers'.format(100 * y_train.mean()))
```

```
(400000, 18) (400000,)
0.0% outliers
```

[4]:
```
mean, stdev = X_train.mean(axis=0), X_train.std(axis=0)
```

Apply standardization:

[5]:
```
X_train = (X_train - mean) / stdev
```

26.4 Load or define outlier detector

The pretrained outlier and adversarial detectors used in the example notebooks can be found here. You can use the built-in `fetch_detector` function which saves the pre-trained models in a local directory `filepath` and loads the detector. Alternatively, you can train a detector from scratch:

[6]:
```
load_outlier_detector = True
```

[7]:
```
filepath = 'my_path'  # change to directory where model is downloaded
if load_outlier_detector:  # load pretrained outlier detector
    detector_type = 'outlier'
    dataset = 'kddcup'
    detector_name = 'IForest'
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
```

(continues on next page)

(continued from previous page)

```
else:  # define model, initialize, train and save outlier detector

    # initialize outlier detector
    od = IForest(threshold=None,  # threshold for outlier score
                 n_estimators=100)

    # train
    od.fit(X_train)

    # save the trained outlier detector
    save_detector(od, filepath)
```

If `load_outlier_detector` equals False, the warning tells us we still need to set the outlier threshold. This can be done with the `infer_threshold` method. We need to pass a batch of instances and specify what percentage of those we consider to be normal via `threshold_perc`. Let's assume we have some data which we know contains around 5% outliers. The percentage of outliers can be set with `perc_outlier` in the `create_outlier_batch` function.

```
[8]: np.random.seed(0)
perc_outlier = 5
threshold_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=1000,␣
↪perc_outlier=perc_outlier)
X_threshold, y_threshold = threshold_batch.data.astype('float'), threshold_batch.
↪target
X_threshold = (X_threshold - mean) / stdev
print('{}% outliers'.format(100 * y_threshold.mean()))
```

```
5.0% outliers
```

```
[9]: od.infer_threshold(X_threshold, threshold_perc=100-perc_outlier)
print('New threshold: {}'.format(od.threshold))
```

```
New threshold: 0.08008174509752322
```

Let's save the outlier detector with updated threshold:

```
[10]: save_detector(od, filepath)
```

26.5 Detect outliers

We now generate a batch of data with 10% outliers and detect the outliers in the batch.

```
[11]: np.random.seed(1)
outlier_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=1000, perc_
↪outlier=10)
X_outlier, y_outlier = outlier_batch.data.astype('float'), outlier_batch.target
X_outlier = (X_outlier - mean) / stdev
print(X_outlier.shape, y_outlier.shape)
print('{}% outliers'.format(100 * y_outlier.mean()))
```

```
(1000, 18) (1000,)
10.0% outliers
```

Predict outliers:

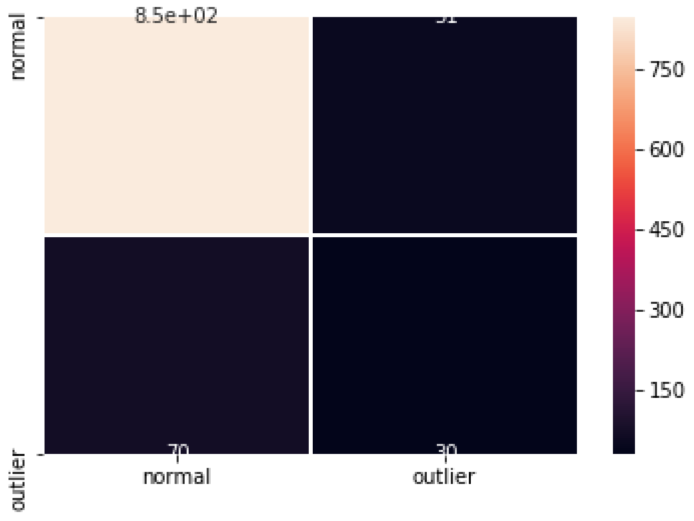
[12]:
```
od_preds = od.predict(X_outlier, return_instance_score=True)
```

26.6 Display results

F1 score and confusion matrix:

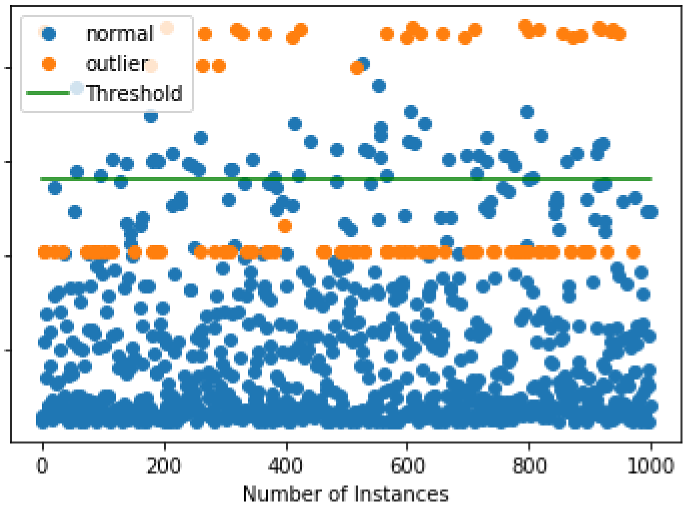
[13]:
```
labels = outlier_batch.target_names
y_pred = od_preds['data']['is_outlier']
f1 = f1_score(y_outlier, y_pred)
print('F1 score: {:.4f}'.format(f1))
cm = confusion_matrix(y_outlier, y_pred)
df_cm = pd.DataFrame(cm, index=labels, columns=labels)
sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
plt.show()
```

```
F1 score: 0.3315
```



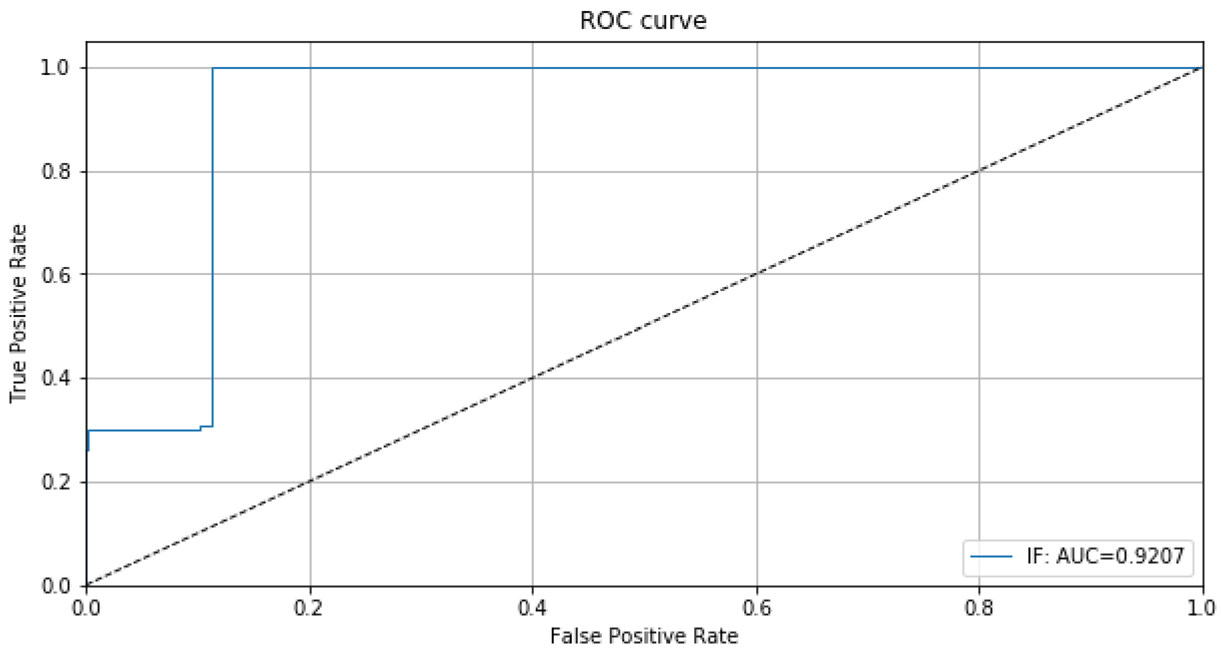
Plot instance level outlier scores vs. the outlier threshold:

[14]:
```
plot_instance_score(od_preds, y_outlier, labels, od.threshold)
```



We can see that the isolation forest does not do a good job at detecting 1 type of outliers with an outlier score around 0. This makes inferring a good threshold without explicit knowledge about the outliers hard. Setting the threshold just below 0 would lead to significantly better detector performance for the outliers in the dataset. This is also reflected by the ROC curve:

```
[15]: roc_data = {'IF': {'scores': od_preds['data']['instance_score'], 'labels': y_outlier}}
plot_roc(roc_data)
```



CHAPTER

TWENTYSEVEN

VAE OUTLIER DETECTION ON KDD CUP '99 DATASET

27.1 Method

The Variational Auto-Encoder (VAE) outlier detector is first trained on a batch of unlabeled, but normal (*inlier*) data. Unsupervised training is desireable since labeled data is often scarce. The VAE detector tries to reconstruct the input it receives. If the input data cannot be reconstructed well, the reconstruction error is high and the data can be flagged as an outlier. The reconstruction error is either measured as the mean squared error (MSE) between the input and the reconstructed instance or as the probability that both the input and the reconstructed instance are generated by the same process.

27.2 Dataset

The outlier detector needs to detect computer network intrusions using TCP dump data for a local-area network (LAN) simulating a typical U.S. Air Force LAN. A connection is a sequence of TCP packets starting and ending at some well defined times, between which data flows to and from a source IP address to a target IP address under some well defined protocol. Each connection is labeled as either normal, or as an attack.

There are 4 types of attacks in the dataset:

- DOS: denial-of-service, e.g. syn flood;
- R2L: unauthorized access from a remote machine, e.g. guessing password;
- U2R: unauthorized access to local superuser (root) privileges;
- probing: surveillance and other probing, e.g., port scanning.

The dataset contains about 5 million connection records.

There are 3 types of features:

- basic features of individual connections, e.g. duration of connection
- content features within a connection, e.g. number of failed log in attempts
- traffic features within a 2 second window, e.g. number of connections to the same host as the current connection

[1]:
```
import logging
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.metrics import confusion_matrix, f1_score
```

(continues on next page)

(continued from previous page)

```
import tensorflow as tf
tf.keras.backend.clear_session()
from tensorflow.keras.layers import Dense, InputLayer

from alibi_detect.datasets import fetch_kdd
from alibi_detect.models.tensorflow.losses import elbo
from alibi_detect.od import OutlierVAE
from alibi_detect.utils.data import create_outlier_batch
from alibi_detect.utils.fetching import fetch_detector
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.utils.visualize import plot_instance_score, plot_feature_outlier_
↪tabular, plot_roc

logger = tf.get_logger()
logger.setLevel(logging.ERROR)
```

```
ERROR:fbprophet:Importing plotly failed. Interactive plots will not work.
```

27.3 Load dataset

We only keep a number of continuous (18 out of 41) features.

```
[2]: kddcup = fetch_kdd(percent10=True)  # only load 10% of the dataset
print(kddcup.data.shape, kddcup.target.shape)
```

```
(494021, 18) (494021,)
```

Assume that a model is trained on *normal* instances of the dataset (not outliers) and standardization is applied:

```
[3]: np.random.seed(0)
normal_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=400000,␣
↪perc_outlier=0)
X_train, y_train = normal_batch.data.astype('float'), normal_batch.target
print(X_train.shape, y_train.shape)
print('{}% outliers'.format(100 * y_train.mean()))
```

```
(400000, 18) (400000,)
0.0% outliers
```

```
[4]: mean, stdev = X_train.mean(axis=0), X_train.std(axis=0)
```

Apply standardization:

```
[5]: X_train = (X_train - mean) / stdev
```

27.4 Load or define outlier detector

The pretrained outlier and adversarial detectors used in the example notebooks can be found here. You can use the built-in `fetch_detector` function which saves the pre-trained models in a local directory `filepath` and loads the detector. Alternatively, you can train a detector from scratch:

[6]:
```
load_outlier_detector = True
```

[7]:
```
filepath = 'my_dir'  # change to directory (absolute path) where model is downloaded
if load_outlier_detector:  # load pretrained outlier detector
    detector_type = 'outlier'
    dataset = 'kddcup'
    detector_name = 'OutlierVAE'
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
else:  # define model, initialize, train and save outlier detector
    n_features = X_train.shape[1]
    latent_dim = 2

    encoder_net = tf.keras.Sequential(
      [
          InputLayer(input_shape=(n_features,)),
          Dense(20, activation=tf.nn.relu),
          Dense(15, activation=tf.nn.relu),
          Dense(7, activation=tf.nn.relu)
      ])

    decoder_net = tf.keras.Sequential(
      [
          InputLayer(input_shape=(latent_dim,)),
          Dense(7, activation=tf.nn.relu),
          Dense(15, activation=tf.nn.relu),
          Dense(20, activation=tf.nn.relu),
          Dense(n_features, activation=None)
      ])

    # initialize outlier detector
    od = OutlierVAE(threshold=None,  # threshold for outlier score
                    score_type='mse',  # use MSE of reconstruction error for outlier␣
↪detection
                    encoder_net=encoder_net,  # can also pass VAE model instead
                    decoder_net=decoder_net,  # of separate encoder and decoder
                    latent_dim=latent_dim,
                    samples=5)
    # train
    od.fit(X_train,
           loss_fn=elbo,
           cov_elbo=dict(sim=.01),
           epochs=30,
           verbose=True)

    # save the trained outlier detector
    save_detector(od, filepath)
```

```
WARNING:alibi_detect.od.vae:No threshold level set. Need to infer threshold using␣
↪`infer_threshold`.
```

The warning tells us we still need to set the outlier threshold. This can be done with the `infer_threshold` method. We need to pass a batch of instances and specify what percentage of those we consider to be normal via `threshold_perc`. Let's assume we have some data which we know contains around 5% outliers. The percentage of outliers can be set with `perc_outlier` in the `create_outlier_batch` function.

[8]:
```
np.random.seed(0)
perc_outlier = 5
threshold_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=1000,␣
↪perc_outlier=perc_outlier)
X_threshold, y_threshold = threshold_batch.data.astype('float'), threshold_batch.
↪target
X_threshold = (X_threshold - mean) / stdev
print('{}% outliers'.format(100 * y_threshold.mean()))
```

```
5.0% outliers
```

[9]:
```
od.infer_threshold(X_threshold, threshold_perc=100-perc_outlier)
print('New threshold: {}'.format(od.threshold))
```

```
New threshold: 1.7367815971374498
```

We could have also inferred the threshold from the normal training data by setting `threshold_perc` e.g. at 99 and adding a bit of margin on top of the inferred threshold. Let's save the outlier detector with updated threshold:

[10]:
```
save_detector(od, filepath)
```

27.5 Detect outliers

We now generate a batch of data with 10% outliers and detect the outliers in the batch.

[11]:
```
np.random.seed(1)
outlier_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=1000, perc_
↪outlier=10)
X_outlier, y_outlier = outlier_batch.data.astype('float'), outlier_batch.target
X_outlier = (X_outlier - mean) / stdev
print(X_outlier.shape, y_outlier.shape)
print('{}% outliers'.format(100 * y_outlier.mean()))
```

```
(1000, 18) (1000,)
10.0% outliers
```

Predict outliers:

[12]:
```
od_preds = od.predict(X_outlier,
                      outlier_type='instance',    # use 'feature' or 'instance' level
                      return_feature_score=True,  # scores used to determine outliers
                      return_instance_score=True)
print(list(od_preds['data'].keys()))
```

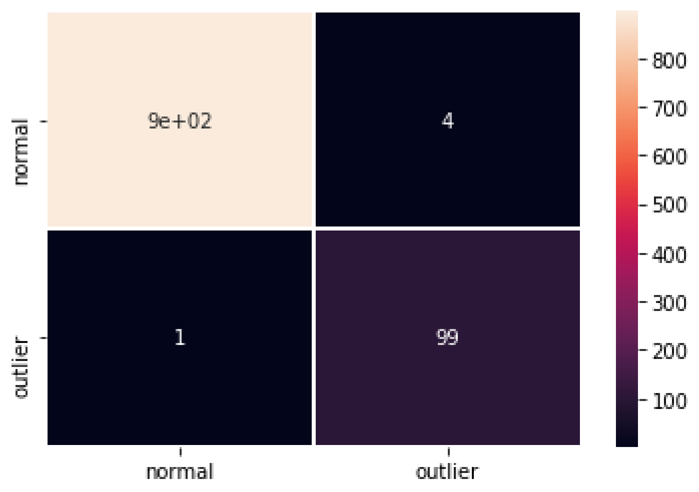
```
['instance_score', 'feature_score', 'is_outlier']
```

27.6 Display results

F1 score and confusion matrix:

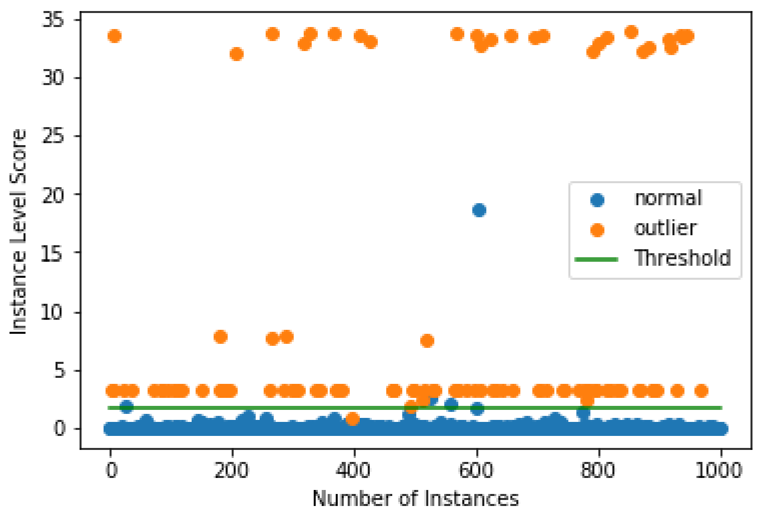
```
[13]: labels = outlier_batch.target_names
y_pred = od_preds['data']['is_outlier']
f1 = f1_score(y_outlier, y_pred)
print('F1 score: {:.4f}'.format(f1))
cm = confusion_matrix(y_outlier, y_pred)
df_cm = pd.DataFrame(cm, index=labels, columns=labels)
sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
plt.show()
```

```
F1 score: 0.9754
```



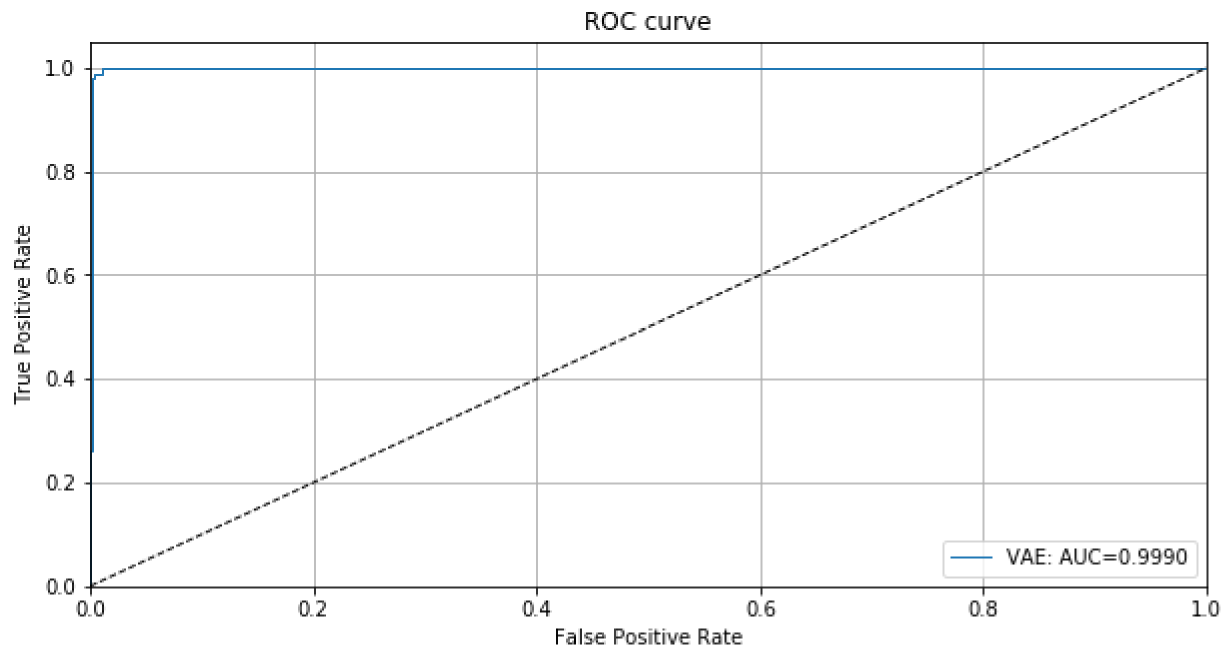
Plot instance level outlier scores vs. the outlier threshold:

```
[14]: plot_instance_score(od_preds, y_outlier, labels, od.threshold)
```



We can clearly see that some outliers are very easy to detect while others have outlier scores closer to the normal data. We can also plot the ROC curve for the outlier scores of the detector:

```
[15]: roc_data = {'VAE': {'scores': od_preds['data']['instance_score'], 'labels': y_outlier}
      ↪}
      plot_roc(roc_data)
```

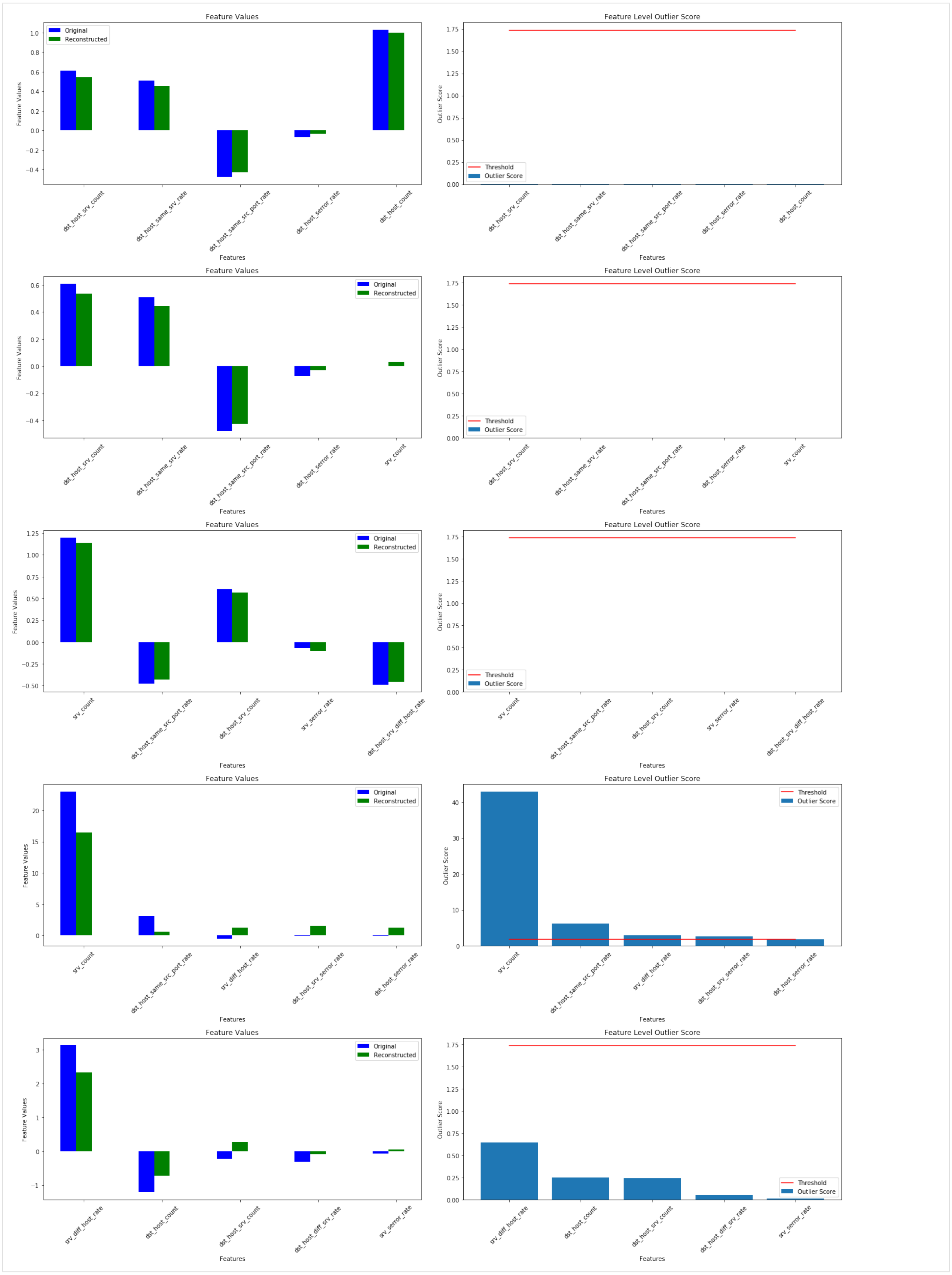


27.7 Investigate instance level outlier

We can now take a closer look at some of the individual predictions on `X_outlier`.

```
[16]: X_recon = od.vae(X_outlier).numpy()  # reconstructed instances by the VAE
```

```
[17]: plot_feature_outlier_tabular(od_preds,
                                   X_outlier,
                                   X_recon=X_recon,
                                   threshold=od.threshold,
                                   instance_ids=None,  # pass a list with indices of␣
      ↪instances to display
                                   max_instances=5,  # max nb of instances to display
                                   top_n=5,  # only show top_n features ordered by outlier␣
      ↪score
                                   outliers_only=False,  # only show outlier predictions
                                   feature_names=kddcup.feature_names,  # add feature names
                                   figsize=(20, 30))
```


Feature Values
Original
Reconstructed
Feature Values
Features
dst_host_srv_count
dst_host_same_srv_rate
dst_host_same_src_port_rate
dst_host_serror_rate
dst_host_count
Feature Level Outlier Score
Threshold
Outlier Score
Outlier Score
Features
Feature Values
dst_host_srv_count
dst_host_same_srv_rate
dst_host_same_src_port_rate
dst_host_serror_rate
srv_count
Feature Level Outlier Score
Feature Values
srv_count
dst_host_same_src_port_rate
dst_host_srv_count
srv_serror_rate
dst_host_srv_diff_host_rate
Feature Level Outlier Score
Feature Values
srv_count
dst_host_same_src_port_rate
srv_diff_host_rate
dst_host_srv_serror_rate
dst_host_serror_rate
Feature Level Outlier Score
Feature Values
srv_diff_host_rate
dst_host_count
dst_host_srv_count
dst_host_diff_srv_rate
srv_serror_rate
Feature Level Outlier Score

The `srv_count` feature is responsible for a lot of the displayed outliers.

CHAPTER

TWENTYEIGHT

VAE OUTLIER DETECTION ON CIFAR10

28.1 Method

The Variational Auto-Encoder (VAE) outlier detector is first trained on a batch of unlabeled, but normal (*inlier*) data. Unsupervised training is desireable since labeled data is often scarce. The VAE detector tries to reconstruct the input it receives. If the input data cannot be reconstructed well, the reconstruction error is high and the data can be flagged as an outlier. The reconstruction error is either measured as the mean squared error (MSE) between the input and the reconstructed instance or as the probability that both the input and the reconstructed instance are generated by the same process.

28.2 Dataset

CIFAR10 consists of 60,000 32 by 32 RGB images equally distributed over 10 classes.

```
[1]: import logging
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
tf.keras.backend.clear_session()
from tensorflow.keras.layers import Conv2D, Conv2DTranspose, Dense, Layer, Reshape,␣
↪InputLayer
from tqdm import tqdm

from alibi_detect.models.tensorflow.losses import elbo
from alibi_detect.od import OutlierVAE
from alibi_detect.utils.fetching import fetch_detector
from alibi_detect.utils.perturbation import apply_mask
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.utils.visualize import plot_instance_score, plot_feature_outlier_
↪image

logger = tf.get_logger()
logger.setLevel(logging.ERROR)
```

28.3 Load CIFAR10 data

```
[2]: train, test = tf.keras.datasets.cifar10.load_data()
X_train, y_train = train
X_test, y_test = test

X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

```
(50000, 32, 32, 3) (50000, 1) (10000, 32, 32, 3) (10000, 1)
```

28.4 Load or define outlier detector

The pretrained outlier and adversarial detectors used in the example notebooks can be found here. You can use the built-in `fetch_detector` function which saves the pre-trained models in a local directory `filepath` and loads the detector. Alternatively, you can train a detector from scratch:

```
[3]: load_outlier_detector = True
```

```
[4]: filepath = 'my_path'  # change to directory where model is downloaded
if load_outlier_detector:  # load pretrained outlier detector
    detector_type = 'outlier'
    dataset = 'cifar10'
    detector_name = 'OutlierVAE'
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
else:  # define model, initialize, train and save outlier detector
    latent_dim = 1024

    encoder_net = tf.keras.Sequential(
      [
          InputLayer(input_shape=(32, 32, 3)),
          Conv2D(64, 4, strides=2, padding='same', activation=tf.nn.relu),
          Conv2D(128, 4, strides=2, padding='same', activation=tf.nn.relu),
          Conv2D(512, 4, strides=2, padding='same', activation=tf.nn.relu)
      ])

    decoder_net = tf.keras.Sequential(
      [
          InputLayer(input_shape=(latent_dim,)),
          Dense(4*4*128),
          Reshape(target_shape=(4, 4, 128)),
          Conv2DTranspose(256, 4, strides=2, padding='same', activation=tf.nn.relu),
          Conv2DTranspose(64, 4, strides=2, padding='same', activation=tf.nn.relu),
          Conv2DTranspose(3, 4, strides=2, padding='same', activation='sigmoid')
      ])

    # initialize outlier detector
    od = OutlierVAE(threshold=.015,  # threshold for outlier score
                    score_type='mse',  # use MSE of reconstruction error for outlier␣
↪detection
                    encoder_net=encoder_net,  # can also pass VAE model instead
                    decoder_net=decoder_net,  # of separate encoder and decoder
```

(continues on next page)

(continued from previous page)

```
                        latent_dim=latent_dim,
                        samples=2)
    # train
    od.fit(X_train,
           loss_fn=elbo,
           cov_elbo=dict(sim=.05),
           epochs=50,
           verbose=False)

    # save the trained outlier detector
    save_detector(od, filepath)
```

28.5 Check quality VAE model

[5]:
```
idx = 8
X = X_train[idx].reshape(1, 32, 32, 3)
X_recon = od.vae(X)
```

[6]:
```
plt.imshow(X.reshape(32, 32, 3))
plt.axis('off')
plt.show()
```

[7]:
```
plt.imshow(X_recon.numpy().reshape(32, 32, 3))
plt.axis('off')
plt.show()
```

28.6 Check outliers on original CIFAR images

```
[8]: X = X_train[:500]
     print(X.shape)
```

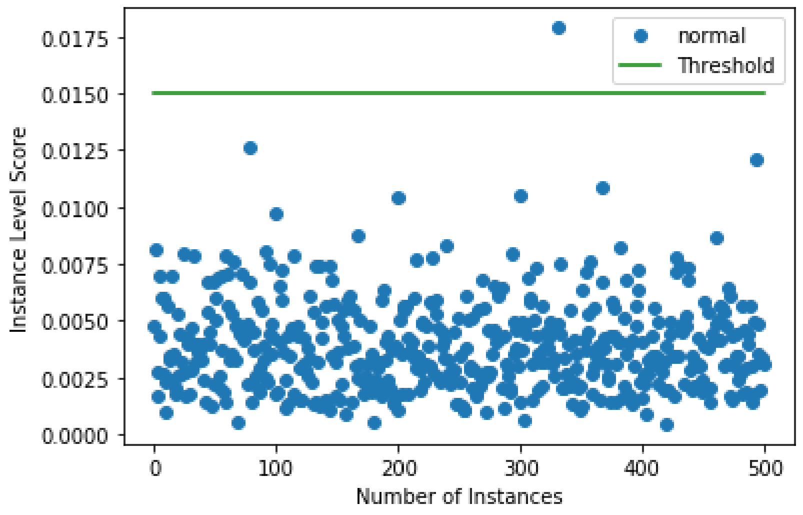
```
(500, 32, 32, 3)
```

```
[9]: od_preds = od.predict(X,
                           outlier_type='instance',    # use 'feature' or 'instance' level
                           return_feature_score=True,  # scores used to determine outliers
                           return_instance_score=True)
     print(list(od_preds['data'].keys()))
```

```
['instance_score', 'feature_score', 'is_outlier']
```

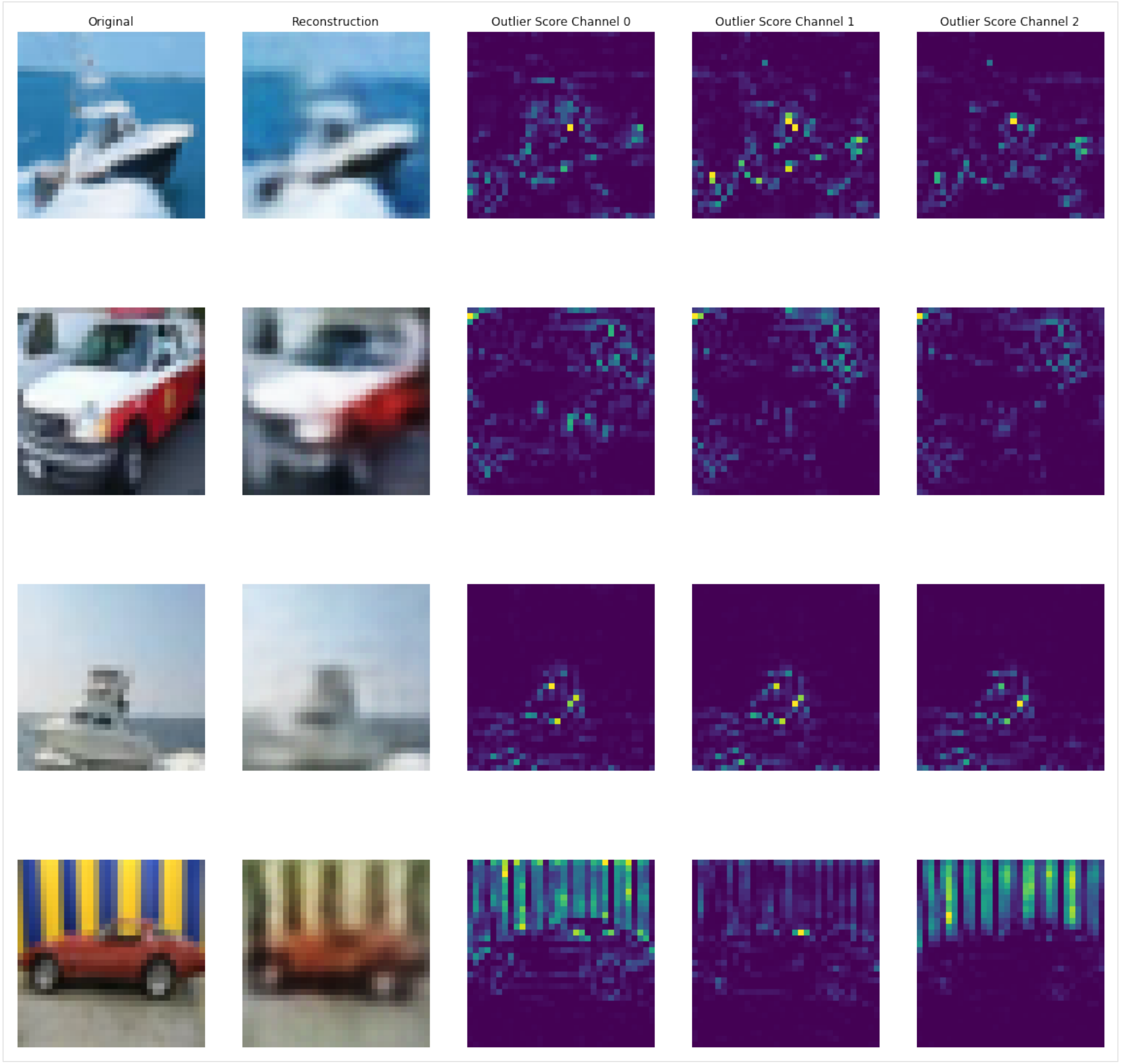
28.6.1 Plot instance level outlier scores

```
[10]: target = np.zeros(X.shape[0],).astype(int)  # all normal CIFAR10 training instances
      labels = ['normal', 'outlier']
      plot_instance_score(od_preds, target, labels, od.threshold)
```



28.6.2 Visualize predictions

```
[11]: X_recon = od.vae(X).numpy()
plot_feature_outlier_image(od_preds,
                           X,
                           X_recon=X_recon,
                           instance_ids=[8, 60, 100, 330],  # pass a list with␣
 ↪indices of instances to display
                           max_instances=5,  # max nb of instances to display
                           outliers_only=False)  # only show outlier predictions
```



28.7 Predict outliers on perturbed CIFAR images

We perturb CIFAR images by adding random noise to patches (masks) of the image. For each mask size in `n_mask_sizes`, sample `n_masks` and apply those to each of the `n_imgs` images. Then we predict outliers on the masked instances:

```
[12]: # nb of predictions per image: n_masks * n_mask_sizes
n_mask_sizes = 10
n_masks = 20
n_imgs = 50
```

Define masks and get images:

```
[13]: mask_sizes = [(2*n,2*n) for n in range(1,n_mask_sizes+1)]
      print(mask_sizes)
      img_ids = np.arange(n_imgs)
      X_orig = X[img_ids].reshape(img_ids.shape[0], 32, 32, 3)
      print(X_orig.shape)
```

```
[(2, 2), (4, 4), (6, 6), (8, 8), (10, 10), (12, 12), (14, 14), (16, 16), (18, 18),␣
↪(20, 20)]
(50, 32, 32, 3)
```

Calculate instance level outlier scores:

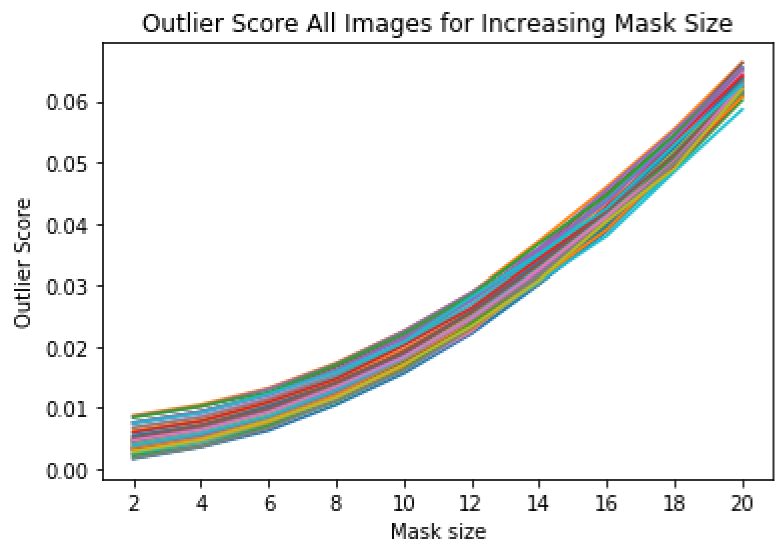
```
[14]: all_img_scores = []
      for i in tqdm(range(X_orig.shape[0])):
          img_scores = np.zeros((len(mask_sizes),))
          for j, mask_size in enumerate(mask_sizes):
              # create masked instances
              X_mask, mask = apply_mask(X_orig[i].reshape(1, 32, 32, 3),
                                        mask_size=mask_size,
                                        n_masks=n_masks,
                                        channels=[0,1,2],
                                        mask_type='normal',
                                        noise_distr=(0,1),
                                        clip_rng=(0,1))
              # predict outliers
              od_preds_mask = od.predict(X_mask)
              score = od_preds_mask['data']['instance_score']
              # store average score over `n_masks` for a given mask size
              img_scores[j] = np.mean(score)
          all_img_scores.append(img_scores)
```

```
100%|| 50/50 [00:39<00:00,  1.26it/s]
```

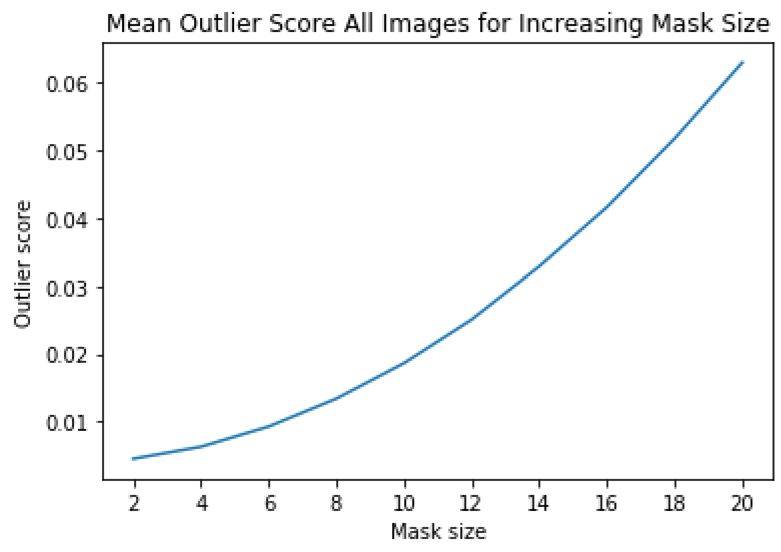
28.7.1 Visualize outlier scores vs. mask sizes

```
[15]: x_plt = [mask[0] for mask in mask_sizes]
```

```
[16]: for ais in all_img_scores:
          plt.plot(x_plt, ais)
          plt.xticks(x_plt)
      plt.title('Outlier Score All Images for Increasing Mask Size')
      plt.xlabel('Mask size')
      plt.ylabel('Outlier Score')
      plt.show()
```



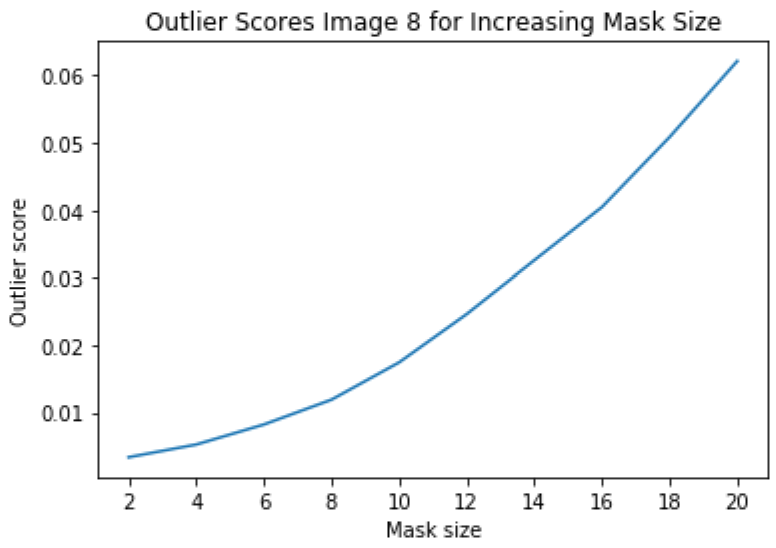
[17]:
```
ais_np = np.zeros((len(all_img_scores), all_img_scores[0].shape[0]))
for i, ais in enumerate(all_img_scores):
    ais_np[i, :] = ais
ais_mean = np.mean(ais_np, axis=0)
plt.title('Mean Outlier Score All Images for Increasing Mask Size')
plt.xlabel('Mask size')
plt.ylabel('Outlier score')
plt.plot(x_plt, ais_mean)
plt.xticks(x_plt)
plt.show()
```



28.7.2 Investigate instance level outlier

```
[18]: i = 8  # index of instance to look at
```

```
[19]: plt.plot(x_plt, all_img_scores[i])
plt.xticks(x_plt)
plt.title('Outlier Scores Image {} for Increasing Mask Size'.format(i))
plt.xlabel('Mask size')
plt.ylabel('Outlier score')
plt.show()
```

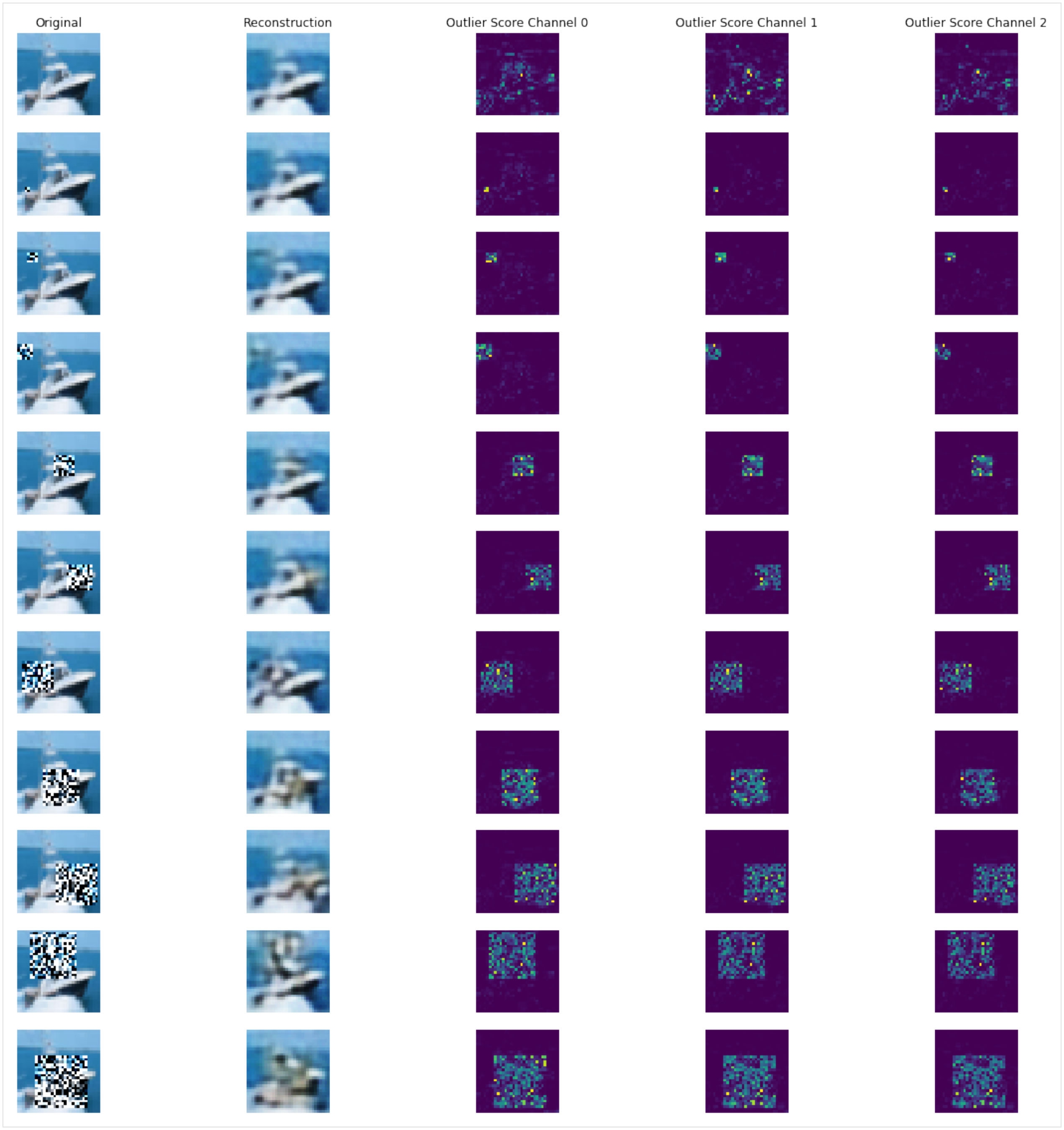


Reconstruction of masked images and outlier scores per channel:

```
[20]: all_X_mask = []
X_i = X_orig[i].reshape(1, 32, 32, 3)
all_X_mask.append(X_i)
# apply masks
for j, mask_size in enumerate(mask_sizes):
    # create masked instances
    X_mask, mask = apply_mask(X_i,
                              mask_size=mask_size,
                              n_masks=1,  # just 1 for visualization purposes
                              channels=[0,1,2],
                              mask_type='normal',
                              noise_distr=(0,1),
                              clip_rng=(0,1))
    all_X_mask.append(X_mask)
all_X_mask = np.concatenate(all_X_mask, axis=0)
all_X_recon = od.vae(all_X_mask).numpy()
od_preds = od.predict(all_X_mask)
```

Visualize:

```
[21]: plot_feature_outlier_image(od_preds,
                           all_X_mask,
                           X_recon=all_X_recon,
                           max_instances=all_X_mask.shape[0],
                           n_channels=3)
```


Original
Reconstruction
Outlier Score Channel 0
Outlier Score Channel 1
Outlier Score Channel 2

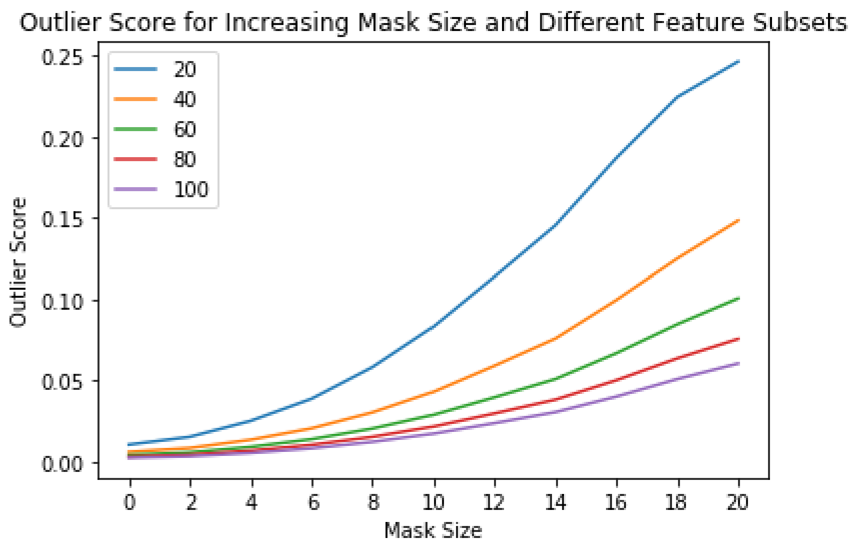
28.8 Predict outliers on a subset of features

The sensitivity of the outlier detector can not only be controlled via the `threshold`, but also by selecting the percentage of the features used for the instance level outlier score computation. For instance, we might want to flag outliers if 40% of the features (pixels for images) have an average outlier score above the threshold. This is possible via the `outlier_perc` argument in the `predict` function. It specifies the percentage of the features that are used for outlier detection, sorted in descending outlier score order.

[22]:
```
perc_list = [20, 40, 60, 80, 100]

all_perc_scores = []
for perc in perc_list:
    od_preds_perc = od.predict(all_X_mask, outlier_perc=perc)
    iscore = od_preds_perc['data']['instance_score']
    all_perc_scores.append(iscore)
```

Visualize outlier scores vs. mask sizes and percentage of features used:

[23]:
```
x_plt = [0] + x_plt
for aps in all_perc_scores:
    plt.plot(x_plt, aps)
    plt.xticks(x_plt)
plt.legend(perc_list)
plt.title('Outlier Score for Increasing Mask Size and Different Feature Subsets')
plt.xlabel('Mask Size')
plt.ylabel('Outlier Score')
plt.show()
```



28.9 Infer outlier threshold value

Finding good threshold values can be tricky since they are typically not easy to interpret. The `infer_threshold` method helps finding a sensible value. We need to pass a batch of instances `X` and specify what percentage of those we consider to be normal via `threshold_perc`.

[24]:
```
print('Current threshold: {}'.format(od.threshold))
od.infer_threshold(X, threshold_perc=99)  # assume 1% of the training data are␣
↪outliers
print('New threshold: {}'.format(od.threshold))
```

```
Current threshold: 0.015
New threshold: 0.010383214280009267
```

CHAPTER

TWENTYNINE

AE OUTLIER DETECTION ON CIFAR10

29.1 Method

The Auto-Encoder (AE) outlier detector is first trained on a batch of unlabeled, but normal (inlier) data. Unsupervised training is desireable since labeled data is often scarce. The AE detector tries to reconstruct the input it receives. If the input data cannot be reconstructed well, the reconstruction error is high and the data can be flagged as an outlier. The reconstruction error is measured as the mean squared error (MSE) between the input and the reconstructed instance.

29.1.1 Dataset

CIFAR10 consists of 60,000 32 by 32 RGB images equally distributed over 10 classes.

[1]:
```
import logging
import matplotlib.pyplot as plt
import numpy as np
import os
import tensorflow as tf
tf.keras.backend.clear_session()
from tensorflow.keras.layers import Conv2D, Conv2DTranspose, \
    Dense, Layer, Reshape, InputLayer, Flatten
from tqdm import tqdm

from alibi_detect.od import OutlierAE
from alibi_detect.utils.fetching import fetch_detector
from alibi_detect.utils.perturbation import apply_mask
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.utils.visualize import plot_instance_score, plot_feature_outlier_
↪image

logger = tf.get_logger()
logger.setLevel(logging.ERROR)
```

```
ERROR:fbprophet:Importing plotly failed. Interactive plots will not work.
```

29.1.2 Load CIFAR10 data

```
[2]: train, test = tf.keras.datasets.cifar10.load_data()
X_train, y_train = train
X_test, y_test = test

X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

```
(50000, 32, 32, 3) (50000, 1) (10000, 32, 32, 3) (10000, 1)
```

29.1.3 Load or define outlier detector

The pretrained outlier and adversarial detectors used in the example notebooks can be found here. You can use the built-in `fetch_detector` function which saves the pre-trained models in a local directory `filepath` and loads the detector. Alternatively, you can train a detector from scratch:

```
[3]: load_outlier_detector = True
```

```
[4]: filepath = 'my_path'  # change to (absolute) directory where model is downloaded
if load_outlier_detector:  # load pretrained outlier detector
    detector_type = 'outlier'
    dataset = 'cifar10'
    detector_name = 'OutlierAE'
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
else:  # define model, initialize, train and save outlier detector
    encoding_dim = 1024

    encoder_net = tf.keras.Sequential(
      [
          InputLayer(input_shape=(32, 32, 3)),
          Conv2D(64, 4, strides=2, padding='same', activation=tf.nn.relu),
          Conv2D(128, 4, strides=2, padding='same', activation=tf.nn.relu),
          Conv2D(512, 4, strides=2, padding='same', activation=tf.nn.relu),
          Flatten(),
          Dense(encoding_dim,)
      ])

    decoder_net = tf.keras.Sequential(
      [
          InputLayer(input_shape=(encoding_dim,)),
          Dense(4*4*128),
          Reshape(target_shape=(4, 4, 128)),
          Conv2DTranspose(256, 4, strides=2, padding='same', activation=tf.nn.relu),
          Conv2DTranspose(64, 4, strides=2, padding='same', activation=tf.nn.relu),
          Conv2DTranspose(3, 4, strides=2, padding='same', activation='sigmoid')
      ])

    # initialize outlier detector
    od = OutlierAE(threshold=.015,  # threshold for outlier score
                    encoder_net=encoder_net,  # can also pass AE model instead
                    decoder_net=decoder_net,  # of separate encoder and decoder
                    )
    # train
```

(continues on next page)

(continued from previous page)

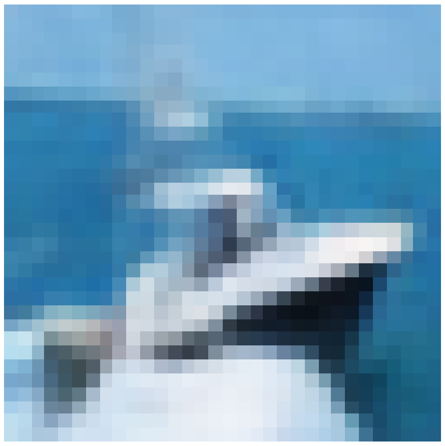
```
    od.fit(X_train,
           epochs=50,
           verbose=True)

    # save the trained outlier detector
    save_detector(od, filepath)
```

29.1.4 Check quality AE model

```
[5]: idx = 8
     X = X_train[idx].reshape(1, 32, 32, 3)
     X_recon = od.ae(X)
```

```
[6]: plt.imshow(X.reshape(32, 32, 3))
     plt.axis('off')
     plt.show()
```

```
[7]: plt.imshow(X_recon.numpy().reshape(32, 32, 3))
     plt.axis('off')
     plt.show()
```



29.1.5 Check outliers on original CIFAR images

[8]:
```
X = X_train[:500]
print(X.shape)
```

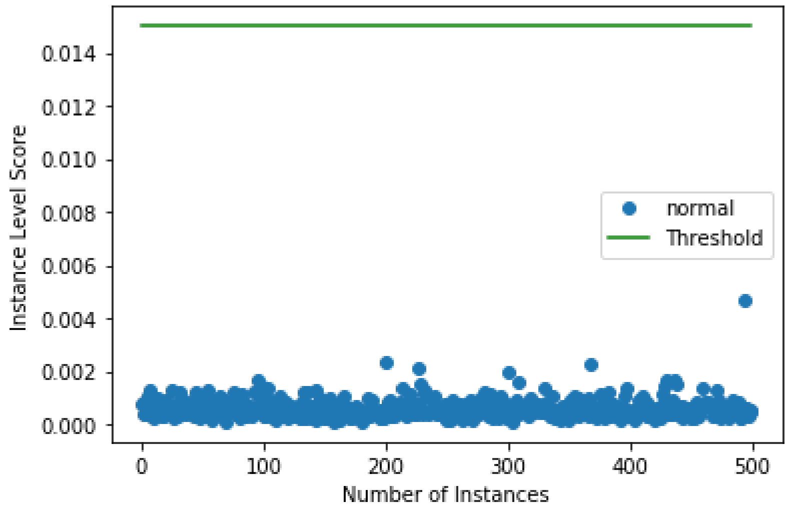
```
(500, 32, 32, 3)
```

[9]:
```
od_preds = od.predict(X,
                      outlier_type='instance',    # use 'feature' or 'instance' level
                      return_feature_score=True,  # scores used to determine outliers
                      return_instance_score=True)
print(list(od_preds['data'].keys()))
```

```
['instance_score', 'feature_score', 'is_outlier']
```

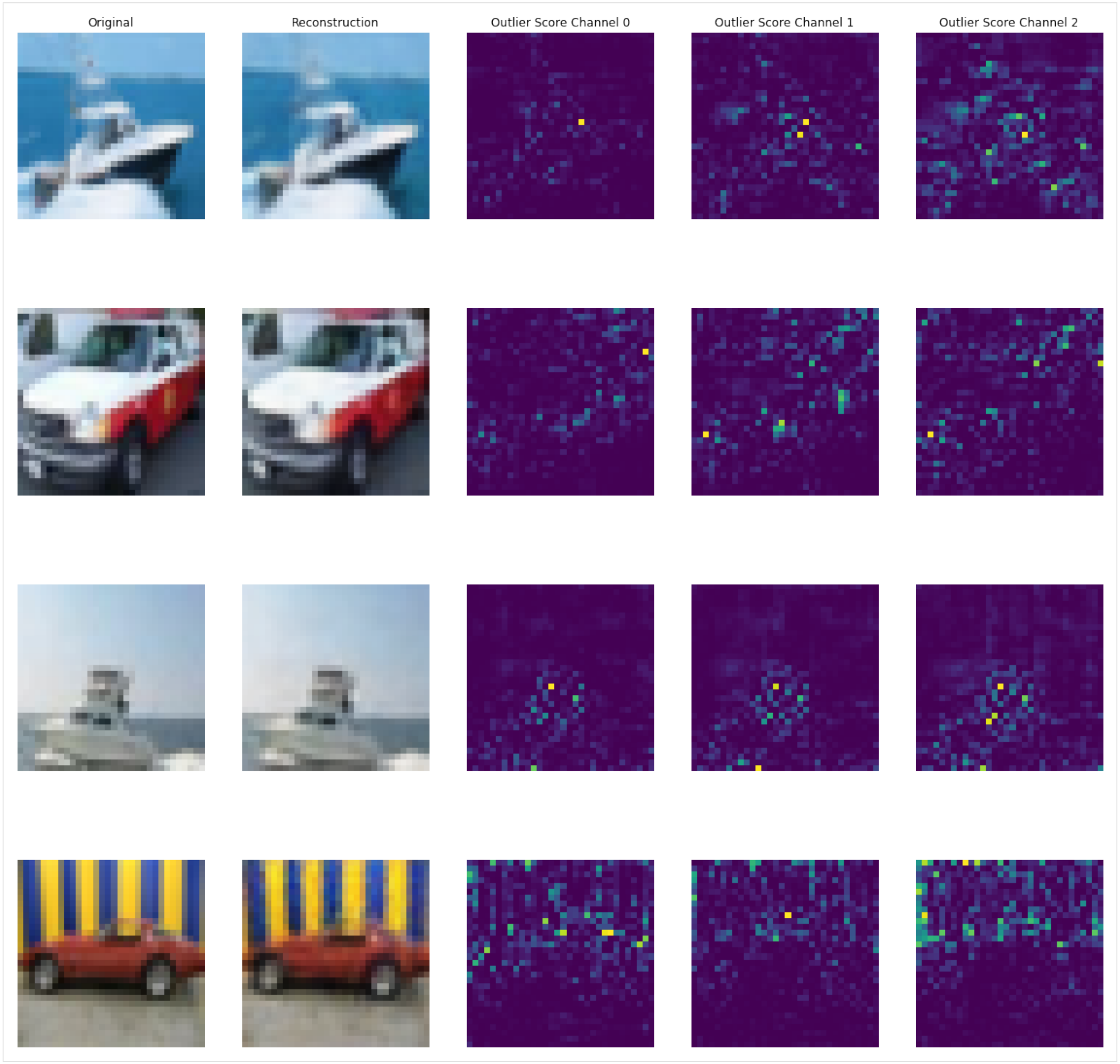
29.2 Plot instance level outlier scores

[10]:
```
target = np.zeros(X.shape[0],).astype(int)  # all normal CIFAR10 training instances
labels = ['normal', 'outlier']
plot_instance_score(od_preds, target, labels, od.threshold)
```



29.3 Visualize predictions

[11]:
```
X_recon = od.ae(X).numpy()
plot_feature_outlier_image(od_preds,
                           X,
                           X_recon=X_recon,
                           instance_ids=[8, 60, 100, 330],  # pass a list with␣
↪indices of instances to display
                           max_instances=5,  # max nb of instances to display
                           outliers_only=False)  # only show outlier predictions
```



29.3.1 Predict outliers on perturbed CIFAR images

We perturb CIFAR images by adding random noise to patches (masks) of the image. For each mask size in `n_mask_sizes`, sample `n_masks` and apply those to each of the `n_imgs` images. Then we predict outliers on the masked instances:

[12]:
```
# nb of predictions per image: n_masks * n_mask_sizes
n_mask_sizes = 10
n_masks = 20
n_imgs = 50
```

Define masks and get images:

```
[13]: mask_sizes = [(2*n,2*n) for n in range(1,n_mask_sizes+1)]
      print(mask_sizes)
      img_ids = np.arange(n_imgs)
      X_orig = X[img_ids].reshape(img_ids.shape[0], 32, 32, 3)
      print(X_orig.shape)
```

```
[(2, 2), (4, 4), (6, 6), (8, 8), (10, 10), (12, 12), (14, 14), (16, 16), (18, 18),␣
↪(20, 20)]
(50, 32, 32, 3)
```

Calculate instance level outlier scores:

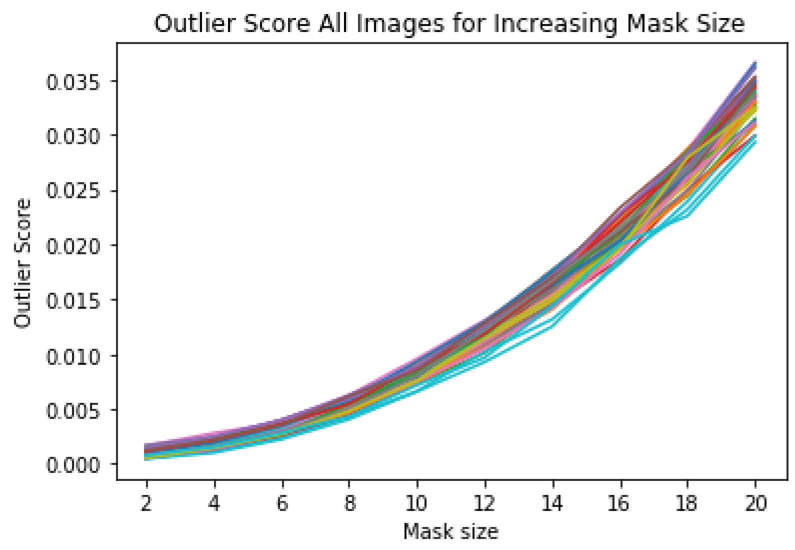
```
[14]: all_img_scores = []
      for i in tqdm(range(X_orig.shape[0])):
          img_scores = np.zeros((len(mask_sizes),))
          for j, mask_size in enumerate(mask_sizes):
              # create masked instances
              X_mask, mask = apply_mask(X_orig[i].reshape(1, 32, 32, 3),
                                        mask_size=mask_size,
                                        n_masks=n_masks,
                                        channels=[0,1,2],
                                        mask_type='normal',
                                        noise_distr=(0,1),
                                        clip_rng=(0,1))
              # predict outliers
              od_preds_mask = od.predict(X_mask)
              score = od_preds_mask['data']['instance_score']
              # store average score over `n_masks` for a given mask size
              img_scores[j] = np.mean(score)
          all_img_scores.append(img_scores)
```

```
100%|| 50/50 [00:17<00:00,  2.90it/s]
```

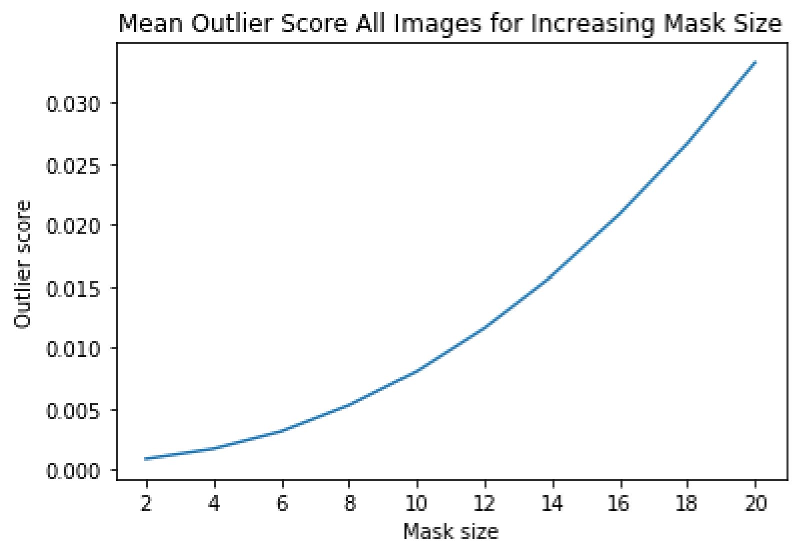
29.4 Visualize outlier scores vs. mask sizes

```
[15]: x_plt = [mask[0] for mask in mask_sizes]
```

```
[16]: for ais in all_img_scores:
          plt.plot(x_plt, ais)
          plt.xticks(x_plt)
      plt.title('Outlier Score All Images for Increasing Mask Size')
      plt.xlabel('Mask size')
      plt.ylabel('Outlier Score')
      plt.show()
```



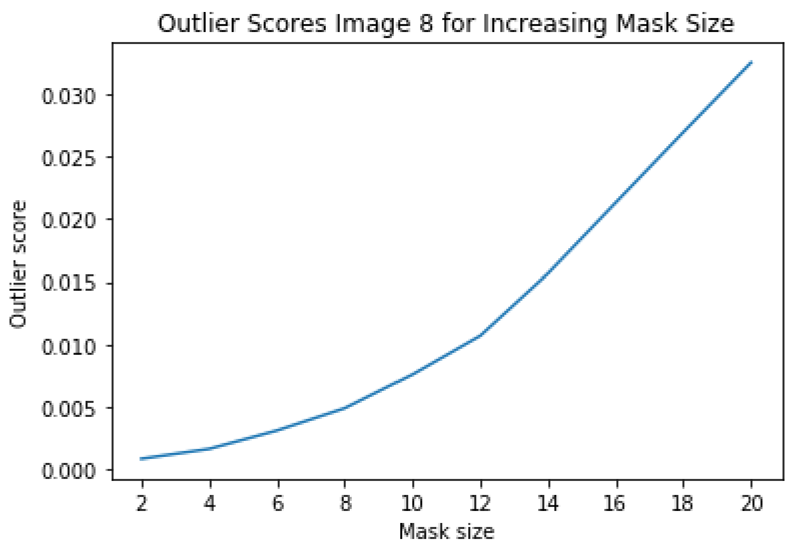
```
[17]: ais_np = np.zeros((len(all_img_scores), all_img_scores[0].shape[0]))
      for i, ais in enumerate(all_img_scores):
          ais_np[i, :] = ais
      ais_mean = np.mean(ais_np, axis=0)
      plt.title('Mean Outlier Score All Images for Increasing Mask Size')
      plt.xlabel('Mask size')
      plt.ylabel('Outlier score')
      plt.plot(x_plt, ais_mean)
      plt.xticks(x_plt)
      plt.show()
```



29.5 Investigate instance level outlier

[18]:
```
i = 8  # index of instance to look at
```

[19]:
```
plt.plot(x_plt, all_img_scores[i])
plt.xticks(x_plt)
plt.title('Outlier Scores Image {} for Increasing Mask Size'.format(i))
plt.xlabel('Mask size')
plt.ylabel('Outlier score')
plt.show()
```



Reconstruction of masked images and outlier scores per channel:

[20]:
```
all_X_mask = []
X_i = X_orig[i].reshape(1, 32, 32, 3)
all_X_mask.append(X_i)
# apply masks
for j, mask_size in enumerate(mask_sizes):
    # create masked instances
    X_mask, mask = apply_mask(X_i,
                              mask_size=mask_size,
                              n_masks=1,  # just 1 for visualization purposes
                              channels=[0,1,2],
                              mask_type='normal',
                              noise_distr=(0,1),
                              clip_rng=(0,1))
    all_X_mask.append(X_mask)
all_X_mask = np.concatenate(all_X_mask, axis=0)
all_X_recon = od.ae(all_X_mask).numpy()
od_preds = od.predict(all_X_mask)
```

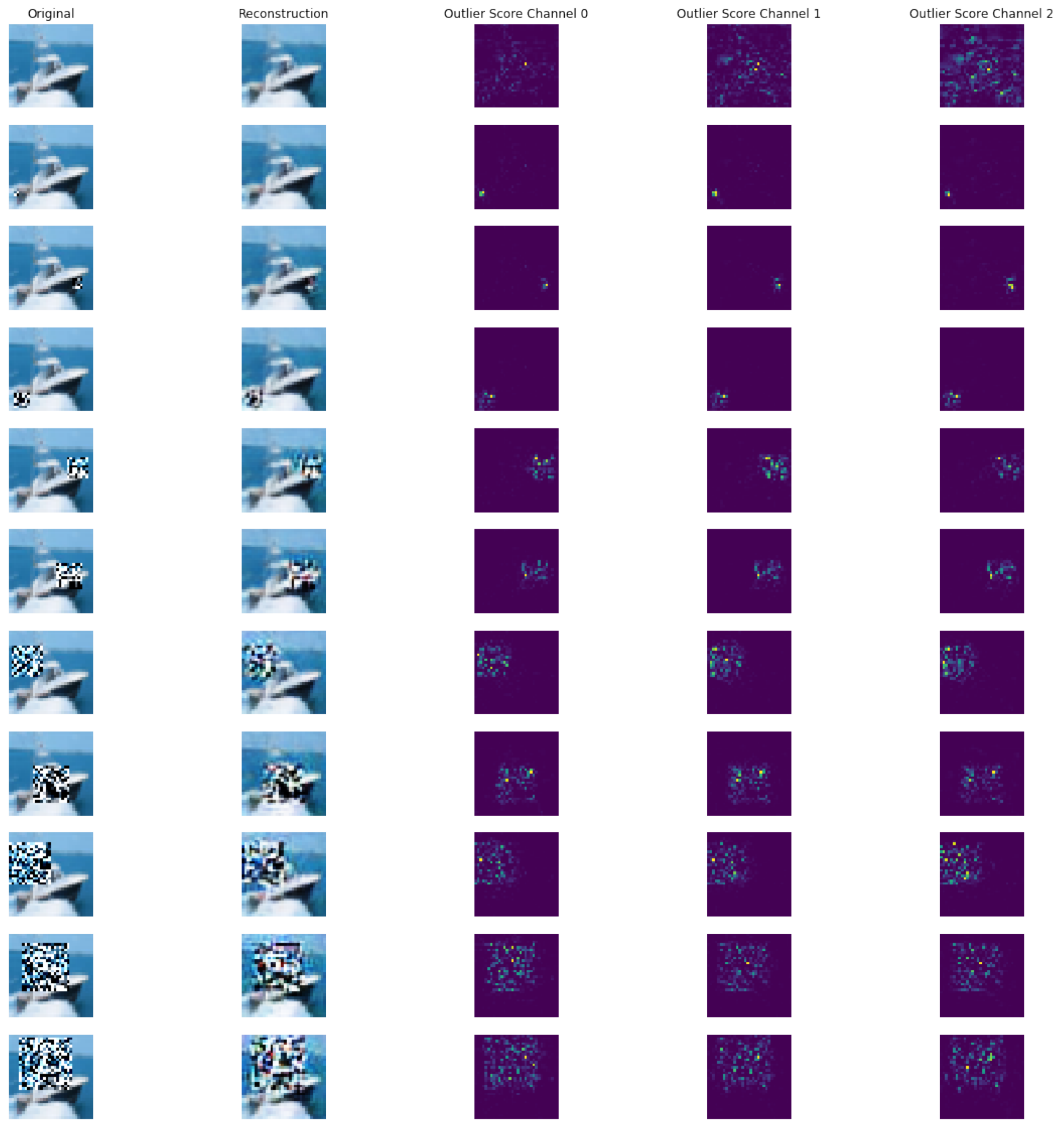
Visualize:

[21]:
```
plot_feature_outlier_image(od_preds,
                           all_X_mask,
                           X_recon=all_X_recon,
                           max_instances=all_X_mask.shape[0],
```

(continues on next page)

(continued from previous page)

```
                              n_channels=3)
```



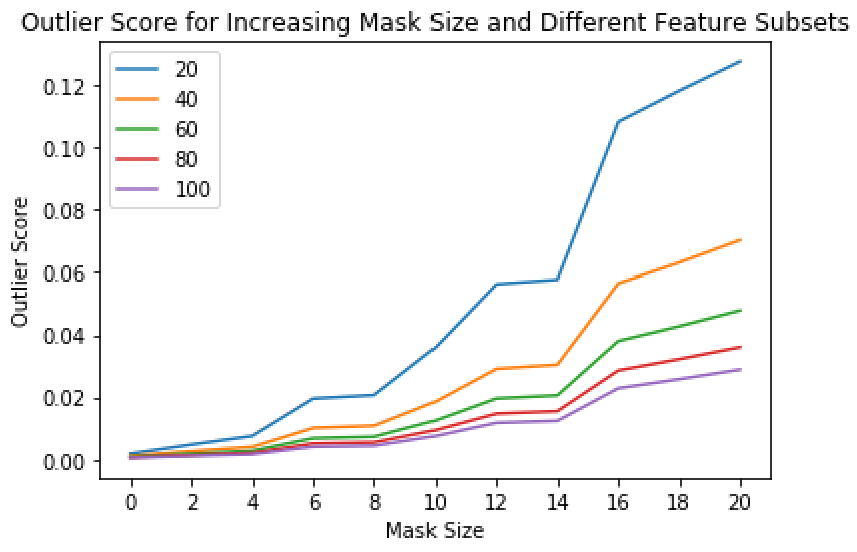
29.5.1 Predict outliers on a subset of features

The sensitivity of the outlier detector can not only be controlled via the `threshold`, but also by selecting the percentage of the features used for the instance level outlier score computation. For instance, we might want to flag outliers if 40% of the features (pixels for images) have an average outlier score above the threshold. This is possible via the `outlier_perc` argument in the `predict` function. It specifies the percentage of the features that are used for outlier detection, sorted in descending outlier score order.

[22]:
```
perc_list = [20, 40, 60, 80, 100]

all_perc_scores = []
for perc in perc_list:
    od_preds_perc = od.predict(all_X_mask, outlier_perc=perc)
    iscore = od_preds_perc['data']['instance_score']
    all_perc_scores.append(iscore)
```

Visualize outlier scores vs. mask sizes and percentage of features used:

[23]:
```
x_plt = [0] + x_plt
for aps in all_perc_scores:
    plt.plot(x_plt, aps)
    plt.xticks(x_plt)
plt.legend(perc_list)
plt.title('Outlier Score for Increasing Mask Size and Different Feature Subsets')
plt.xlabel('Mask Size')
plt.ylabel('Outlier Score')
plt.show()
```



29.5.2 Infer outlier threshold value

Finding good threshold values can be tricky since they are typically not easy to interpret. The `infer_threshold` method helps finding a sensible value. We need to pass a batch of instances `X` and specify what percentage of those we consider to be normal via `threshold_perc`.

```
[24]: print('Current threshold: {}'.format(od.threshold))
      od.infer_threshold(X, threshold_perc=99)  # assume 1% of the training data are␣
      ↪outliers
      print('New threshold: {}'.format(od.threshold))
```

```
Current threshold: 0.015
New threshold: 0.0017021061840932791
```

CHAPTER

THIRTY

AEGMM AND VAEGMM OUTLIER DETECTION ON KDD CUP '99 DATASET

30.1 Method

The *AEGMM* method follows the Deep Autoencoding Gaussian Mixture Model for Unsupervised Anomaly Detection ICLR 2018 paper. The encoder compresses the data while the reconstructed instances generated by the decoder are used to create additional features based on the reconstruction error between the input and the reconstructions. These features are combined with encodings and fed into a Gaussian Mixture Model (GMM). Training of the *AEGMM* model is unsupervised on *normal* (inlier) data. The sample energy of the GMM can then be used to determine whether an instance is an outlier (*high sample energy*) or not (*low sample energy*). *VAEGMM* on the other hand uses a variational autoencoder instead of a plain autoencoder.

30.2 Dataset

The outlier detector needs to detect computer network intrusions using TCP dump data for a local-area network (LAN) simulating a typical U.S. Air Force LAN. A connection is a sequence of TCP packets starting and ending at some well defined times, between which data flows to and from a source IP address to a target IP address under some well defined protocol. Each connection is labeled as either normal, or as an attack.

There are 4 types of attacks in the dataset:

- DOS: denial-of-service, e.g. syn flood;
- R2L: unauthorized access from a remote machine, e.g. guessing password;
- U2R: unauthorized access to local superuser (root) privileges;
- probing: surveillance and other probing, e.g., port scanning.

The dataset contains about 5 million connection records.

There are 3 types of features:

- basic features of individual connections, e.g. duration of connection
- content features within a connection, e.g. number of failed log in attempts
- traffic features within a 2 second window, e.g. number of connections to the same host as the current connection

[1]:
```
import logging
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
import seaborn as sns
from sklearn.metrics import confusion_matrix, f1_score
import tensorflow as tf
tf.keras.backend.clear_session()
from tensorflow.keras.layers import Dense, InputLayer

from alibi_detect.datasets import fetch_kdd
from alibi_detect.models.autoencoder import eucl_cosim_features
from alibi_detect.od import OutlierAEGMM, OutlierVAEGMM
from alibi_detect.utils.data import create_outlier_batch
from alibi_detect.utils.fetching import fetch_detector
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.utils.visualize import plot_instance_score, plot_feature_outlier_
↪tabular, plot_roc

logger = tf.get_logger()
logger.setLevel(logging.ERROR)
```

30.3 Load dataset

We only keep a number of continuous (18 out of 41) features.

[2]:
```
kddcup = fetch_kdd(percent10=True)  # only load 10% of the dataset
print(kddcup.data.shape, kddcup.target.shape)
```

```
(494021, 18) (494021,)
```

Assume that a model is trained on *normal* instances of the dataset (not outliers) and standardization is applied:

[3]:
```
np.random.seed(0)
normal_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=400000,␣
↪perc_outlier=0)
X_train, y_train = normal_batch.data.astype('float32'), normal_batch.target
print(X_train.shape, y_train.shape)
print('{}% outliers'.format(100 * y_train.mean()))
```

```
(400000, 18) (400000,)
0.0% outliers
```

[4]:
```
mean, stdev = X_train.mean(axis=0), X_train.std(axis=0)
```

Apply standardization:

[5]:
```
X_train = (X_train - mean) / stdev
```

30.4 Load or define AEGMM outlier detector

The pretrained outlier and adversarial detectors used in the example notebooks can be found here. You can use the built-in `fetch_detector` function which saves the pre-trained models in a local directory `filepath` and loads the detector. Alternatively, you can train a detector from scratch:

[6]:
```
load_outlier_detector = True
```

[7]:
```
filepath = 'my_path'  # change to directory (absolute path) where model is downloaded
if load_outlier_detector:  # load pretrained outlier detector
    detector_type = 'outlier'
    dataset = 'kddcup'
    detector_name = 'OutlierAEGMM'
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
else:  # define model, initialize, train and save outlier detector
    # the model defined here is similar to the one defined in the original paper
    n_features = X_train.shape[1]
    latent_dim = 1
    n_gmm = 2  # nb of components in GMM

    encoder_net = tf.keras.Sequential(
    [
        InputLayer(input_shape=(n_features,)),
        Dense(60, activation=tf.nn.tanh),
        Dense(30, activation=tf.nn.tanh),
        Dense(10, activation=tf.nn.tanh),
        Dense(latent_dim, activation=None)
    ])

    decoder_net = tf.keras.Sequential(
    [
        InputLayer(input_shape=(latent_dim,)),
        Dense(10, activation=tf.nn.tanh),
        Dense(30, activation=tf.nn.tanh),
        Dense(60, activation=tf.nn.tanh),
        Dense(n_features, activation=None)
    ])

    gmm_density_net = tf.keras.Sequential(
    [
        InputLayer(input_shape=(latent_dim + 2,)),
        Dense(10, activation=tf.nn.tanh),
        Dense(n_gmm, activation=tf.nn.softmax)
    ])

    # initialize outlier detector
    od = OutlierAEGMM(threshold=None,  # threshold for outlier score
                      encoder_net=encoder_net,         # can also pass AEGMM model␣
↪instead
                      decoder_net=decoder_net,         # of separate encoder, decoder
                      gmm_density_net=gmm_density_net, # and gmm density net
                      n_gmm=n_gmm,
                      recon_features=eucl_cosim_features)  # fn used to derive␣
↪features
                                                           # from the reconstructed
                                                           # instances based on cosine
```

(continues on next page)

(continued from previous page)

```
                                                                   # similarity and Eucl␣
↪distance

    # train
    od.fit(X_train,
           epochs=50,
           batch_size=1024,
           save_path=filepath,
           verbose=True)

    # save the trained outlier detector
    save_detector(od, filepath)
```

```
WARNING:alibi_detect.od.aegmm:No threshold level set. Need to infer threshold using␣
↪`infer_threshold`.
```

The warning tells us we still need to set the outlier threshold. This can be done with the `infer_threshold` method. We need to pass a batch of instances and specify what percentage of those we consider to be normal via `threshold_perc`. Let's assume we have some data which we know contains around 5% outliers. The percentage of outliers can be set with `perc_outlier` in the `create_outlier_batch` function.

```
[8]: np.random.seed(0)
perc_outlier = 5
threshold_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=1000,␣
↪perc_outlier=perc_outlier)
X_threshold, y_threshold = threshold_batch.data.astype('float32'), threshold_batch.
↪target
X_threshold = (X_threshold - mean) / stdev
print('{}% outliers'.format(100 * y_threshold.mean()))
```

```
5.0% outliers
```

```
[9]: od.infer_threshold(X_threshold, threshold_perc=100-perc_outlier)
print('New threshold: {}'.format(od.threshold))
```

```
New threshold: 29.55636606216426
```

Save outlier detector with updated threshold:

```
[10]: save_detector(od, filepath)
```

30.5 Detect outliers

We now generate a batch of data with 10% outliers and detect the outliers in the batch.

```
[11]: np.random.seed(1)
outlier_batch = create_outlier_batch(kddcup.data, kddcup.target, n_samples=1000, perc_
↪outlier=10)
X_outlier, y_outlier = outlier_batch.data.astype('float32'), outlier_batch.target
X_outlier = (X_outlier - mean) / stdev
print(X_outlier.shape, y_outlier.shape)
print('{}% outliers'.format(100 * y_outlier.mean()))
```

```
(1000, 18) (1000,)
10.0% outliers
```

Predict outliers:

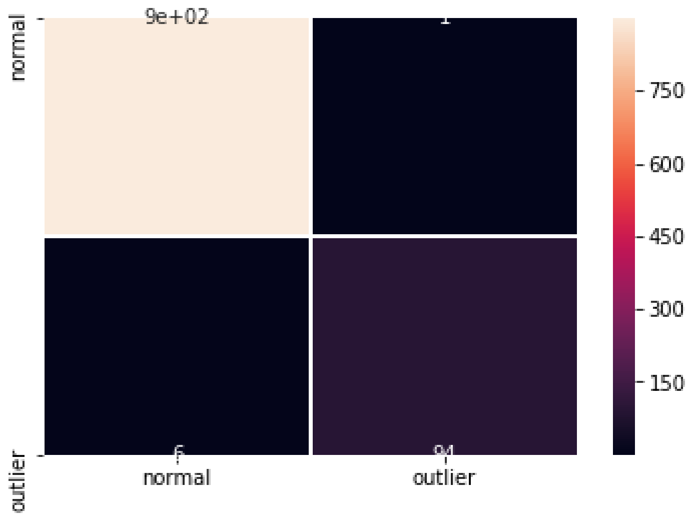
```
[12]: od_preds = od.predict(X_outlier, return_instance_score=True)
```

30.6 Display results

F1 score and confusion matrix:

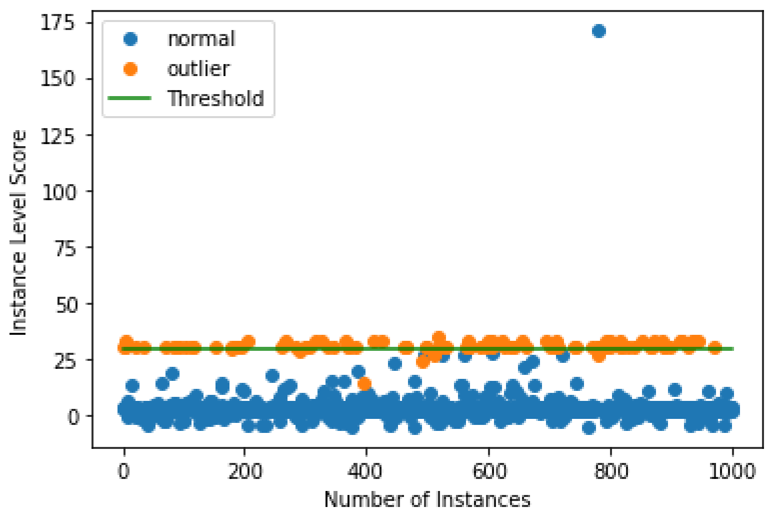
```
[13]: labels = outlier_batch.target_names
y_pred = od_preds['data']['is_outlier']
f1 = f1_score(y_outlier, y_pred)
print('F1 score: {:.4f}'.format(f1))
cm = confusion_matrix(y_outlier, y_pred)
df_cm = pd.DataFrame(cm, index=labels, columns=labels)
sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
plt.show()
```

```
F1 score: 0.9641
```



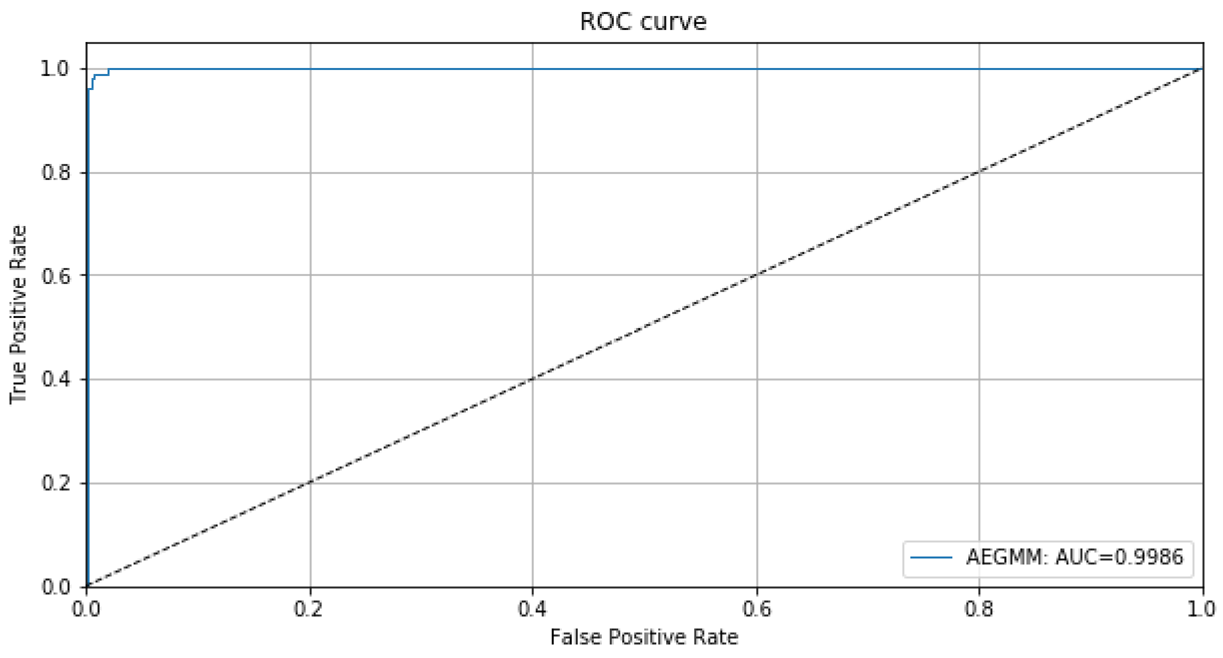
Plot instance level outlier scores vs. the outlier threshold:

```
[14]: plot_instance_score(od_preds, y_outlier, labels, od.threshold, ylim=(None, None))
```



We can also plot the ROC curve for the outlier scores of the detector:

```
[15]: roc_data = {'AEGMM': {'scores': od_preds['data']['instance_score'], 'labels': y_
      ↪outlier}}
      plot_roc(roc_data)
```

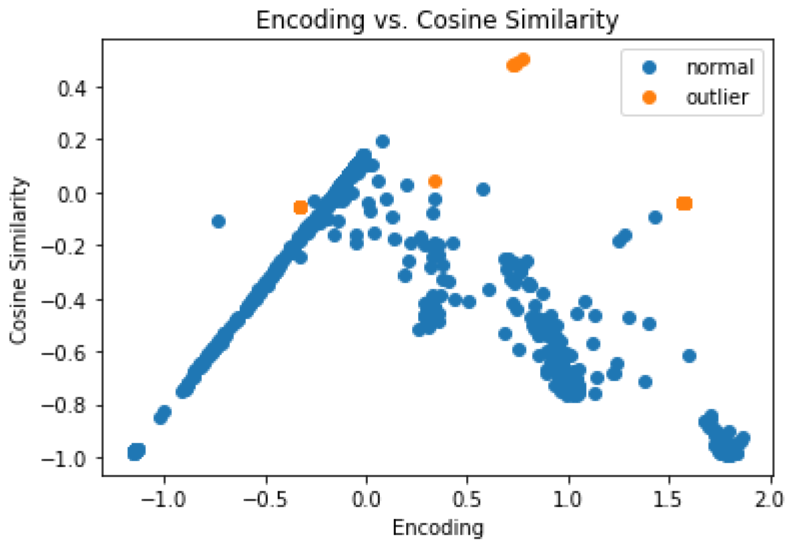


30.7 Investigate results

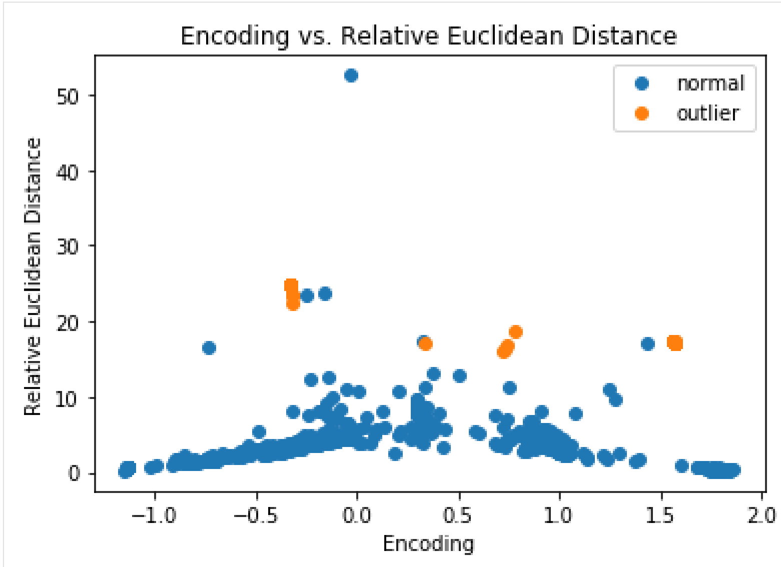
We can visualize the encodings of the instances in the latent space and the features derived from the instance reconstructions by the decoder. The encodings and features are then fed into the GMM density network.

```
[16]: enc = od.aegmm.encoder(X_outlier)  # encoding
X_recon = od.aegmm.decoder(enc)  # reconstructed instances
recon_features = od.aegmm.recon_features(X_outlier, X_recon)  # reconstructed features
```

```
[17]: df = pd.DataFrame(dict(enc=enc[:, 0].numpy(),
                       cos=recon_features[:, 0].numpy(),
                       eucl=recon_features[:, 1].numpy(),
                       label=y_outlier))

groups = df.groupby('label')
fig, ax = plt.subplots()
for name, group in groups:
    ax.plot(group.enc, group.cos, marker='o',
            linestyle='', ms=6, label=labels[name])
plt.title('Encoding vs. Cosine Similarity')
plt.xlabel('Encoding')
plt.ylabel('Cosine Similarity')
ax.legend()
plt.show()
```



```
[18]: fig, ax = plt.subplots()
for name, group in groups:
    ax.plot(group.enc, group.eucl, marker='o',
            linestyle='', ms=6, label=labels[name])
plt.title('Encoding vs. Relative Euclidean Distance')
plt.xlabel('Encoding')
plt.ylabel('Relative Euclidean Distance')
ax.legend()
plt.show()
```



A lot of the outliers are already separated well in the latent space.

30.8 Use VAEGMM outlier detector

We can again instantiate the pretrained VAEGMM detector from the Google Cloud Bucket. You can use the built-in `fetch_detector` function which saves the pre-trained models in a local directory `filepath` and loads the detector. Alternatively, you can train a detector from scratch:

[19]:
```
load_outlier_detector = True
```

[20]:
```
filepath = 'my_path'  # change to directory (absolute path) where model is downloaded
if load_outlier_detector:  # load pretrained outlier detector
    detector_type = 'outlier'
    dataset = 'kddcup'
    detector_name = 'OutlierVAEGMM'
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
else:  # define model, initialize, train and save outlier detector
    # the model defined here is similar to the one defined in
    # the OutlierVAE notebook
    n_features = X_train.shape[1]
    latent_dim = 2
    n_gmm = 2

    encoder_net = tf.keras.Sequential(
    [
        InputLayer(input_shape=(n_features,)),
        Dense(20, activation=tf.nn.relu),
        Dense(15, activation=tf.nn.relu),
        Dense(7, activation=tf.nn.relu)
    ])

    decoder_net = tf.keras.Sequential(
    [
        InputLayer(input_shape=(latent_dim,)),
```

(continues on next page)

(continued from previous page)

```
        Dense(7, activation=tf.nn.relu),
        Dense(15, activation=tf.nn.relu),
        Dense(20, activation=tf.nn.relu),
        Dense(n_features, activation=None)
    ])

    gmm_density_net = tf.keras.Sequential(
    [
        InputLayer(input_shape=(latent_dim + 2,)),
        Dense(10, activation=tf.nn.relu),
        Dense(n_gmm, activation=tf.nn.softmax)
    ])


    # initialize outlier detector
    od = OutlierVAEGMM(threshold=None,
                       encoder_net=encoder_net,
                       decoder_net=decoder_net,
                       gmm_density_net=gmm_density_net,
                       n_gmm=n_gmm,
                       latent_dim=latent_dim,
                       samples=10,
                       recon_features=eucl_cosim_features)

    # train
    od.fit(X_train,
           epochs=50,
           batch_size=1024,
           cov_elbo=dict(sim=.0025),  # standard deviation assumption
           verbose=True)            # for elbo training

    # save the trained outlier detector
    save_detector(od, filepath)
```

```
WARNING:alibi_detect.od.vaegmm:No threshold level set. Need to infer threshold using␣
↪`infer_threshold`.
```

Need to infer the threshold again:

[21]:
```
od.infer_threshold(X_threshold, threshold_perc=100-perc_outlier)
print('New threshold: {}'.format(od.threshold))
```

```
New threshold: 11.703912305831903
```

Save outlier detector with updated threshold:

[22]:
```
save_detector(od, filepath)
```

30.9 Detect outliers and display results

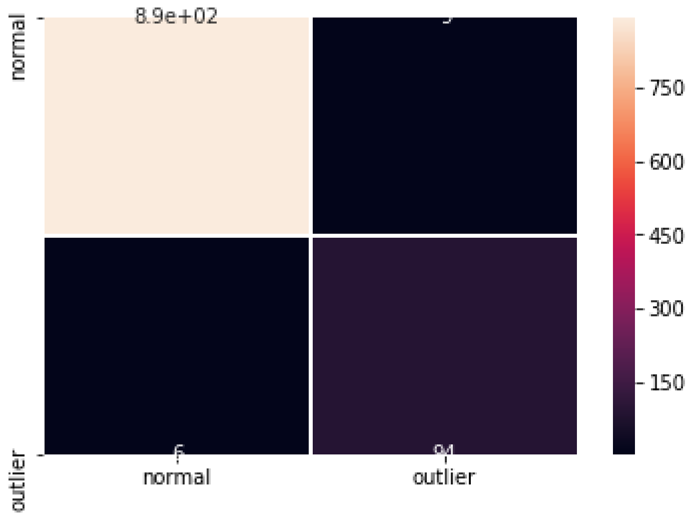
Predict:

```
[23]: od_preds = od.predict(X_outlier, return_instance_score=True)
```

F1 score and confusion matrix:

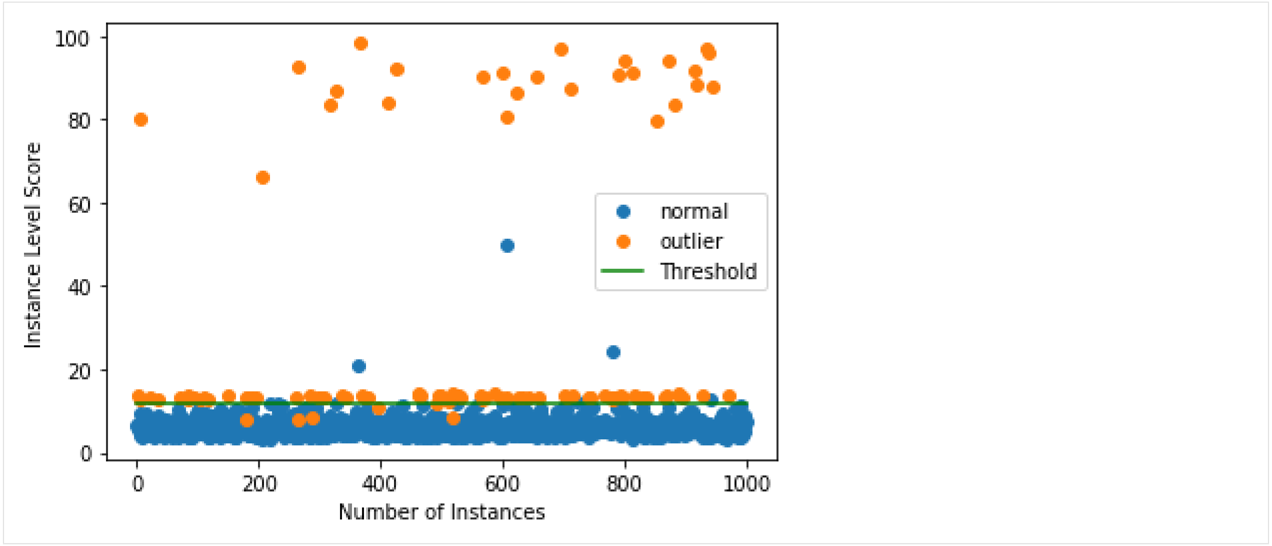
```
[24]: labels = outlier_batch.target_names
y_pred = od_preds['data']['is_outlier']
f1 = f1_score(y_outlier, y_pred)
print('F1 score: {:.4f}'.format(f1))
cm = confusion_matrix(y_outlier, y_pred)
df_cm = pd.DataFrame(cm, index=labels, columns=labels)
sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
plt.show()
```

```
F1 score: 0.9261
```



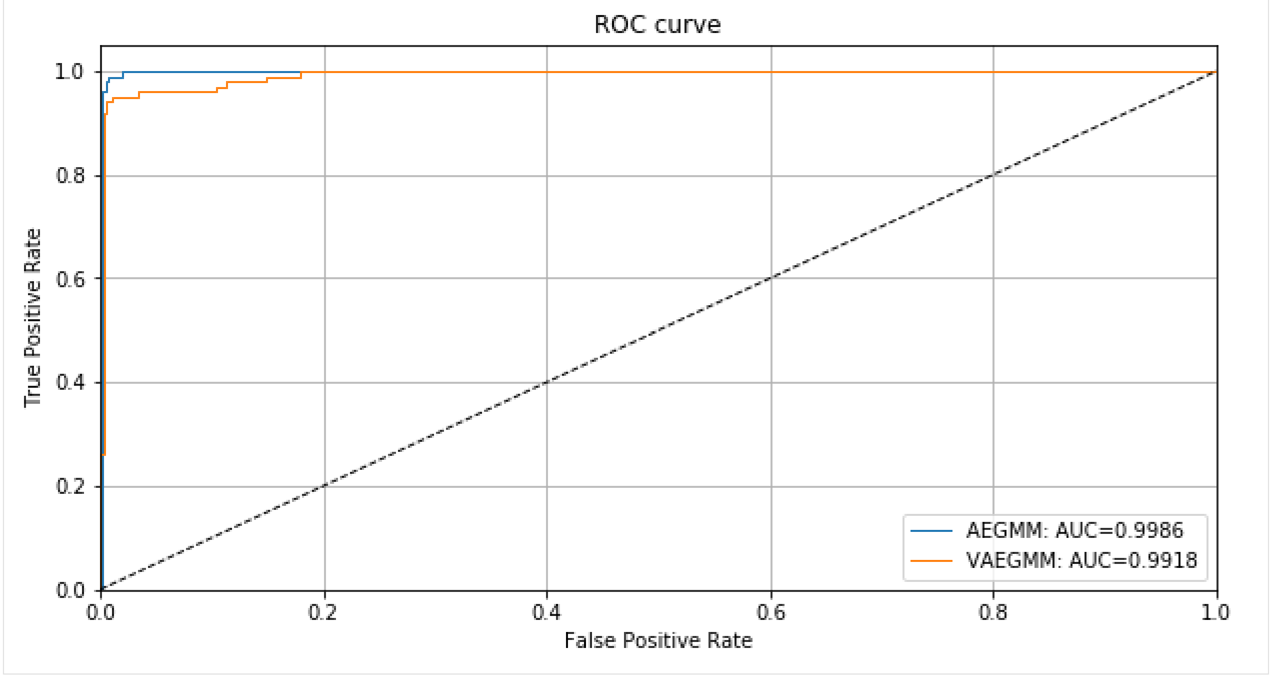
Plot instance level outlier scores vs. the outlier threshold:

```
[25]: plot_instance_score(od_preds, y_outlier, labels, od.threshold, ylim=(None, None))
```



You can zoom in by adjusting the min and max values in `ylim`. We can also compare the VAEGMM ROC curve with AEGMM:

```
[26]: roc_data['VAEGMM'] = {'scores': od_preds['data']['instance_score'], 'labels': y_
↪outlier}
plot_roc(roc_data)
```



CHAPTER

THIRTYONE

LIKELIHOOD RATIO OUTLIER DETECTION WITH PIXELCNN++

31.1 Method

The outlier detector described by Ren et al. (2019) in Likelihood Ratios for Out-of-Distribution Detection uses the likelihood ratio between 2 generative models as the outlier score. One model is trained on the original data while the other is trained on a perturbed version of the dataset. This is based on the observation that the likelihood score for an instance under a generative model can be heavily affected by population level background statistics. The second generative model is therefore trained to capture the background statistics still present in the perturbed data while the semantic features have been erased by the perturbations.

The perturbations are added using an independent and identical Bernoulli distribution with rate μ which substitutes a feature with one of the other possible feature values with equal probability. For images, this means changing a pixel with a different pixel randomly sampled within the 0 to 255 pixel range.

The generative model used in the example is a PixelCNN++, adapted from the official TensorFlow Probability implementation, and available as a standalone model in `from alibi_detect.models.tensorflow import PixelCNN`.

31.2 Dataset

The training set Fashion-MNIST consists of 60,000 28 by 28 grayscale images distributed over 10 classes. The classes represent items of clothing such as shirts or trousers. At test time, we want to distinguish the Fashion-MNIST test set from MNIST, which represents 28 by 28 grayscale numbers from 0 to 9.

[1]:
```
import os
from functools import partial
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
import tensorflow as tf

from alibi_detect.od import LLR
from alibi_detect.models.tensorflow import PixelCNN
from alibi_detect.utils.fetching import fetch_detector
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.utils.prediction import predict_batch
from alibi_detect.utils.visualize import plot_roc
```

31.2.1 Utility Functions

[2]:
```
def load_data(dataset: str) -> tuple:
    if dataset == 'mnist':
        (X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
    elif dataset == 'fashion_mnist':
        (X_train, y_train), (X_test, y_test) = tf.keras.datasets.fashion_mnist.load_
↪data()
    else:
        raise NotImplementedError
    X_train = X_train.astype('float32')
    X_test = X_test.astype('float32')
    y_train = y_train.astype('int64').reshape(-1,)
    y_test = y_test.astype('int64').reshape(-1,)
    if len(X_train.shape) == 3:
        shape = (-1,) + X_train.shape[1:] + (1,)
        X_train = X_train.reshape(shape)
        X_test = X_test.reshape(shape)
    return (X_train, y_train), (X_test, y_test)


def plot_grid_img(X: np.ndarray, figsize: tuple = (10, 6)) -> None:
    n = X.shape[0]
    nrows = int(n**.5)
    ncols = int(np.ceil(n / nrows))
    fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)
    n_subplot = 1
    for r in range(nrows):
        for c in range(ncols):
            plt.subplot(nrows, ncols, n_subplot)
            plt.axis('off')
            plt.imshow(X[n_subplot-1, :, :, 0])
            n_subplot += 1


def plot_grid_logp(idx: list, X: np.ndarray, logp_s: np.ndarray,
                   logp_b: np.ndarray, figsize: tuple = (10, 6)) -> None:
    nrows, ncols = len(idx), 4
    fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)
    n_subplot = 1
    for r in range(nrows):
        plt.subplot(nrows, ncols, n_subplot)
        plt.imshow(X[idx[r], :, :, 0])
        plt.colorbar()
        plt.axis('off')
        if r == 0:
            plt.title('Image')
        n_subplot += 1

        plt.subplot(nrows, ncols, n_subplot)
        plt.imshow(logp_s[idx[r], :, :])
        plt.colorbar()
        plt.axis('off')
        if r == 0:
            plt.title('Semantic Logp')
        n_subplot += 1
```

(continues on next page)

(continued from previous page)

```
        plt.subplot(nrows, ncols, n_subplot)
        plt.imshow(logp_b[idx[r], :, :])
        plt.colorbar()
        plt.axis('off')
        if r == 0:
            plt.title('Background Logp')
        n_subplot += 1

        plt.subplot(nrows, ncols, n_subplot)
        plt.imshow(logp_s[idx[r], :, :] - logp_b[idx[r], :, :])
        plt.colorbar()
        plt.axis('off')
        if r == 0:
            plt.title('LLR')
        n_subplot += 1
```

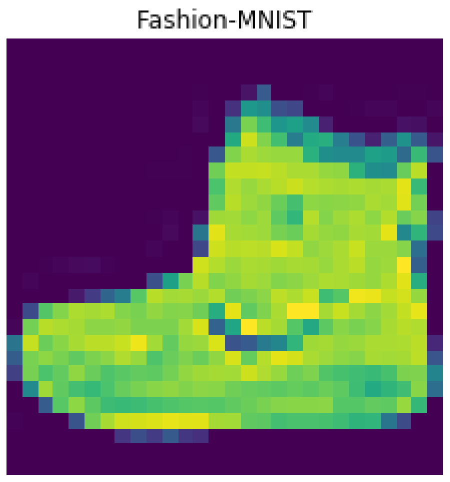
31.2.2 Load data

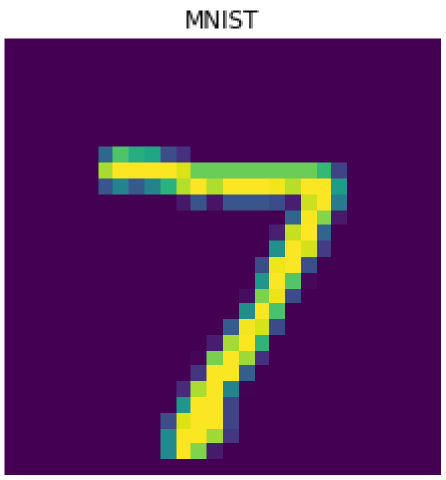
The in-distribution dataset is Fashion-MNIST and the out-of-distribution dataset we'd like to detect is MNIST.

[3]:
```
(X_train_in, y_train_in), (X_test_in, y_test_in) = load_data('fashion_mnist')
X_test_ood, y_test_ood = load_data('mnist')[1]
input_shape = X_train_in.shape[1:]
print(X_train_in.shape, X_test_in.shape, X_test_ood.shape)
```

```
(60000, 28, 28, 1) (10000, 28, 28, 1) (10000, 28, 28, 1)
```

[4]:
```
i = 0
plt.imshow(X_train_in[i].reshape(input_shape[:-1]))
plt.title('Fashion-MNIST')
plt.axis('off')
plt.show();
plt.imshow(X_test_ood[i].reshape(input_shape[:-1]))
plt.title('MNIST')
plt.axis('off')
plt.show();
```





31.2.3 Define PixelCNN++ model

We now need to define our generative model. This is not necessary if the pretrained detector is later loaded from the Google Bucket.

Key PixelCNN++ arguments in a nutshell:

- *num_resnet*: number of layers (Fig.2 PixelCNN) within each hierarchical block (Fig.2 PixelCNN++).
- *num_hierarchies*: number of blocks separated by expansions or contractions of dimensions. See Fig.2 PixelCNN++.
- *num_filters*: number of convolutional filters.
- *num_logistic_mix*: number of components in the logistic mixture distribution.
- *receptive_field_dims*: height and width in pixels of the receptive field above and to the left of a given pixel.

Optionally, a different model can be passed to the detector with argument *model_background*. The Likelihood Ratio paper mentions that additional $L2$-regularization (*l2_weight*) for the background model could improve detection performance.

```
[5]: model = PixelCNN(
         image_shape=input_shape,
         num_resnet=5,
         num_hierarchies=2,
         num_filters=32,
         num_logistic_mix=1,
         receptive_field_dims=(3, 3),
         dropout_p=.3,
         l2_weight=0.
     )
```

31.2.4 Load or train the outlier detector

We can again either fetch the pretrained detector from a Google Cloud Bucket or train one from scratch:

```
[6]: load_pretrained = True
```

```
[7]: filepath = os.path.join(os.getcwd(), 'my_path')  # change to download directory
if load_pretrained:  # load pretrained outlier detector
    detector_type = 'outlier'
    dataset = 'fashion_mnist'
    detector_name = 'LLR'
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
else:
    # initialize detector
    od = LLR(threshold=None, model=model)

    # train
    od.fit(
        X_train_in,
        mutate_fn_kwargs=dict(rate=.2),
        mutate_batch_size=1000,
        optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
        epochs=20,
        batch_size=32,
        verbose=False
    )

    # save the trained outlier detector
    save_detector(od, filepath)
```

```
WARNING:alibi_detect.od.llr:No threshold level set. Need to infer threshold using␣
↪`infer_threshold`.
```

We can load our saved detector again by defining the PixelCNN architectures for the semantic and background models as well as providing the shape of the input data:

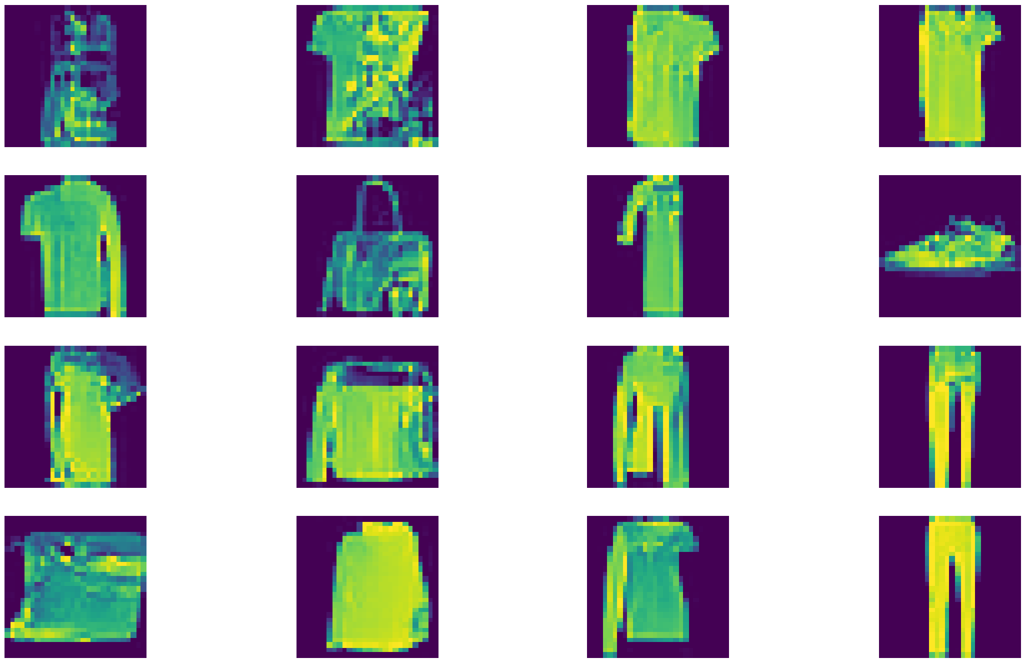
```
[8]: kwargs = {'dist_s': model, 'dist_b': model.copy(), 'input_shape': input_shape}
od = load_detector(filepath, **kwargs)
```

```
WARNING:alibi_detect.od.llr:No threshold level set. Need to infer threshold using␣
↪`infer_threshold`.
```

Let's sample some instances from the semantic model to check how good our generative model is:

```
[9]: n_sample = 16
X_sample = od.dist_s.sample(n_sample).numpy()
```

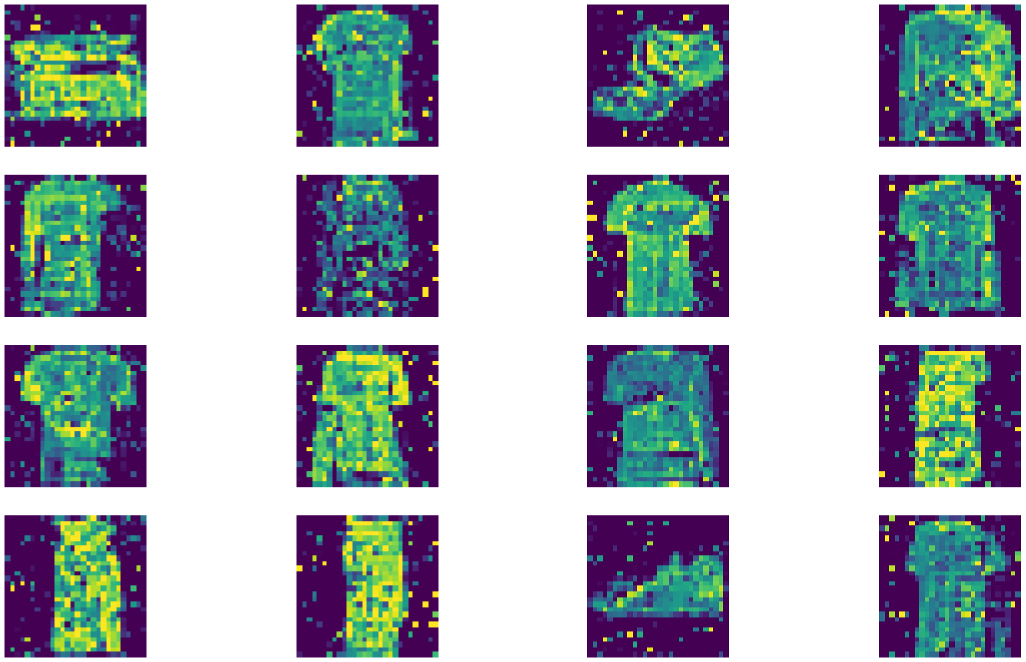
```
[10]: plot_grid_img(X_sample)
```



Most of the instances look like they represent the dataset well. When we do the same thing for our background model, we see that there is some background noise injected:

```
[11]: X_sample = od.dist_b.sample(n_sample).numpy()
```

```
[12]: plot_grid_img(X_sample)
```



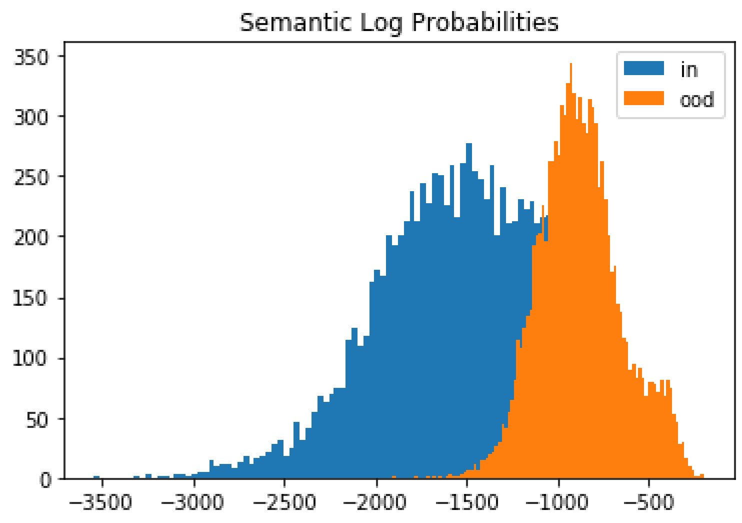
31.2.5 Compare the log likelihoods

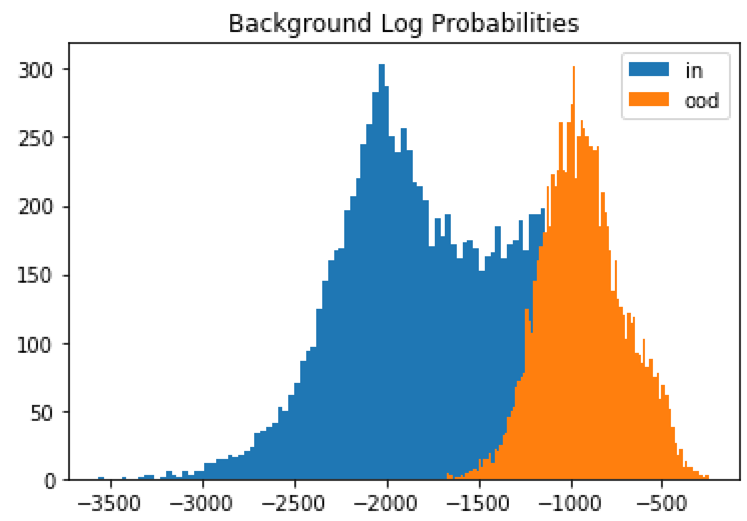
Let's compare the log likelihoods of the inliers vs. the outlier data under the semantic and background models. Although MNIST data looks very distinct from Fashion-MNIST, the generative model does not distinguish well between the 2 datasets as shown by the histograms of the log likelihoods:

```
[13]: shape_in, shape_ood = X_test_in.shape[0], X_test_ood.shape[0]
```

```
[14]: # semantic model
logp_s_in = predict_batch(od.dist_s.log_prob, X_test_in, batch_size=32, shape=shape_
↪in)
logp_s_ood = predict_batch(od.dist_s.log_prob, X_test_ood, batch_size=32, shape=shape_
↪ood)
logp_s = np.concatenate([logp_s_in, logp_s_ood])
# background model
logp_b_in = predict_batch(od.dist_b.log_prob, X_test_in, batch_size=32, shape=shape_
↪in)
logp_b_ood = predict_batch(od.dist_b.log_prob, X_test_ood, batch_size=32, shape=shape_
↪ood)
```

```
[15]: # show histograms
plt.hist(logp_s_in, bins=100, label='in');
plt.hist(logp_s_ood, bins=100, label='ood');
plt.title('Semantic Log Probabilities')
plt.legend()
plt.show()

plt.hist(logp_b_in, bins=100, label='in');
plt.hist(logp_b_ood, bins=100, label='ood');
plt.title('Background Log Probabilities')
plt.legend()
plt.show()
```

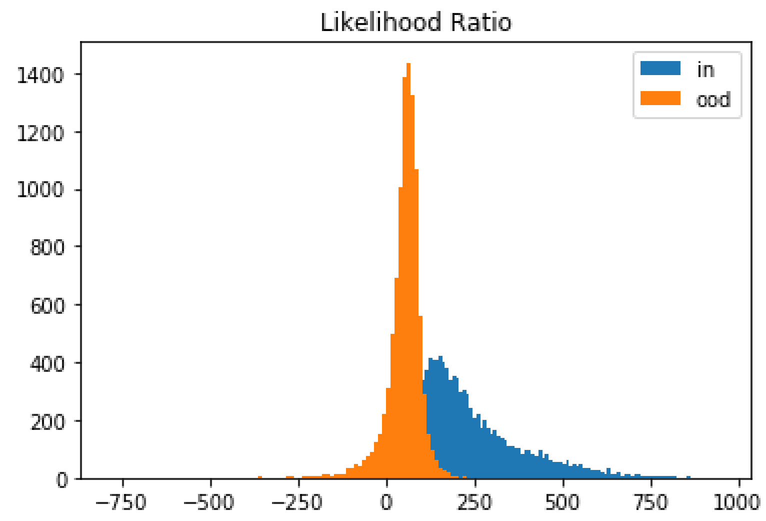




This is due to the dominance of the background which is similar (basically lots of 0's for both datasets). If we however take the likelihood ratio, the MNIST data are detected as outliers. And this is exactly what the outlier detector does as well:

[16]:
```
llr_in = logp_s_in - logp_b_in
llr_ood = logp_s_ood - logp_b_ood
```

[17]:
```
plt.hist(llr_in, bins=100, label='in');
plt.hist(llr_ood, bins=100, label='ood');
plt.title('Likelihood Ratio')
plt.legend()
plt.show()
```



31.2.6 Detect outliers

We follow the same procedure with the outlier detector. First we need to set an outlier threshold with `infer_threshold`. We need to pass a batch of instances and specify what percentage of those we consider to be normal via `threshold_perc`. Let's assume we have a small batch of data with roughly 50% outliers but we don't know exactly which ones.

```
[18]: n, frac_outlier = 500, .5
      perc_outlier = 100 * frac_outlier
      n_in, n_ood = int(n * (1 - frac_outlier)), int(n * frac_outlier)
      idx_in = np.random.choice(shape_in, size=n_in, replace=False)
      idx_ood = np.random.choice(shape_ood, size=n_ood, replace=False)
      X_threshold = np.concatenate([X_test_in[idx_in], X_test_ood[idx_ood]])
```

```
[19]: od.infer_threshold(X_threshold, threshold_perc=perc_outlier, batch_size=32)
      print('New threshold: {}'.format(od.threshold))
```

```
New threshold: -96.67073059082031
```

Let's save the outlier detector with updated threshold:

```
[20]: save_detector(od, filepath)
```

Let's now predict outliers on the combined Fashion-MNIST and MNIST datasets:

```
[21]: X_test = np.concatenate([X_test_in, X_test_ood])
      y_test = np.concatenate([np.zeros(X_test_in.shape[0]), np.ones(X_test_ood.shape[0])])
      print(X_test.shape, y_test.shape)
```

```
(20000, 28, 28, 1) (20000,)
```

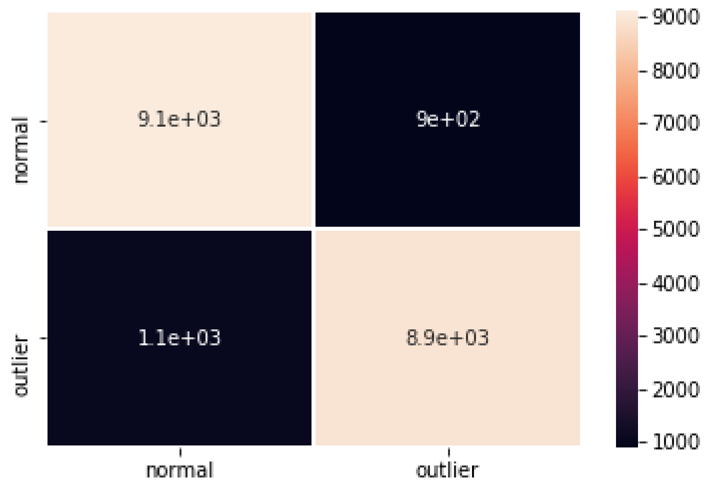
```
[22]: od_preds = od.predict(X_test,
                            batch_size=32,
                            outlier_type='instance',     # use 'feature' or 'instance' level
                            return_feature_score=True,   # scores used to determine outliers
                            return_instance_score=True)
```

31.2.7 Display results

F1 score, accuracy, precision, recall and confusion matrix:

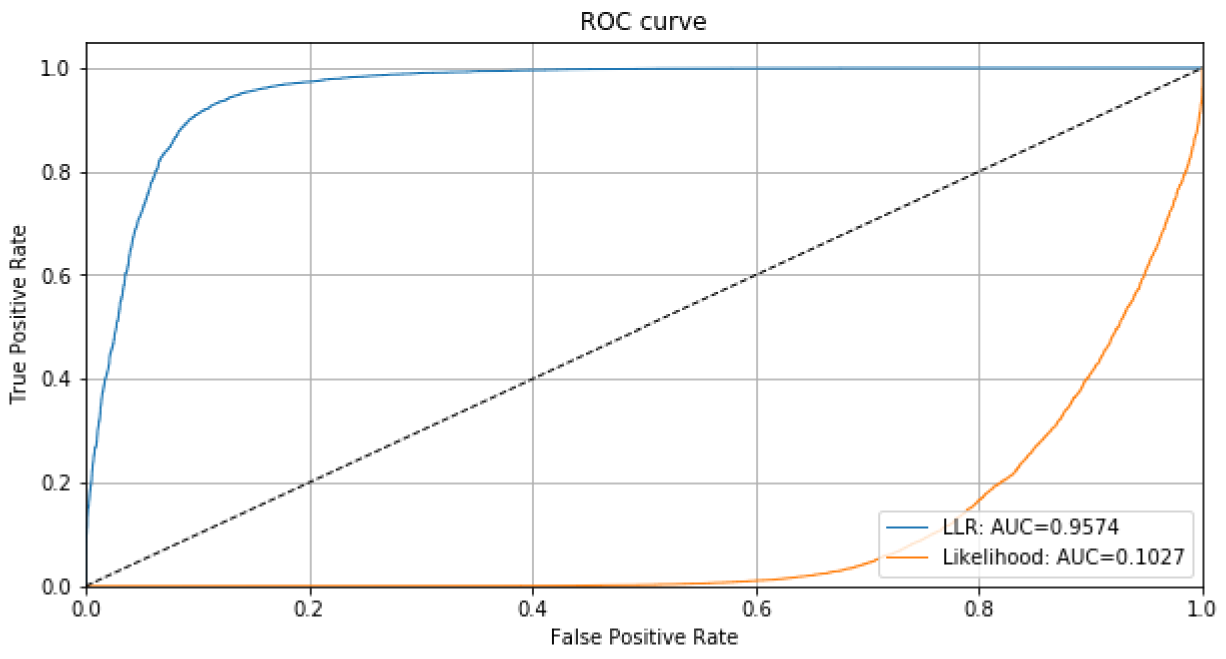
```
[23]: y_pred = od_preds['data']['is_outlier']
      labels = ['normal', 'outlier']
      f1 = f1_score(y_test, y_pred)
      acc = accuracy_score(y_test, y_pred)
      prec = precision_score(y_test, y_pred)
      rec = recall_score(y_test, y_pred)
      print('F1 score: {:.3f} -- Accuracy: {:.3f} -- Precision: {:.3f} '
            '-- Recall: {:.3f}'.format(f1, acc, prec, rec))
      cm = confusion_matrix(y_test, y_pred)
      df_cm = pd.DataFrame(cm, index=labels, columns=labels)
      sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
      plt.show()
```

```
F1 score: 0.901 -- Accuracy: 0.901 -- Precision: 0.909 -- Recall: 0.892
```



We can also plot the ROC curve based on the instance level outlier scores and compare it with the likelihood of only the semantic model:

[24]:
```
roc_data = {
    'LLR': {'scores': od_preds['data']['instance_score'], 'labels': y_test},
    'Likelihood': {'scores': -logp_s, 'labels': y_test}  # negative b/c outlier score
}
plot_roc(roc_data)
```



31.2.8 Analyse feature scores

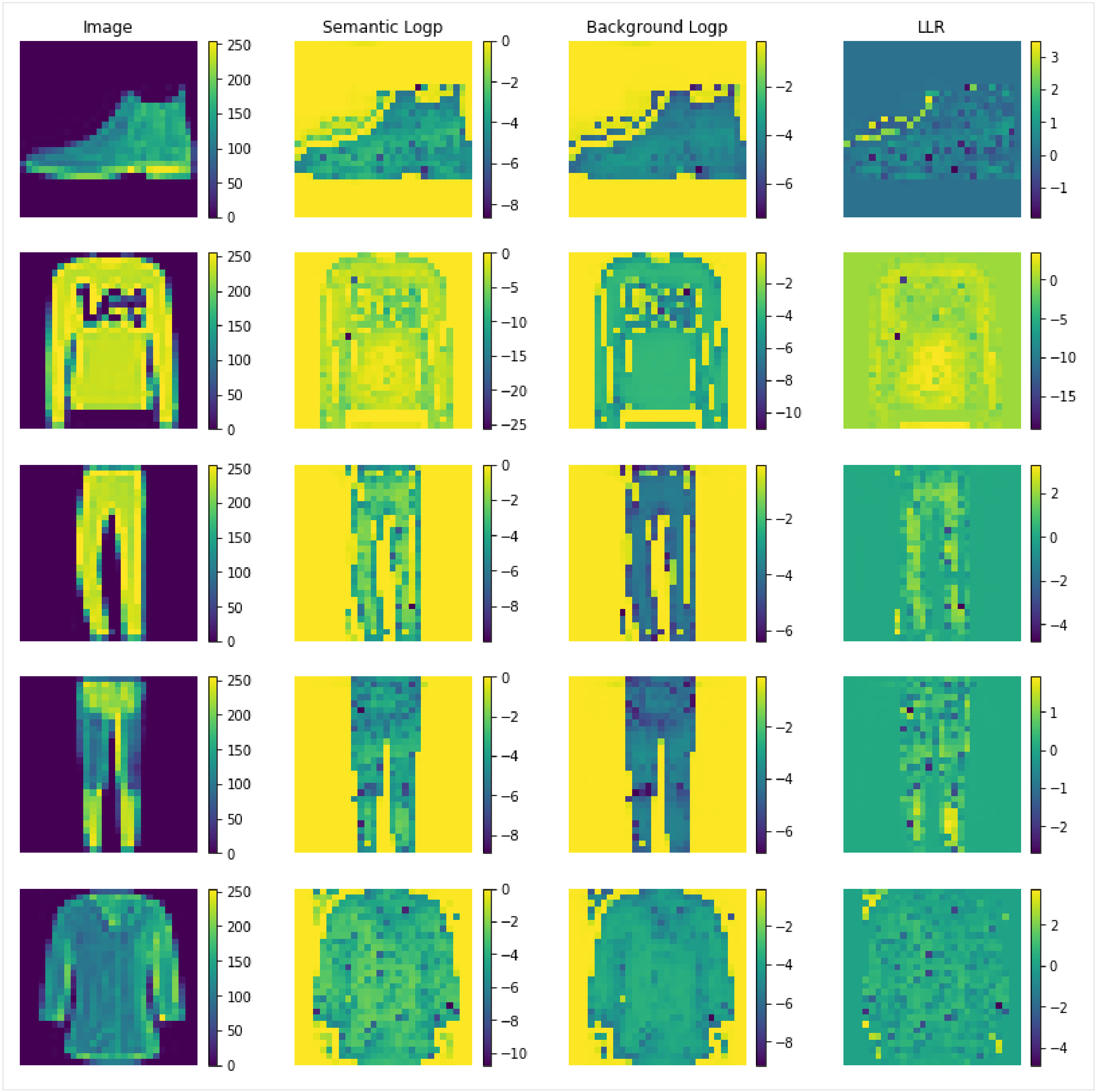
To understand why the likelihood ratio works to detect outliers but the raw log likelihoods don't, it is helpful to look at the pixel-wise log likelihoods of both the semantic and background models.

[25]:
```
n_plot = 5
```

[26]:
```
# semantic model
logp_fn_s = partial(od.dist_s.log_prob, return_per_feature=True)
logp_s_pixel_in = predict_batch(logp_fn_s, X_test_in[:n_plot], batch_size=32)
logp_s_pixel_ood = predict_batch(logp_fn_s, X_test_ood[:n_plot], batch_size=32)

# background model
logp_fn_b = partial(od.dist_b.log_prob, return_per_feature=True)
logp_b_pixel_in = predict_batch(logp_fn_b, X_test_in[:n_plot], batch_size=32)
logp_b_pixel_ood = predict_batch(logp_fn_b, X_test_ood[:n_plot], batch_size=32)

# pixel-wise likelihood ratios
llr_pixel_in = logp_s_pixel_in - logp_b_pixel_in
llr_pixel_ood = logp_s_pixel_ood - logp_b_pixel_ood
```

Plot in-distribution instances:

[27]:
```
idx = list(np.arange(n_plot))
plot_grid_logp(idx, X_test_in, logp_s_pixel_in, logp_b_pixel_in, figsize=(14,14))
```



It is clear that both the semantic and background model attach high probabilities to the background pixels. This effect is cancelled out in the likelihood ratio in the last column. The same applies to the out-of-distribution instances:

```
[28]: idx = list(np.arange(n_plot))
      plot_grid_logp(idx, X_test_ood, logp_s_pixel_ood, logp_b_pixel_ood, figsize=(14,14))
```

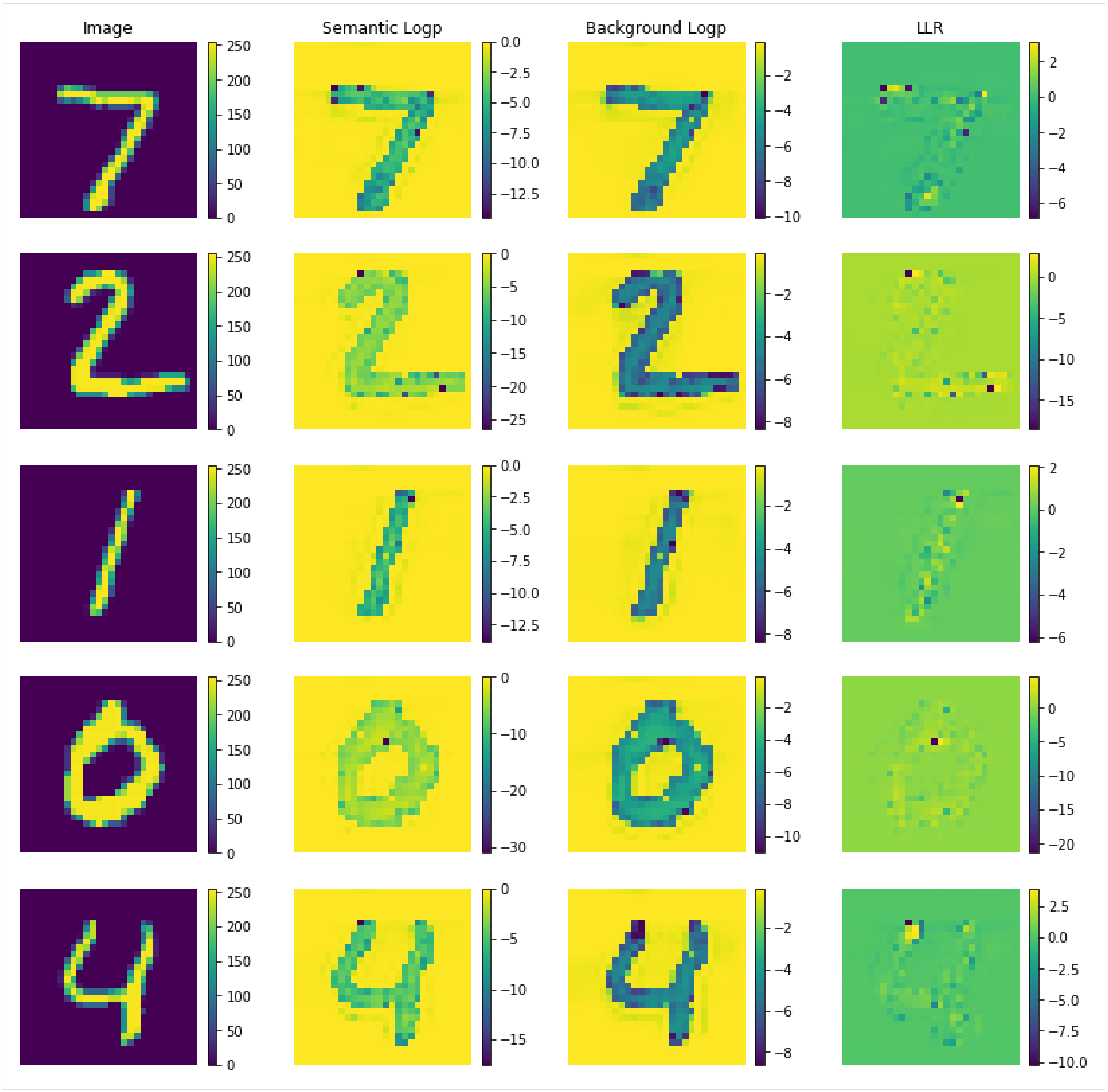


Image
Semantic Logp
Background Logp
LLR

CHAPTER

THIRTYTWO

LIKELIHOOD RATIO OUTLIER DETECTION ON GENOMIC SEQUENCES

32.1 Method

The outlier detector described by Ren et al. (2019) in Likelihood Ratios for Out-of-Distribution Detection uses the likelihood ratio between 2 generative models as the outlier score. One model is trained on the original data while the other is trained on a perturbed version of the dataset. This is based on the observation that the likelihood score for an instance under a generative model can be heavily affected by population level background statistics. The second generative model is therefore trained to capture the background statistics still present in the perturbed data while the semantic features have been erased by the perturbations.

The perturbations are added using an independent and identical Bernoulli distribution with rate μ which substitutes a feature with one of the other possible feature values with equal probability. Each feature in the genome dataset can take 4 values (one of the ACGT nucleobases). This means that a perturbed feature is swapped with one of the other nucleobases. The generative model used in the example is a simple LSTM network.

32.2 Dataset

The bacteria genomics dataset for out-of-distribution detection was released as part of the Likelihood Ratios for Out-of-Distribution Detection paper. From the original *TL;DR*: *The dataset contains genomic sequences of 250 base pairs from 10 in-distribution bacteria classes for training, 60 OOD bacteria classes for validation, and another 60 different OOD bacteria classes for test*. There are respectively 1, 7 and again 7 million sequences in the training, validation and test sets. For detailed info on the dataset check the README.

[1]:
```
import os
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
import tensorflow as tf
from tensorflow.keras.layers import Dense, Input, LSTM

from alibi_detect.od import LLR
from alibi_detect.datasets import fetch_genome
from alibi_detect.utils.fetching import fetch_detector
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.utils.prediction import predict_batch
from alibi_detect.utils.visualize import plot_roc
```

32.2.1 Load genome data

X represents the genome sequences and *y* whether they are outliers (1) or not (0).

[2]:
```
(X_train, y_train), (X_val, y_val), (X_test, y_test) = \
        fetch_genome(return_X_y=True, return_labels=False)
print(X_train.shape, y_train.shape, X_val.shape, y_val.shape, X_test.shape, y_test.
↪shape)
```

```
(1000000, 250) (1000000,) (6999774, 250) (6999774,) (7000000, 250) (7000000,)
```

There are no outliers in the training set and a majority of outliers (compared to the training data) in the validation and test sets:

[3]:
```
print('Fraction of outliers in train, val and test sets: '
      '{:.2f}, {:.2f} and {:.2f}'.format(y_train.mean(), y_val.mean(), y_test.mean()))
```

```
Fraction of outliers in train, val and test sets: 0.00, 0.86 and 0.86
```

32.2.2 Define model

We need to define a generative model which models the genome sequences. We follow the paper and opt for a simple LSTM. Note that we don't actually need to define the model below if we simply load the pretrained detector later on:

[4]:
```
genome_dim = 249  # not 250 b/c we use 1->249 as input and 2->250 as target
input_dim = 4  # ACGT nucleobases
hidden_dim = 2000

inputs = Input(shape=(genome_dim,), dtype=tf.int8)
x = tf.one_hot(tf.cast(inputs, tf.int32), input_dim)
x = LSTM(hidden_dim, return_sequences=True)(x)
logits = Dense(input_dim, activation=None)(x)
model = tf.keras.Model(inputs=inputs, outputs=logits, name='LlrLSTM')
```

We also need to define our loss function which we can utilize to evaluate the log-likelihood for the outlier detector:

[5]:
```
def loss_fn(y, x):
    y = tf.one_hot(tf.cast(y, tf.int32), 4)  # ACGT on-hot encoding
    return tf.nn.softmax_cross_entropy_with_logits(y, x, axis=-1)
```

[6]:
```
def likelihood_fn(y, x):
    return -loss_fn(y, x)
```

32.2.3 Load or train the outlier detector

We can again either fetch the pretrained detector from a Google Cloud Bucket or train one from scratch:

[7]:
```
load_pretrained = True
```

[8]:
```
filepath = os.path.join(os.getcwd(), 'my_path')  # change to download directory
if load_pretrained:  # load pretrained outlier detector
    detector_type = 'outlier'
    dataset = 'genome'
    detector_name = 'LLR'
```

(continues on next page)

(continued from previous page)

```
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
else:
    # initialize detector
    od = LLR(threshold=None, model=model, log_prob=likelihood_fn, sequential=True)

    # train
    od.fit(
        X_train,
        mutate_fn_kwargs=dict(rate=.2, feature_range=(0,3)),
        mutate_batch_size=1000,
        loss_fn=loss_fn,
        optimizer=tf.keras.optimizers.Adam(learning_rate=5e-4),
        epochs=20,
        batch_size=100,
        verbose=False
    )

    # save the trained outlier detector
    save_detector(od, filepath)
```

```
Downloading data from https://storage.googleapis.com/seldon-models/alibi-detect/od/
↪LLR/genome/model/model_s.h5
```

```
WARNING:alibi_detect.od.llr:No threshold level set. Need to infer threshold using␣
↪`infer_threshold`.
```

32.2.4 Compare the log likelihoods

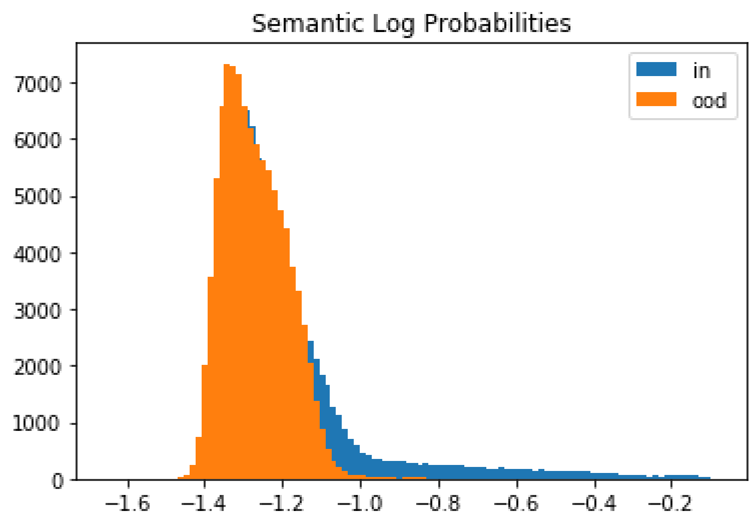
Let's compare the log likelihoods of the inliers vs. the outlier test set data under the semantic and background models. We randomly sample $100,000$ instances from both distributions since the full test set contains $7,000,000$ genomic sequences. The histograms show that the generative model does not distinguish well between inliers and outliers.

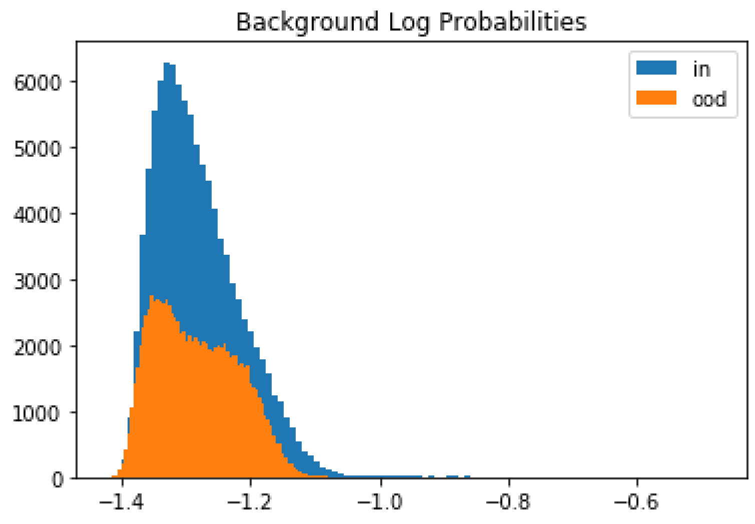
```
[9]: idx_in, idx_ood = np.where(y_test == 0)[0], np.where(y_test == 1)[0]
n_in, n_ood = idx_in.shape[0], idx_ood.shape[0]
n_sample = 100000  # sample 100k inliers and outliers each
sample_in = np.random.choice(n_in, size=n_sample, replace=False)
sample_ood = np.random.choice(n_ood, size=n_sample, replace=False)
X_test_in, X_test_ood = X_test[idx_in[sample_in]], X_test[idx_ood[sample_ood]]
y_test_in, y_test_ood = y_test[idx_in[sample_in]], y_test[idx_ood[sample_ood]]
X_test_sample = np.concatenate([X_test_in, X_test_ood])
y_test_sample = np.concatenate([y_test_in, y_test_ood])
print(X_test_in.shape, X_test_ood.shape)
```

```
(100000, 250) (100000, 250)
```

```
[10]: # semantic model
logp_s_in = od.logp_alt(od.dist_s, X_test_in, batch_size=100)
logp_s_ood = od.logp_alt(od.dist_s, X_test_ood, batch_size=100)
logp_s = np.concatenate([logp_s_in, logp_s_ood])
# background model
logp_b_in = od.logp_alt(od.dist_b, X_test_in, batch_size=100)
logp_b_ood = od.logp_alt(od.dist_b, X_test_ood, batch_size=100)
logp_b = np.concatenate([logp_b_in, logp_b_ood])
```

[11]:
```
# show histograms
plt.hist(logp_s_in, bins=100, label='in');
plt.hist(logp_s_ood, bins=100, label='ood');
plt.title('Semantic Log Probabilities')
plt.legend()
plt.show()

plt.hist(logp_b_in, bins=100, label='in');
plt.hist(logp_b_ood, bins=100, label='ood');
plt.title('Background Log Probabilities')
plt.legend()
plt.show()
```

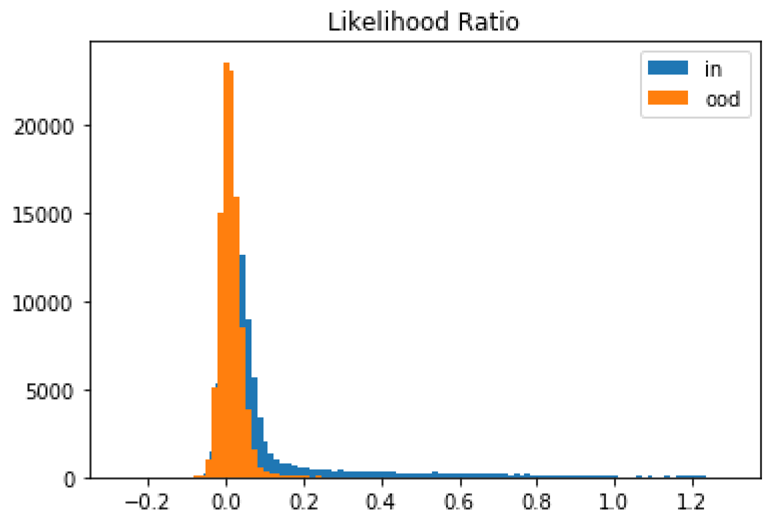




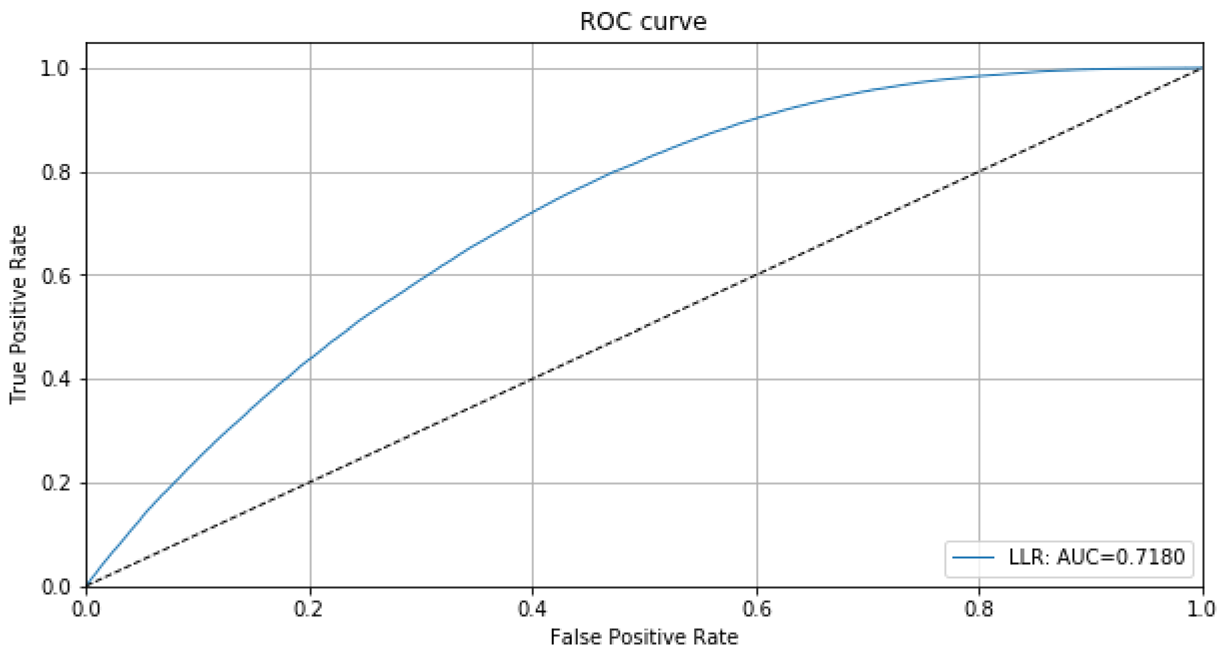
This is because of the background-effect which is in this case the GC-content in the genomic sequences. This effect is partially reduced when taking the likelihood ratio:

[12]:
```
llr_in = logp_s_in - logp_b_in
llr_ood = logp_s_ood - logp_b_ood
```

[13]:
```
plt.hist(llr_in, bins=100, label='in');
plt.hist(llr_ood, bins=100, label='ood');
plt.title('Likelihood Ratio')
plt.legend()
plt.show()
```



[14]:
```
llr = np.concatenate([llr_in, llr_ood])
roc_data = {'LLR': {'scores': -llr, 'labels': y_test_sample}}
plot_roc(roc_data)
```



32.2.5 Detect outliers

We follow the same procedure with the outlier detector. First we need to set an outlier threshold with `infer_threshold`. We need to pass a batch of instances and specify what percentage of those we consider to be normal via `threshold_perc`. Let's assume we have a small batch of data with roughly 30% outliers but we don't know exactly which ones.

[15]:
```
n, frac_outlier = 1000, .3
perc_outlier = 100 * frac_outlier
n_sample_in, n_sample_ood = int(n * (1 - frac_outlier)), int(n * frac_outlier)
idx_in, idx_ood = np.where(y_val == 0)[0], np.where(y_val == 1)[0]
n_in, n_ood = idx_in.shape[0], idx_ood.shape[0]
sample_in = np.random.choice(n_in, size=n_sample_in, replace=False)
sample_ood = np.random.choice(n_ood, size=n_sample_ood, replace=False)
X_thr_in, X_thr_ood = X_val[idx_in[sample_in]], X_val[idx_ood[sample_ood]]
X_threshold = np.concatenate([X_thr_in, X_thr_ood])
print(X_threshold.shape)
```

```
(1000, 250)
```

[16]:
```
od.infer_threshold(X_threshold, threshold_perc=perc_outlier, batch_size=100)
print('New threshold: {}'.format(od.threshold))
```

```
New threshold: -0.0506670355796814
```

Let's save the outlier detector with updated threshold:

[17]:
```
save_detector(od, filepath)
```

Let'spredict outliers on a sample of the test set:

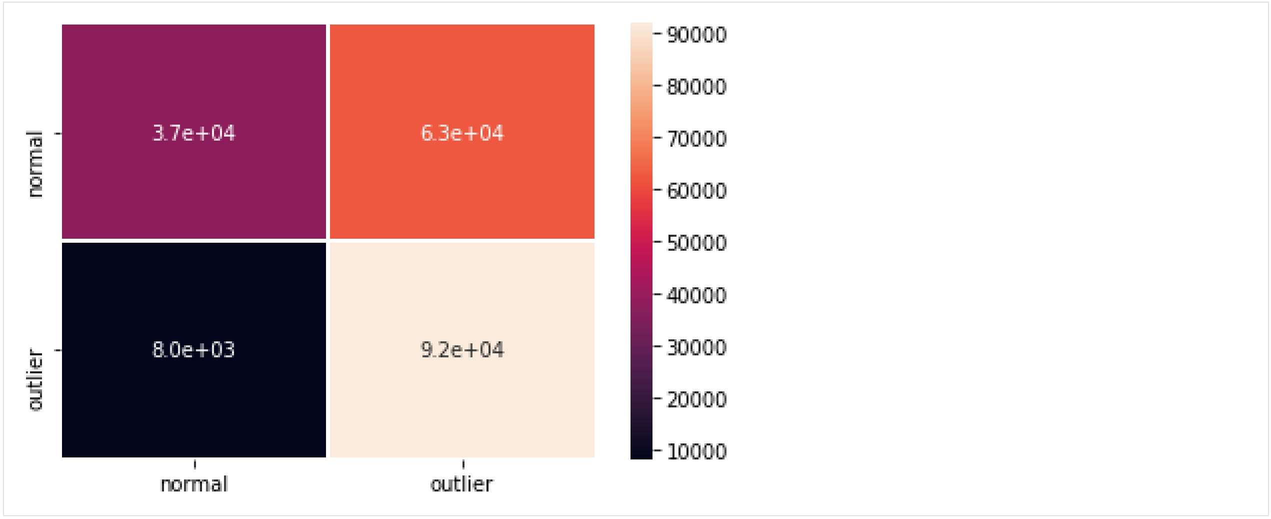
[18]:
```
od_preds = od.predict(X_test_sample, batch_size=100)
```

32.2.6 Display results

F1 score, accuracy, precision, recall and confusion matrix:

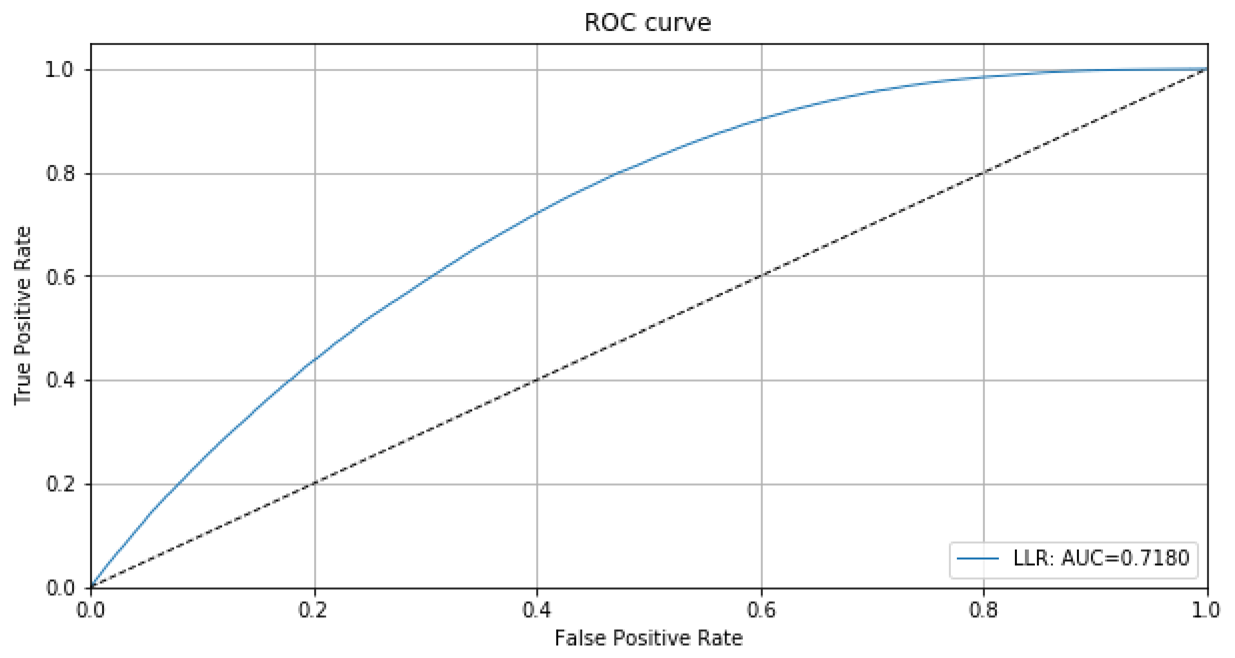
[19]:
```
y_pred = od_preds['data']['is_outlier']
labels = ['normal', 'outlier']
f1 = f1_score(y_test_sample, y_pred)
acc = accuracy_score(y_test_sample, y_pred)
prec = precision_score(y_test_sample, y_pred)
rec = recall_score(y_test_sample, y_pred)
print('F1 score: {:.3f} -- Accuracy: {:.3f} -- Precision: {:.3f} '
      '-- Recall: {:.3f}'.format(f1, acc, prec, rec))
cm = confusion_matrix(y_test_sample, y_pred)
df_cm = pd.DataFrame(cm, index=labels, columns=labels)
sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
plt.show()
```

```
F1 score: 0.722 -- Accuracy: 0.646 -- Precision: 0.595 -- Recall: 0.919
```



We can also plot the ROC curve based on the instance level outlier scores:

```
[20]: roc_data = {'LLR': {'scores': od_preds['data']['instance_score'], 'labels': y_test_
      ↪sample}}
      plot_roc(roc_data)
```



CHAPTER

THIRTYTHREE

TIME-SERIES OUTLIER DETECTION USING PROPHET ON WEATHER DATA

33.1 Method

The Prophet outlier detector uses the Prophet time series forecasting package explained in this excellent paper. The underlying Prophet model is a decomposable univariate time series model combining trend, seasonality and holiday effects. The model forecast also includes an uncertainty interval around the estimated trend component using the MAP estimate of the extrapolated model. Alternatively, full Bayesian inference can be done at the expense of increased compute. The upper and lower values of the uncertainty interval can then be used as outlier thresholds for each point in time. First, the distance from the observed value to the nearest uncertainty boundary (upper or lower) is computed. If the observation is within the boundaries, the outlier score equals the negative distance. As a result, the outlier score is the lowest when the observation equals the model prediction. If the observation is outside of the boundaries, the score equals the distance measure and the observation is flagged as an outlier. One of the main drawbacks of the method however is that you need to refit the model as new data comes in. This is undesirable for applications with high throughput and real-time detection.

To use this detector, first install Prophet by running `pip install alibi-detect[prophet]`.

33.2 Dataset

The example uses a weather time series dataset recorded by the Max-Planck-Institute for Biogeochemistry. The dataset contains 14 different features such as air temperature, atmospheric pressure, and humidity. These were collected every 10 minutes, beginning in 2003. Like the TensorFlow time-series tutorial, we only use data collected between 2009 and 2016.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import tensorflow as tf

from alibi_detect.od import OutlierProphet
from alibi_detect.utils.fetching import fetch_detector
from alibi_detect.utils.saving import save_detector, load_detector
```

33.3 Load dataset

[2]:
```
zip_path = tf.keras.utils.get_file(
    origin='https://storage.googleapis.com/tensorflow/tf-keras-datasets/jena_climate_
↪2009_2016.csv.zip',
    fname='jena_climate_2009_2016.csv.zip',
    extract=True
)
csv_path, _ = os.path.splitext(zip_path)
df = pd.read_csv(csv_path)
df['Date Time'] = pd.to_datetime(df['Date Time'], format='%d.%m.%Y %H:%M:%S')
print(df.shape)
df.head()
```

```
(420551, 15)
```

[2]:
```
            Date Time  p (mbar)  T (degC)  Tpot (K)  Tdew (degC)  rh (%)  \
0 2009-01-01 00:10:00    996.52     -8.02    265.40        -8.90    93.3
1 2009-01-01 00:20:00    996.57     -8.41    265.01        -9.28    93.4
2 2009-01-01 00:30:00    996.53     -8.51    264.91        -9.31    93.9
3 2009-01-01 00:40:00    996.51     -8.31    265.12        -9.07    94.2
4 2009-01-01 00:50:00    996.51     -8.27    265.15        -9.04    94.1

   VPmax (mbar)  VPact (mbar)  VPdef (mbar)  sh (g/kg)  H2OC (mmol/mol)  \
0          3.33          3.11          0.22       1.94             3.12
1          3.23          3.02          0.21       1.89             3.03
2          3.21          3.01          0.20       1.88             3.02
3          3.26          3.07          0.19       1.92             3.08
4          3.27          3.08          0.19       1.92             3.09

   rho (g/m**3)  wv (m/s)  max. wv (m/s)  wd (deg)
0       1307.75      1.03           1.75     152.3
1       1309.80      0.72           1.50     136.1
2       1310.24      0.19           0.63     171.6
3       1309.19      0.34           0.50     198.0
4       1309.00      0.32           0.63     214.3
```

Select subset to test Prophet model on:

[3]:
```
n_prophet = 10000
```

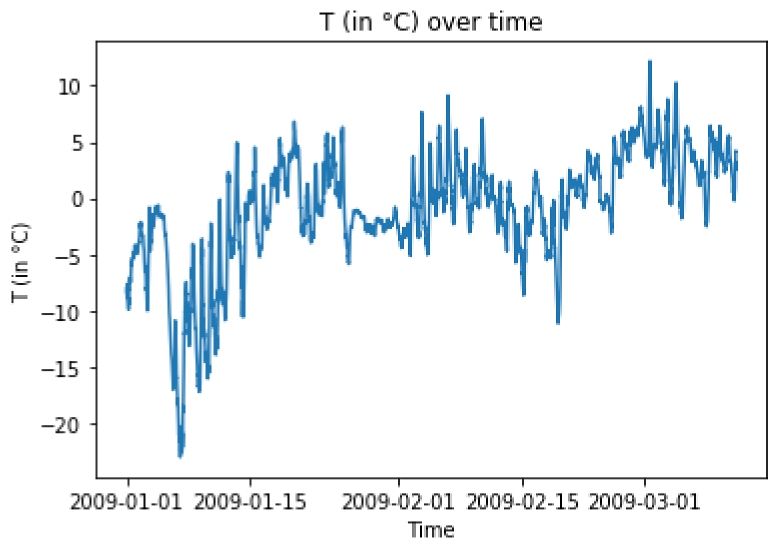
Prophet model expects a DataFrame with 2 columns: one named `ds` with the timestamps and one named `y` with the time series to be evaluated. We will just look at the temperature data:

[4]:
```
d = {'ds': df['Date Time'][:n_prophet], 'y': df['T (degC)'][:n_prophet]}
df_T = pd.DataFrame(data=d)
print(df_T.shape)
df_T.head()
```

```
(10000, 2)
```

[4]:
```
                   ds     y
0 2009-01-01 00:10:00 -8.02
1 2009-01-01 00:20:00 -8.41
2 2009-01-01 00:30:00 -8.51
3 2009-01-01 00:40:00 -8.31
4 2009-01-01 00:50:00 -8.27
```

[5]:
```
plt.plot(df_T['ds'], df_T['y'])
plt.title('T (in °C) over time')
plt.xlabel('Time')
plt.ylabel('T (in °C)')
plt.show()
```



33.4 Load or define outlier detector

The pretrained outlier and adversarial detectors used in the example notebooks can be found here. You can use the built-in `fetch_detector` function which saves the pre-trained models in a local directory `filepath` and loads the detector. Alternatively, you can train a detector from scratch:

[6]:
```
load_outlier_detector = False
```

[7]:
```
filepath = 'my_path'  # change to directory where model is downloaded
if load_outlier_detector:  # load pretrained outlier detector
    detector_type = 'outlier'
    dataset = 'weather'
    detector_name = 'OutlierProphet'
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
else:  # initialize, fit and save outlier detector
    od = OutlierProphet(threshold=.9)
    od.fit(df_T)
    save_detector(od, filepath)
```

```
INFO:fbprophet:Disabling yearly seasonality. Run prophet with yearly_seasonality=True␣
↪to override this.
```

Please check out the documentation as well as the original Prophet documentation on how to customize the Prophet-based outlier detector and add seasonalities, holidays, opt for a saturating logistic growth model or apply parameter regularization.

33.5 Predict outliers on test data

Define the test data. It is important that the timestamps of the test data follow the training data. We check this below by comparing the first few rows of the test DataFrame with the last few of the training DataFrame:

```
[8]: n_periods = 1000
d = {'ds': df['Date Time'][n_prophet:n_prophet+n_periods],
     'y': df['T (degC)'][n_prophet:n_prophet+n_periods]}
df_T_test = pd.DataFrame(data=d)
df_T_test.head()
```

```
[8]:
                        ds     y
10000 2009-03-11 10:50:00  4.12
10001 2009-03-11 11:00:00  4.62
10002 2009-03-11 11:10:00  4.29
10003 2009-03-11 11:20:00  3.95
10004 2009-03-11 11:30:00  3.96
```

```
[9]: df_T.tail()
```

```
[9]:
                       ds     y
9995 2009-03-11 10:00:00  2.69
9996 2009-03-11 10:10:00  2.98
9997 2009-03-11 10:20:00  3.66
9998 2009-03-11 10:30:00  4.21
9999 2009-03-11 10:40:00  4.19
```

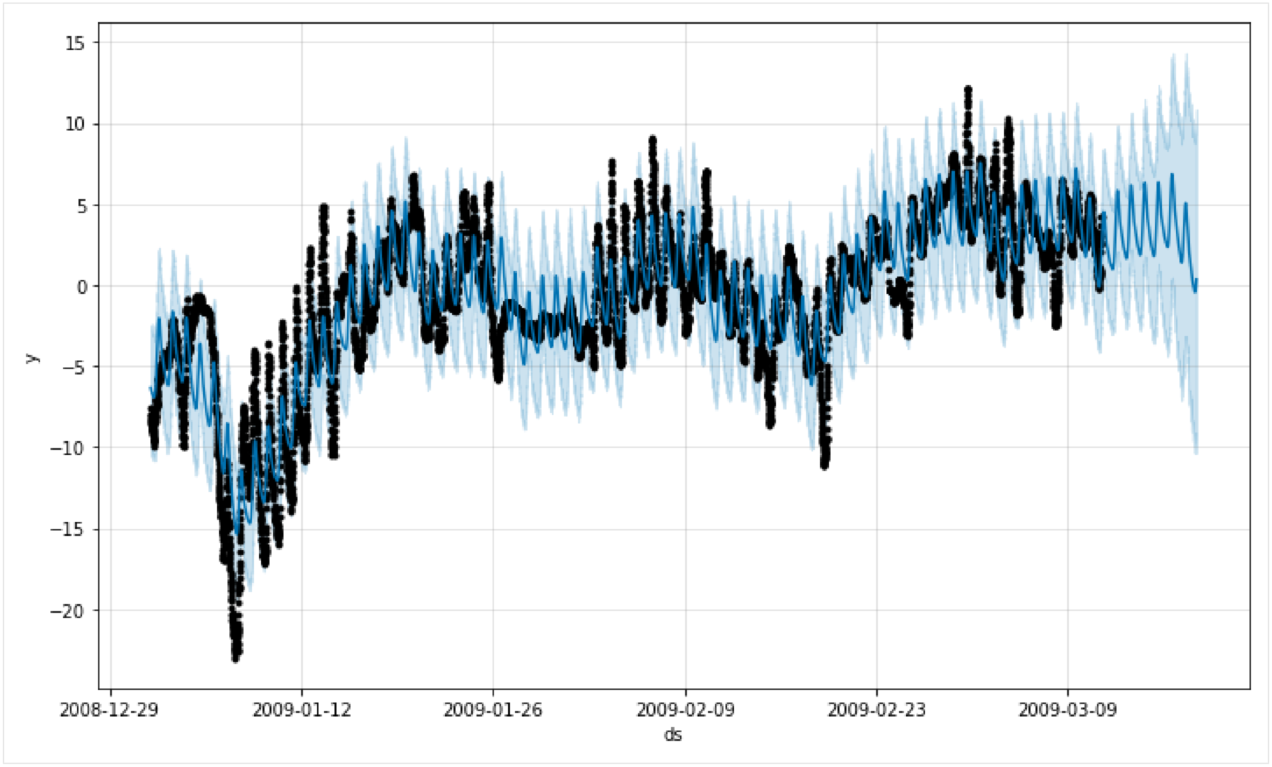
Predict outliers on test data:

```
[10]: od_preds = od.predict(
    df_T_test,
    return_instance_score=True,
    return_forecast=True
)
```

33.6 Visualize results

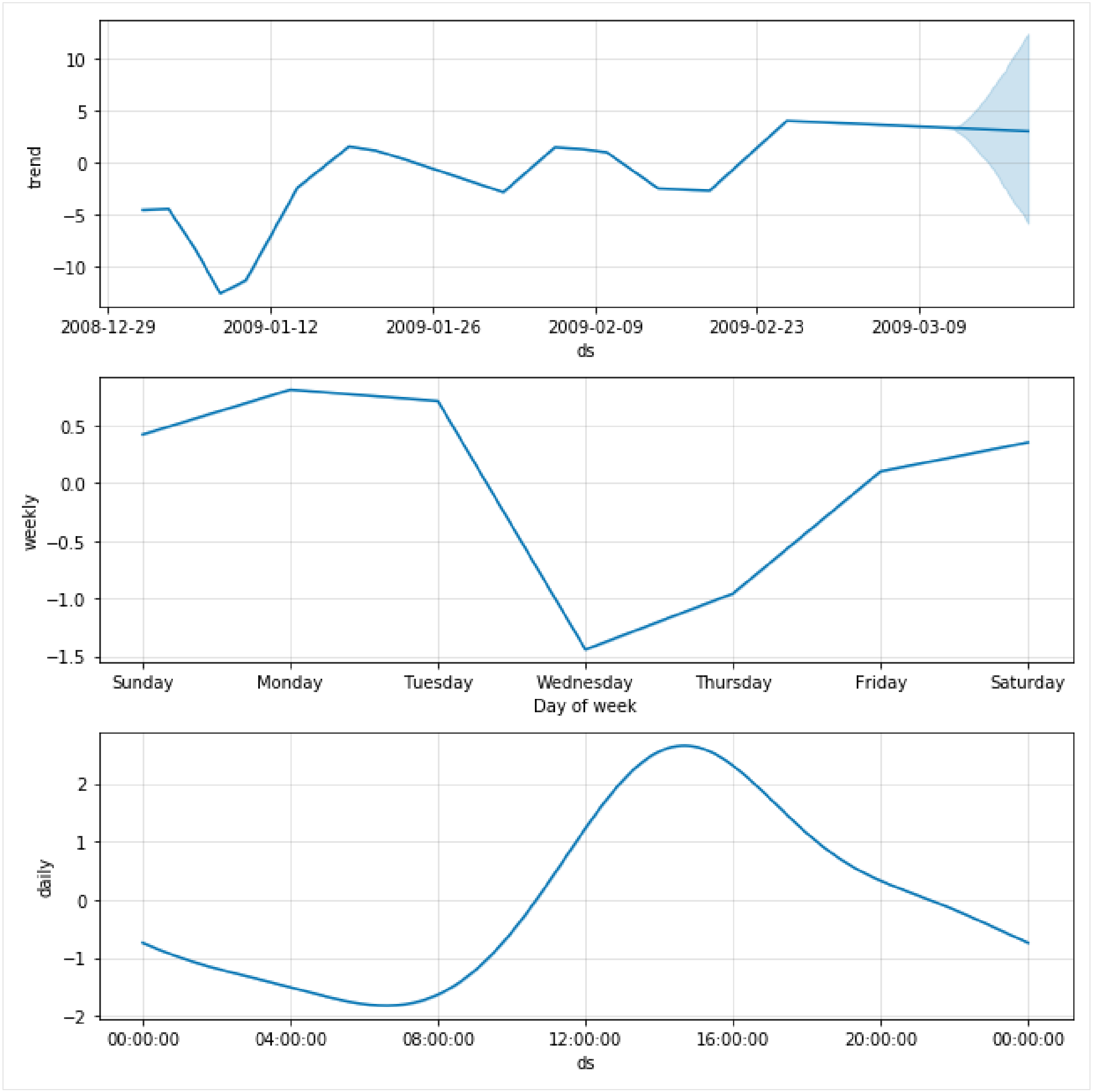
We can first visualize our predictions with Prophet's built in plotting functionality. This also allows us to include historical predictions:

```
[11]: future = od.model.make_future_dataframe(periods=n_periods, freq='10T', include_
↪history=True)
forecast = od.model.predict(future)
fig = od.model.plot(forecast)
```



We can also plot the breakdown of the different components in the forecast. Since we did not do full Bayesian inference with `mcmc_samples`, the uncertaintly intervals of the forecast are determined by the MAP estimate of the extrapolated trend.

[12]:
```
fig = od.model.plot_components(forecast)
```

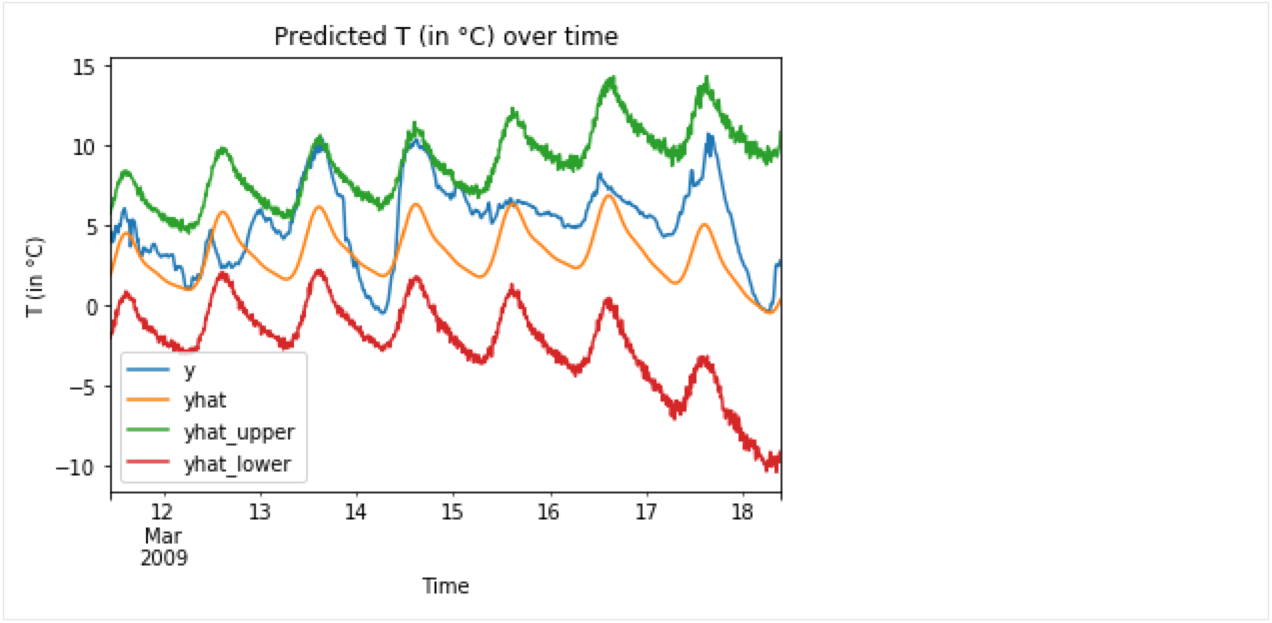


It is clear that the further we predict in the future, the wider the uncertainty intervals which determine the outlier threshold.

Let's overlay the actual data with the upper and lower outlier thresholds predictions and check where we predicted outliers:

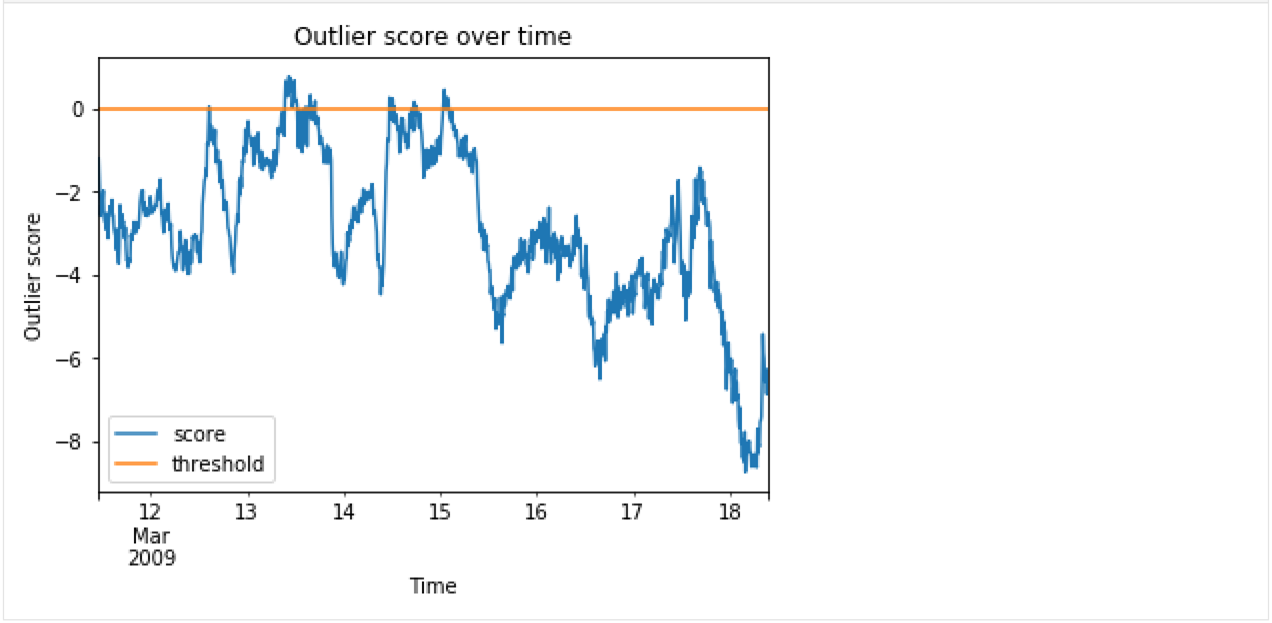
```
[13]: forecast['y'] = df['T (degC)'][:n_prophet+n_periods]
```

```
[14]: pd.plotting.register_matplotlib_converters()  # needed to plot timestamps
      forecast[-n_periods:].plot(x='ds', y=['y', 'yhat', 'yhat_upper', 'yhat_lower'])
      plt.title('Predicted T (in °C) over time')
      plt.xlabel('Time')
      plt.ylabel('T (in °C)')
      plt.show()
```



Outlier scores and predictions:

```
[15]: od_preds['data']['forecast']['threshold'] = np.zeros(n_periods)
od_preds['data']['forecast'][-n_periods:].plot(x='ds', y=['score', 'threshold'])
plt.title('Outlier score over time')
plt.xlabel('Time')
plt.ylabel('Outlier score')
plt.show()
```



The outlier scores naturally trend down as uncertainty increases when we predict further in the future.

Let's look at some individual outliers:

```
[16]: df_fcst = od_preds['data']['forecast']
df_outlier = df_fcst.loc[df_fcst['score'] > 0]
```

[17]:
```
print('Number of outliers: {}'.format(df_outlier.shape[0]))
df_outlier[['ds', 'yhat', 'yhat_lower', 'yhat_upper', 'y']]
```

```
Number of outliers: 42
```

[17]:
```
                     ds      yhat  yhat_lower  yhat_upper      y
166 2009-03-12 14:30:00  5.826341    2.359631    9.523064   2.32
279 2009-03-13 09:20:00  2.438507   -1.834116    6.408030   6.60
280 2009-03-13 09:30:00  2.546610   -1.484339    6.560092   7.22
281 2009-03-13 09:40:00  2.661307   -1.238431    6.712489   7.11
282 2009-03-13 09:50:00  2.782348   -1.284963    6.631783   7.22
283 2009-03-13 10:00:00  2.909426   -1.360733    7.213170   7.50
284 2009-03-13 10:10:00  3.042175   -0.938601    7.138561   7.71
285 2009-03-13 10:20:00  3.180168   -0.542519    7.159230   7.93
286 2009-03-13 10:30:00  3.322914   -0.749686    7.319972   7.98
287 2009-03-13 10:40:00  3.469862   -0.639899    7.248605   7.97
288 2009-03-13 10:50:00  3.620397   -0.536662    7.690310   8.11
289 2009-03-13 11:00:00  3.773845   -0.407588    7.735500   8.31
290 2009-03-13 11:10:00  3.929473   -0.242533    8.195561   8.22
291 2009-03-13 11:20:00  4.086493    0.070751    8.352415   8.47
292 2009-03-13 11:30:00  4.244068   -0.046052    8.216369   8.65
293 2009-03-13 11:40:00  4.401316    0.398563    8.058718   8.73
294 2009-03-13 11:50:00  4.557318    0.532095    8.671582   8.86
295 2009-03-13 12:00:00  4.711127    0.657084    8.844659   9.03
296 2009-03-13 12:10:00  4.861773    0.861597    8.995459   9.20
297 2009-03-13 12:20:00  5.008280    0.987469    9.190047   9.27
310 2009-03-13 14:30:00  6.146584    2.098111   10.394298  10.43
314 2009-03-13 15:10:00  6.108253    1.736977   10.220202  10.24
315 2009-03-13 15:20:00  6.070174    2.139051   10.002806  10.02
316 2009-03-13 15:30:00  6.021822    1.536603   10.264883  10.40
317 2009-03-13 15:40:00  5.963937    1.977076    9.995948  10.32
320 2009-03-13 16:10:00  5.741674    1.486856   10.023755  10.07
324 2009-03-13 16:50:00  5.368270    1.421167    9.341373   9.52
435 2009-03-14 11:20:00  4.284622   -0.291427    8.510237   8.77
437 2009-03-14 11:40:00  4.597406   -0.229902    9.058397   9.11
439 2009-03-14 12:00:00  4.904994    0.715068    9.270239   9.51
440 2009-03-14 12:10:00  5.054459    0.779145    9.515633   9.71
442 2009-03-14 12:30:00  5.339857    0.972692    9.709758   9.76
469 2009-03-14 17:00:00  5.406697    0.555661    9.653830   9.66
472 2009-03-14 17:30:00  5.096989    0.472095    9.414530   9.57
473 2009-03-14 17:40:00  4.996114    0.478616    9.479434   9.49
474 2009-03-14 17:50:00  4.897644    0.509235    9.347031   9.42
516 2009-03-15 00:50:00  2.658244   -2.097107    7.294615   7.56
517 2009-03-15 01:00:00  2.619797   -2.156021    7.082319   7.53
519 2009-03-15 01:20:00  2.547890   -1.977093    7.401913   7.42
521 2009-03-15 01:40:00  2.482139   -2.358331    7.093606   7.36
522 2009-03-15 01:50:00  2.451274   -2.391906    7.132587   7.35
523 2009-03-15 02:00:00  2.421542   -2.561413    7.114390   7.28
```

CHAPTER

THIRTYFOUR

TIME SERIES OUTLIER DETECTION WITH SPECTRAL RESIDUALS ON SYNTHETIC DATA

34.1 Method

The Spectral Residual outlier detector is based on the paper Time-Series Anomaly Detection Service at Microsoft and is suitable for unsupervised online anomaly detection in univariate time series data. The algorithm first computes the Fourier Transform of the original data. Then it computes the *spectral residual* of the log amplitude of the transformed signal before applying the Inverse Fourier Transform to map the sequence back from the frequency to the time domain. This sequence is called the *saliency map*. The anomaly score is then computed as the relative difference between the saliency map values and their moving averages. If this score is above a threshold, the value at a specific timestep is flagged as an outlier. For more details, please check out the paper.

34.2 Dataset

We test the outlier detector on a synthetic dataset generated with the TimeSynth package. It allows you to generate a wide range of time series (e.g. pseudo-periodic, autoregressive or Gaussian Process generated signals) and noise types (white or red noise). It can be installed as follows:

```
!pip install git+https://github.com/TimeSynth/TimeSynth.git
```

[1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.metrics import accuracy_score, confusion_matrix, f1_score, recall_score
import timesynth as ts

from alibi_detect.od import SpectralResidual
from alibi_detect.utils.perturbation import inject_outlier_ts
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.utils.visualize import plot_instance_score, plot_feature_outlier_ts
```

34.3 Create univariate time series

Define number of sampled points and the type of simulated time series. We use TimeSynth to generate a sinusoidal signal with Gaussian noise.

```
[2]: n_points = 100000
```

```
[3]: # timestamps
time_sampler = ts.TimeSampler(stop_time=n_points // 4)
time_samples = time_sampler.sample_regular_time(num_points=n_points)

# harmonic time series with Gaussian noise
sinusoid = ts.signals.Sinusoidal(frequency=0.25)
white_noise = ts.noise.GaussianNoise(std=0.1)
ts_harm = ts.TimeSeries(signal_generator=sinusoid, noise_generator=white_noise)
samples, signals, errors = ts_harm.sample(time_samples)
X = samples.reshape(-1, 1).astype(np.float32)
print(X.shape)
```

```
(100000, 1)
```

We can inject noise in the time series via `inject_outlier_ts`. The noise can be regulated via the percentage of outliers (`perc_outlier`), the strength of the perturbation (`n_std`) and the minimum size of the noise perturbation (`min_std`):

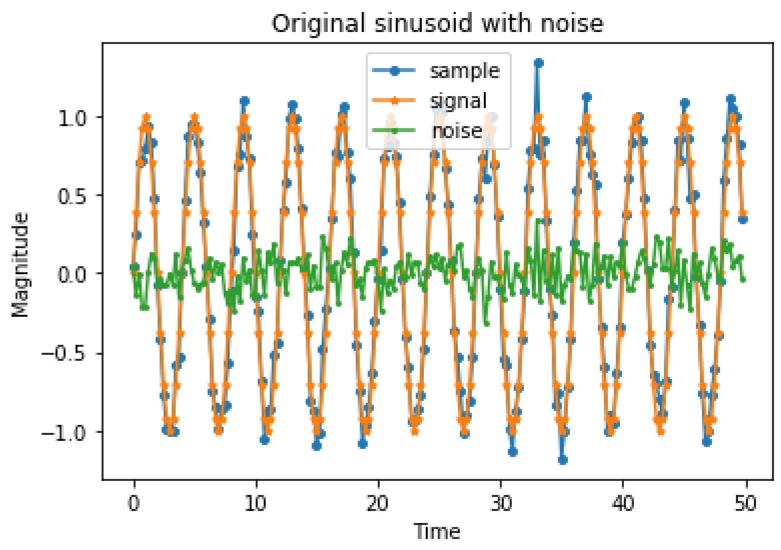
```
[4]: data = inject_outlier_ts(X, perc_outlier=10, perc_window=10, n_std=2., min_std=1.)
X_outlier, y_outlier, labels = data.data, data.target.astype(int), data.target_names
print(X_outlier.shape, y_outlier.shape)
```

```
(100000, 1) (100000,)
```

Visualize part of the original and perturbed time series:

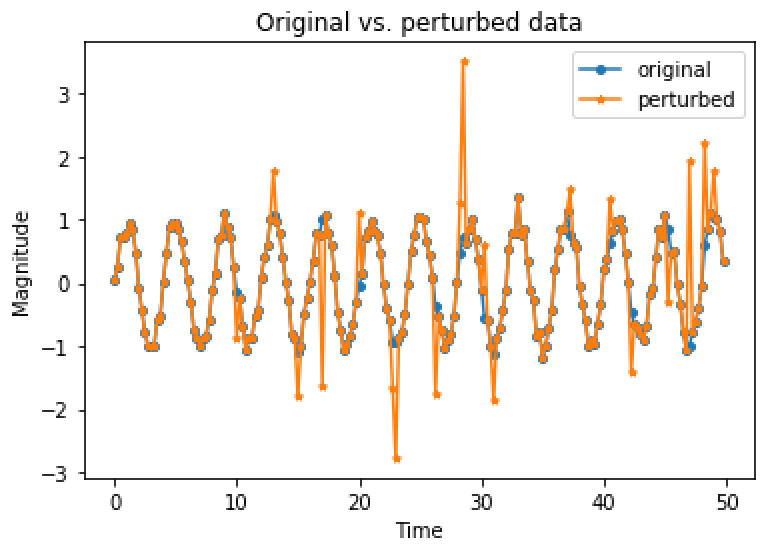
```
[5]: n_plot = 200
```

```
[6]: plt.plot(time_samples[:n_plot], X[:n_plot], marker='o', markersize=4, label='sample')
plt.plot(time_samples[:n_plot], signals[:n_plot], marker='*', markersize=4, label=
↪'signal')
plt.plot(time_samples[:n_plot], errors[:n_plot], marker='.', markersize=4, label=
↪'noise')
plt.xlabel('Time')
plt.ylabel('Magnitude')
plt.title('Original sinusoid with noise')
plt.legend()
plt.show();
```



Perturbed data:

```
[7]: plt.plot(time_samples[:n_plot], X[:n_plot], marker='o', markersize=4, label='original
     ↪')
     plt.plot(time_samples[:n_plot], X_outlier[:n_plot], marker='*', markersize=4, label=
     ↪'perturbed')
     plt.xlabel('Time')
     plt.ylabel('Magnitude')
     plt.title('Original vs. perturbed data')
     plt.legend()
     plt.show();
```



34.4 Define Spectral Residual outlier detector

[8]:
```
od = SpectralResidual(
    threshold=None,  # threshold for outlier score
    window_amp=20,   # window for the average log amplitude
    window_local=20, # window for the average saliency map
    n_est_points=20  # nb of estimated points padded to the end of the sequence
)
```

```
WARNING:alibi_detect.od.sr:No threshold level set. Need to infer threshold using␣
↪`infer_threshold`.
```

The warning tells us that we need to set the outlier threshold. This can be done with the `infer_threshold` method. We need to pass a batch of instances and specify what percentage of those we consider to be normal via `threshold_perc`. Let's assume we have some data which we know contains around 10% outliers:

[9]:
```
X_threshold = X_outlier[:10000, :]
```

Let's infer the threshold:

[10]:
```
od.infer_threshold(X_threshold, time_samples[:10000], threshold_perc=90)
print('New threshold: {:.4f}'.format(od.threshold))
```

```
New threshold: 1.1818
```

Let's save the outlier detector with the updated threshold:

[11]:
```
filepath = 'my_path'
save_detector(od, filepath)
```

We can load the same detector via `load_detector`:

[12]:
```
od = load_detector(filepath)
```

34.5 Detect outliers

Predict outliers:

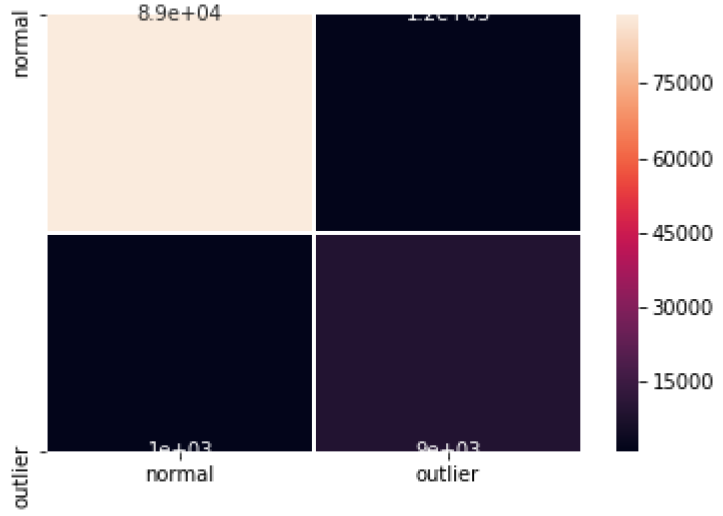
[13]:
```
od_preds = od.predict(X_outlier, time_samples, return_instance_score=True)
```

34.6 Display results

F1 score, accuracy, recall and confusion matrix:

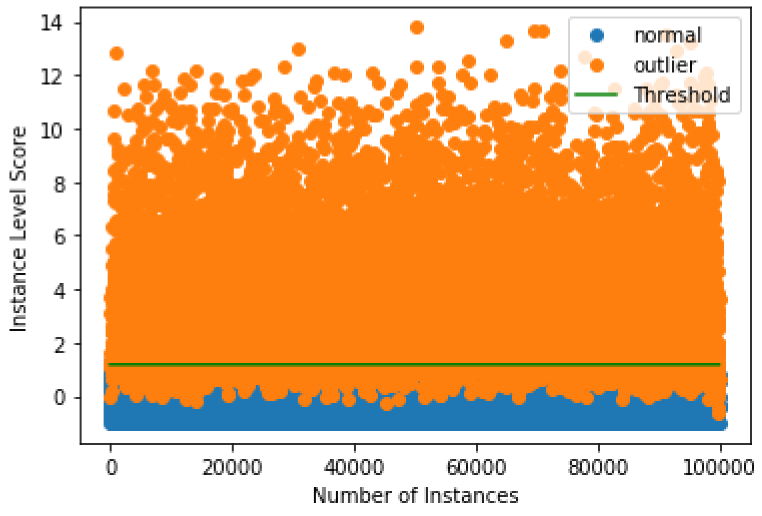
[14]:
```
y_pred = od_preds['data']['is_outlier']
f1 = f1_score(y_outlier, y_pred)
acc = accuracy_score(y_outlier, y_pred)
rec = recall_score(y_outlier, y_pred)
print('F1 score: {} -- Accuracy: {} -- Recall: {}'.format(f1, acc, rec))
cm = confusion_matrix(y_outlier, y_pred)
df_cm = pd.DataFrame(cm, index=labels, columns=labels)
sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
plt.show()
```

```
F1 score: 0.8922542204568024 -- Accuracy: 0.9783 -- Recall: 0.8985
```



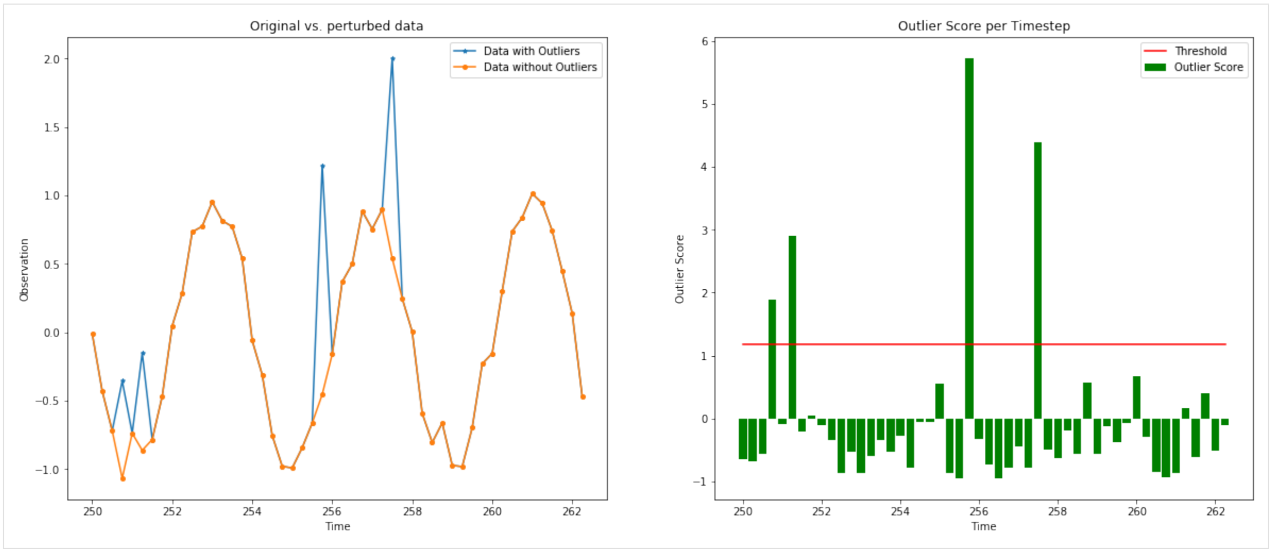
Plot the outlier scores of the time series vs. the outlier threshold. :

```
[15]: plot_instance_score(od_preds, y_outlier, labels, od.threshold)
```



Let's zoom in on a smaller time scale to have a clear picture:

```
[17]: plot_feature_outlier_ts(od_preds,
                              X_outlier,
                              od.threshold,
                              window=(1000, 1050),
                              t=time_samples,
                              X_orig=X)
```


Original vs. perturbed data
Data with Outliers
Data without Outliers
Observation
Time
Outlier Score per Timestep
Threshold
Outlier Score
Outlier Score
Time

CHAPTER

THIRTYFIVE

TIME SERIES OUTLIER DETECTION WITH SEQ2SEQ MODELS ON SYNTHETIC DATA

35.1 Method

The Sequence-to-Sequence (Seq2Seq) outlier detector consists of 2 main building blocks: an encoder and a decoder. The encoder consists of a Bidirectional LSTM which processes the input sequence and initializes the decoder. The LSTM decoder then makes sequential predictions for the output sequence. In our case, the decoder aims to reconstruct the input sequence. If the input data cannot be reconstructed well, the reconstruction error is high and the data can be flagged as an outlier. The reconstruction error is measured as the mean squared error (MSE) between the input and the reconstructed instance.

Since even for normal data the reconstruction error can be state-dependent, we add an outlier threshold estimator network to the Seq2Seq model. This network takes in the hidden state of the decoder at each timestep and predicts the estimated reconstruction error for normal data. As a result, the outlier threshold is not static and becomes a function of the model state. This is similar to Park et al. (2017), but while they train the threshold estimator separately from the Seq2Seq model with a Support-Vector Regressor, we train a neural net regression network end-to-end with the Seq2Seq model.

The detector is first trained on a batch of unlabeled, but normal (*inlier*) data. Unsupervised training is desireable since labeled data is often scarce. The Seq2Seq outlier detector is suitable for both **univariate and multivariate time series**.

35.2 Dataset

We test the outlier detector on a synthetic dataset generated with the TimeSynth package. It allows you to generate a wide range of time series (e.g. pseudo-periodic, autoregressive or Gaussian Process generated signals) and noise types (white or red noise). It can be installed as follows:

```
!pip install git+https://github.com/TimeSynth/TimeSynth.git
```

[1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.metrics import accuracy_score, confusion_matrix, f1_score, recall_score
import tensorflow as tf
import timesynth as ts

from alibi_detect.od import OutlierSeq2Seq
```

(continues on next page)

(continued from previous page)

```
from alibi_detect.utils.perturbation import inject_outlier_ts
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.utils.visualize import plot_feature_outlier_ts, plot_roc
```

35.3 Create multivariate time series

Define number of sampled points and the type of simulated time series. We use TimeSynth to generate sinusoidal signals with noise.

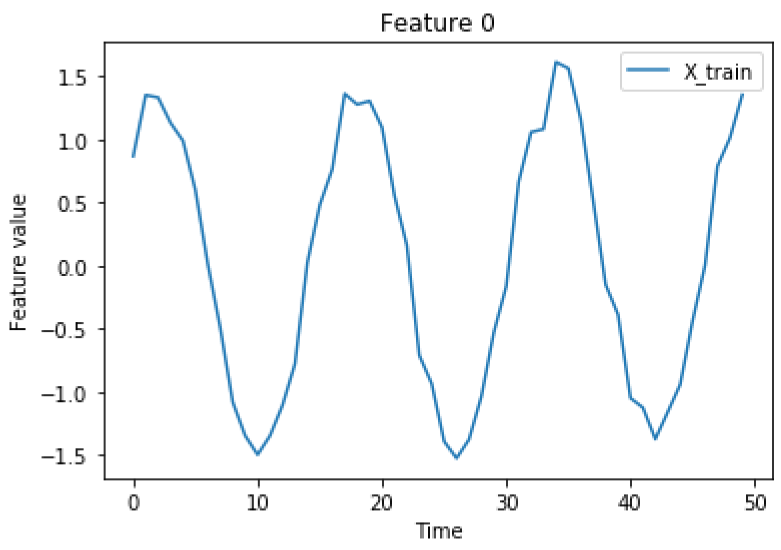
```
[2]: n_points = int(1e6)  # number of timesteps
     perc_train = 80  # percentage of instances used for training
     perc_threshold = 10  # percentage of instances used to determine threshold
     n_train = int(n_points * perc_train * .01)
     n_threshold = int(n_points * perc_threshold * .01)
     n_features = 2  # number of features in the time series
     seq_len = 50  # sequence length
```

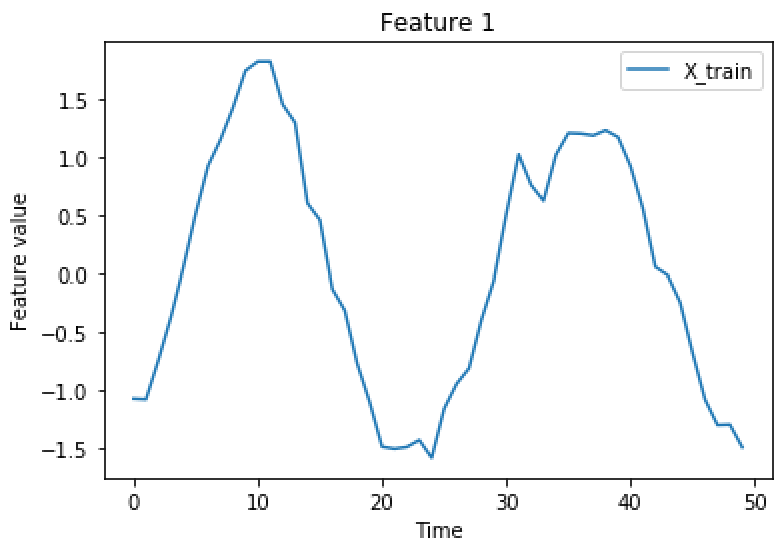
```
[3]: # set random seed
     np.random.seed(0)

     # timestamps
     time_sampler = ts.TimeSampler(stop_time=n_points // 4)
     time_samples = time_sampler.sample_regular_time(num_points=n_points)

     # create time series
     ts1 = ts.TimeSeries(
         signal_generator=ts.signals.Sinusoidal(frequency=0.25),
         noise_generator=ts.noise.GaussianNoise(std=0.1)
     )
     samples1 = ts1.sample(time_samples)[0].reshape(-1, 1)

     ts2 = ts.TimeSeries(
         signal_generator=ts.signals.Sinusoidal(frequency=0.15),
         noise_generator=ts.noise.RedNoise(std=.7, tau=0.5)
     )
     samples2 = ts2.sample(time_samples)[0].reshape(-1, 1)

     # combine signals
     X = np.concatenate([samples1, samples2], axis=1).astype(np.float32)

     # split dataset into train, infer threshold and outlier detection sets
     X_train = X[:n_train]
     X_threshold = X[n_train:n_train+n_threshold]
     X_outlier = X[n_train+n_threshold:]

     # scale using the normal training data
     mu, sigma = X_train.mean(axis=0), X_train.std(axis=0)
     X_train = (X_train - mu) / sigma
     X_threshold = (X_threshold - mu) / sigma
     X_outlier = (X_outlier - mu) / sigma
     print(X_train.shape, X_threshold.shape, X_outlier.shape)
```

```
(800000, 2) (100000, 2) (100000, 2)
```

Visualize:

```
[4]: n_features = X.shape[-1]
     istart, istop = 50, 100
     for f in range(n_features):
         plt.plot(X_train[istart:istop, f], label='X_train')
         plt.title('Feature {}'.format(f))
         plt.xlabel('Time')
         plt.ylabel('Feature value')
         plt.legend()
         plt.show()
```





35.4 Load or define Seq2Seq outlier detector

```
[5]: load_outlier_detector = False
```

```
[6]: filepath = 'my_path'  # change to directory where model is saved
if load_outlier_detector:  # load pretrained outlier detector
    od = load_detector(filepath)
else:  # define model, initialize, train and save outlier detector

    # initialize outlier detector
    od = OutlierSeq2Seq(n_features,
                        seq_len,
                        threshold=None,
                        latent_dim=100)

    # train
    od.fit(X_train,
           epochs=10,
           verbose=False)

    # save the trained outlier detector
    save_detector(od, filepath)
```

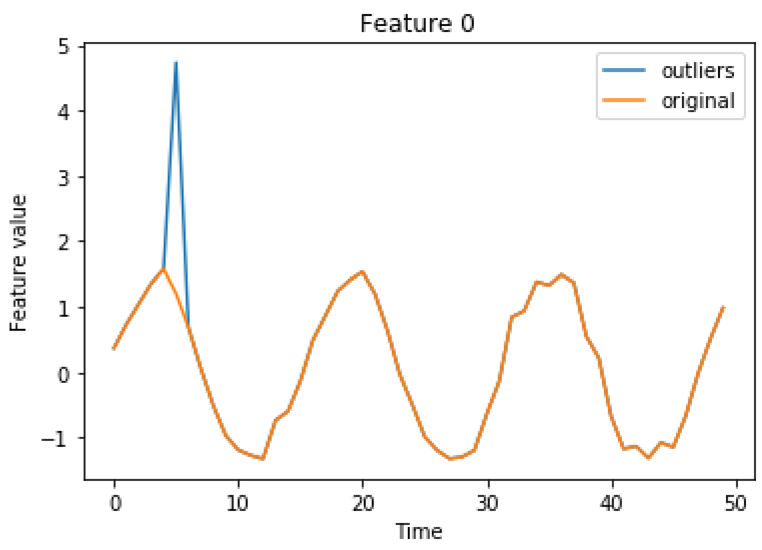
We still need to set the outlier threshold. This can be done with the `infer_threshold` method. We need to pass a time series of instances and specify what percentage of those we consider to be normal via `threshold_perc`. First we create outliers by injecting noise in the time series via `inject_outlier_ts`. The noise can be regulated via the percentage of outliers (`perc_outlier`), the strength of the perturbation (`n_std`) and the minimum size of the noise perturbation (`min_std`). Let's assume we have some data which we know contains around 10% outliers in either of the features:

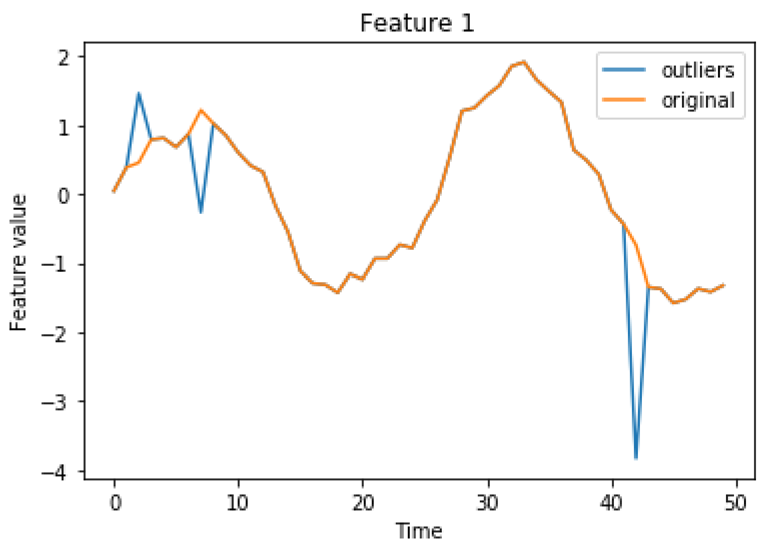
```
[7]: np.random.seed(0)

X_thr = X_threshold.copy()
data = inject_outlier_ts(X_threshold, perc_outlier=10, perc_window=10, n_std=2., min_
↪std=1.)
X_threshold = data.data
print(X_threshold.shape)
```

```
(100000, 2)
```

Visualize outlier data used to determine the threshold:

```
[8]: istart, istop = 0, 50
for f in range(n_features):
    plt.plot(X_threshold[istart:istop, f], label='outliers')
    plt.plot(X_thr[istart:istop, f], label='original')
    plt.title('Feature {}'.format(f))
    plt.xlabel('Time')
    plt.ylabel('Feature value')
    plt.legend()
    plt.show()
```





Let's infer the threshold. The `inject_outlier_ts` method distributes perturbations evenly across features. As a result, each feature contains about 5% outliers. We can either set the threshold over both features combined or determine a feature-wise threshold. Here we opt for the **feature-wise threshold**. This is for instance useful when different features have different variance or sensitivity to outliers. We also manually decrease the threshold a bit to increase the sensitivity of our detector:

```
[9]: od.infer_threshold(X_threshold, threshold_perc=[95, 95])
     od.threshold -= .15
     print('New threshold: {}'.format(od.threshold))
```

```
New threshold: [[0.12968134 0.29728393]]
```

Let's save the outlier detector with the updated threshold:

```
[10]: save_detector(od, filepath)
```

We can load the same detector via `load_detector`:

[11]:
```
od = load_detector(filepath)
```

35.5 Detect outliers

Generate the outliers to detect:

[12]:
```
np.random.seed(1)

X_out = X_outlier.copy()
data = inject_outlier_ts(X_outlier, perc_outlier=10, perc_window=10, n_std=2., min_
↪std=1.)
X_outlier, y_outlier, labels = data.data, data.target.astype(int), data.target_names
print(X_outlier.shape, y_outlier.shape)
```

```
(100000, 2) (100000,)
```

Predict outliers:

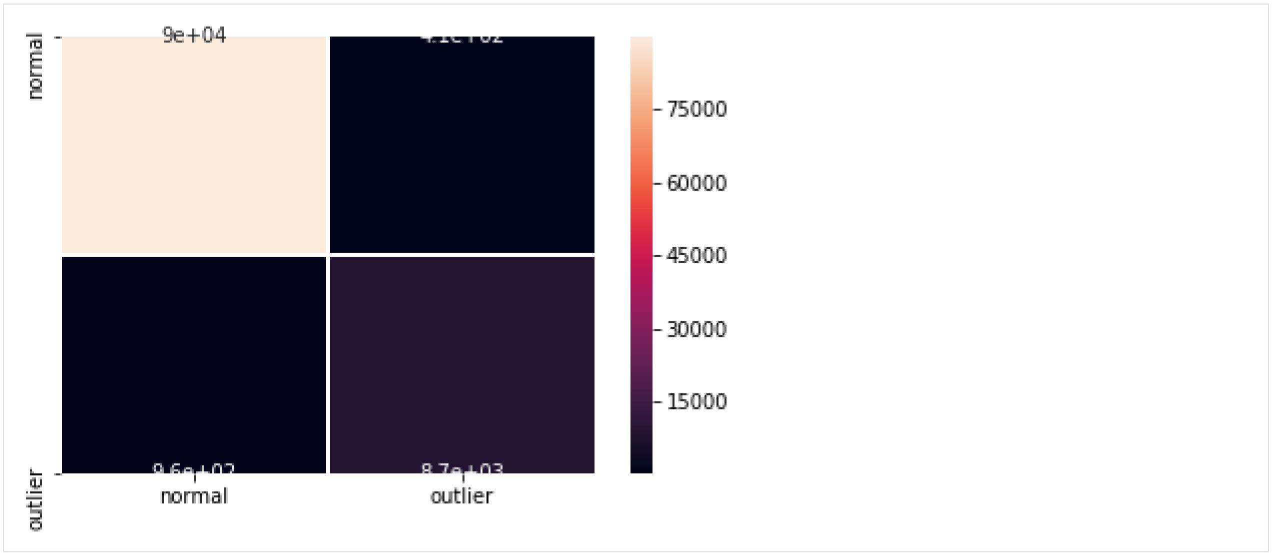
[13]:
```
od_preds = od.predict(X_outlier,
                      outlier_type='instance',    # use 'feature' or 'instance' level
                      return_feature_score=True,  # scores used to determine outliers
                      return_instance_score=True)
```

35.6 Display results

F1 score, accuracy, recall and confusion matrix:

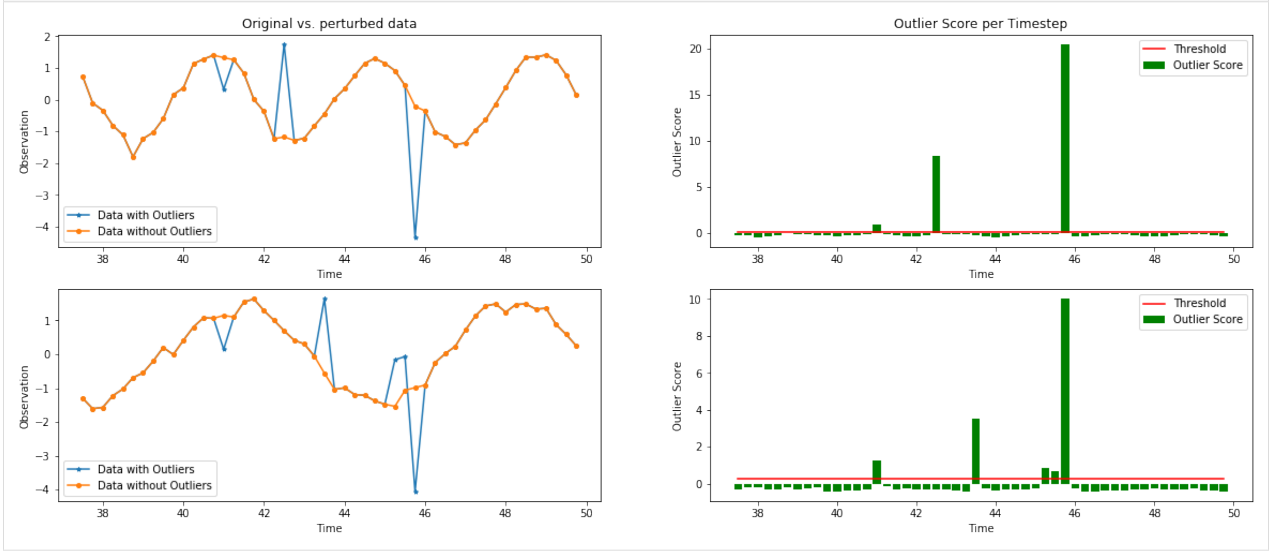
[14]:
```
y_pred = od_preds['data']['is_outlier']
f1 = f1_score(y_outlier, y_pred)
acc = accuracy_score(y_outlier, y_pred)
rec = recall_score(y_outlier, y_pred)
print('F1 score: {:.3f} -- Accuracy: {:.3f} -- Recall: {:.3f}'.format(f1, acc, rec))
cm = confusion_matrix(y_outlier, y_pred)
df_cm = pd.DataFrame(cm, index=labels, columns=labels)
sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
plt.show()
```

```
F1 score: 0.927 -- Accuracy: 0.986 -- Recall: 0.901
```



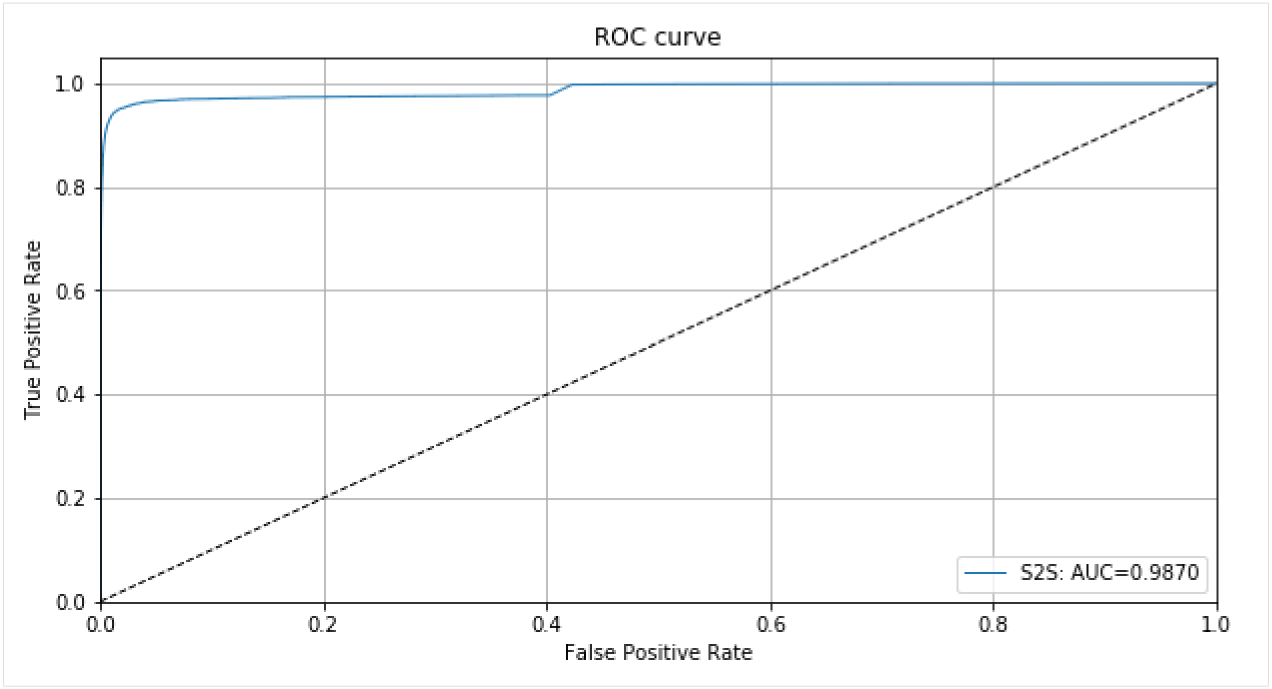
Plot the feature-wise outlier scores of the time series for each timestep vs. the outlier threshold:

```
[17]: plot_feature_outlier_ts(od_preds,
                              X_outlier,
                              od.threshold[0],
                              window=(150, 200),
                              t=time_samples,
                              X_orig=X_out)
```



We can also plot the ROC curve using the instance level outlier scores:

```
[18]: roc_data = {'S2S': {'scores': od_preds['data']['instance_score'], 'labels': y_outlier}
      ↪}
      plot_roc(roc_data)
```


ROC curve
True Positive Rate
False Positive Rate
S2S: AUC=0.9870
0.0
0.2
0.4
0.6
0.8
1.0

CHAPTER

THIRTYSIX

SEQ2SEQ TIME SERIES OUTLIER DETECTION ON ECG DATA

36.1 Method

The Sequence-to-Sequence (Seq2Seq) outlier detector consists of 2 main building blocks: an encoder and a decoder. The encoder consists of a Bidirectional LSTM which processes the input sequence and initializes the decoder. The LSTM decoder then makes sequential predictions for the output sequence. In our case, the decoder aims to reconstruct the input sequence. If the input data cannot be reconstructed well, the reconstruction error is high and the data can be flagged as an outlier. The reconstruction error is measured as the mean squared error (MSE) between the input and the reconstructed instance.

Since even for normal data the reconstruction error can be state-dependent, we add an outlier threshold estimator network to the Seq2Seq model. This network takes in the hidden state of the decoder at each timestep and predicts the estimated reconstruction error for normal data. As a result, the outlier threshold is not static and becomes a function of the model state. This is similar to Park et al. (2017), but while they train the threshold estimator separately from the Seq2Seq model with a Support-Vector Regressor, we train a neural net regression network end-to-end with the Seq2Seq model.

The detector is first trained on a batch of unlabeled, but normal (*inlier*) data. Unsupervised training is desireable since labeled data is often scarce. The Seq2Seq outlier detector is suitable for both **univariate and multivariate time series**.

36.2 Dataset

The outlier detector needs to spot anomalies in electrocardiograms (ECG's). The dataset contains 5000 ECG's, originally obtained from Physionet under the name *BIDMC Congestive Heart Failure Database(chfdb)*, record *chf07*. The data has been pre-processed in 2 steps: first each heartbeat is extracted, and then each beat is made equal length via interpolation. The data is labeled and contains 5 classes. The first class which contains almost 60% of the observations is seen as *normal* while the others are outliers. The detector is trained on heartbeats from the first class and needs to flag the other classes as anomalies.

[1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import os
import pandas as pd
import seaborn as sns
from sklearn.metrics import accuracy_score, confusion_matrix, f1_score, precision_
↪score, recall_score

from alibi_detect.od import OutlierSeq2Seq
from alibi_detect.utils.fetching import fetch_detector
```

(continues on next page)

(continued from previous page)

```
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.datasets import fetch_ecg
from alibi_detect.utils.visualize import plot_roc
```

36.3 Load dataset

Flip train and test data because there are only 500 ECG's in the original training set and 4500 in the test set:

```
[2]: (X_test, y_test), (X_train, y_train) = fetch_ecg(return_X_y=True)
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

```
(4500, 140) (4500,)
(500, 140) (500,)
```

Since we treat the first class as the normal, *inlier* data and the rest of *X_train* as outliers, we need to adjust the training (inlier) data and the labels of the test set.

```
[3]: inlier_idx = np.where(y_train == 1)[0]
X_inlier, y_inlier = X_train[inlier_idx], np.zeros_like(y_train[inlier_idx])
outlier_idx = np.where(y_train != 1)[0]
X_outlier, y_outlier = X_train[outlier_idx], y_train[outlier_idx]
y_test[y_test == 1] = 0  # class 1 represent the inliers
y_test[y_test != 0] = 1
print(X_inlier.shape, X_outlier.shape)
```

```
(2627, 140) (1873, 140)
```

Some of the outliers in *X_train* are used in combination with some of the inlier instances to infer the threshold level:

```
[4]: n_threshold = 1000
perc_inlier = 60
n_inlier = int(perc_inlier * .01 * n_threshold)
n_outlier = int((100 - perc_inlier) * .01 * n_threshold)
idx_thr_in = np.random.choice(X_inlier.shape[0], n_inlier, replace=False)
idx_thr_out = np.random.choice(X_outlier.shape[0], n_outlier, replace=False)
X_threshold = np.concatenate([X_inlier[idx_thr_in], X_outlier[idx_thr_out]], axis=0)
y_threshold = np.zeros(n_threshold).astype(int)
y_threshold[-n_outlier:] = 1
print(X_threshold.shape, y_threshold.shape)
```

```
(1000, 140) (1000,)
```

Apply min-max scaling between 0 and 1 to the observations using the inlier data:

```
[5]: xmin, xmax = X_inlier.min(), X_inlier.max()
rng = (0, 1)
X_inlier = ((X_inlier - xmin) / (xmax - xmin)) * (rng[1] - rng[0]) + rng[0]
X_threshold = ((X_threshold - xmin) / (xmax - xmin)) * (rng[1] - rng[0]) + rng[0]
X_test = ((X_test - xmin) / (xmax - xmin)) * (rng[1] - rng[0]) + rng[0]
X_outlier = ((X_outlier - xmin) / (xmax - xmin)) * (rng[1] - rng[0]) + rng[0]
print('Inlier: min {:.2f} --- max {:.2f}'.format(X_inlier.min(), X_inlier.max()))
print('Threshold: min {:.2f} --- max {:.2f}'.format(X_threshold.min(), X_threshold.
↪max()))
print('Test: min {:.2f} --- max {:.2f}'.format(X_test.min(), X_test.max()))
```

```
Inlier: min 0.00 --- max 1.00
Threshold: min 0.00 --- max 0.99
Test: min 0.11 --- max 0.92
```

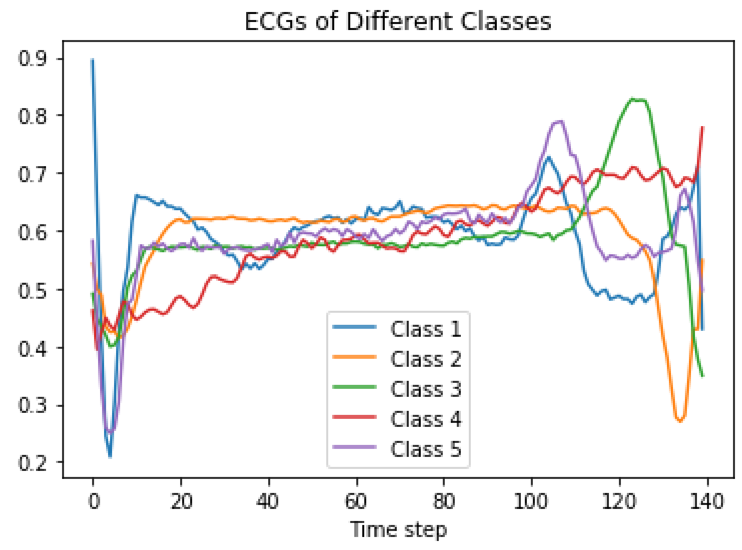
Reshape the observations to *(batch size, sequence length, features)* for the detector:

```
[6]: shape = (-1, X_inlier.shape[1], 1)
X_inlier = X_inlier.reshape(shape)
X_threshold = X_threshold.reshape(shape)
X_test = X_test.reshape(shape)
X_outlier = X_outlier.reshape(shape)
print(X_inlier.shape, X_threshold.shape, X_test.shape)
```

```
(2627, 140, 1) (1000, 140, 1) (500, 140, 1)
```

We can now visualize scaled instances from each class:

```
[7]: idx_plt = [np.where(y_outlier == i)[0][0] for i in list(np.unique(y_outlier))]
X_plt = np.concatenate([X_inlier[0:1], X_outlier[idx_plt]], axis=0)

for i in range(X_plt.shape[0]):
    plt.plot(X_plt[i], label='Class ' + str(i+1))
plt.title('ECGs of Different Classes')
plt.xlabel('Time step')
plt.legend()
plt.show()
```



36.4 Load or define Seq2Seq outlier detector

The pretrained outlier and adversarial detectors used in the example notebooks can be found here. You can use the built-in `fetch_detector` function which saves the pre-trained models in a local directory `filepath` and loads the detector. Alternatively, you can train a detector from scratch:

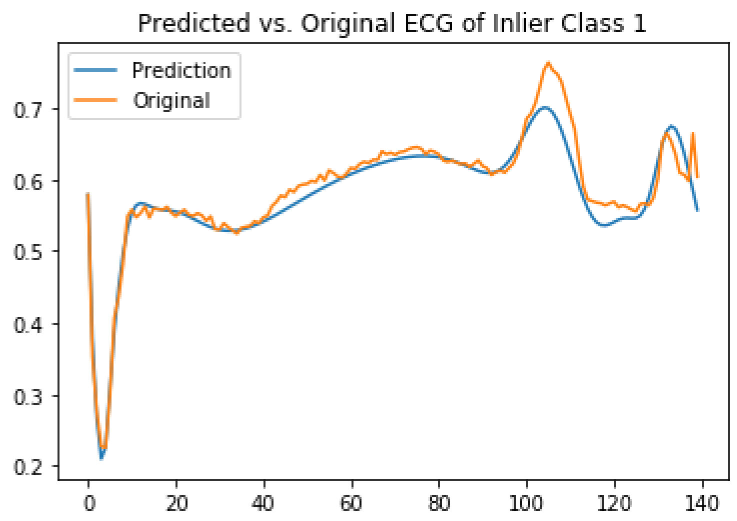
```
[8]: load_outlier_detector = True
```

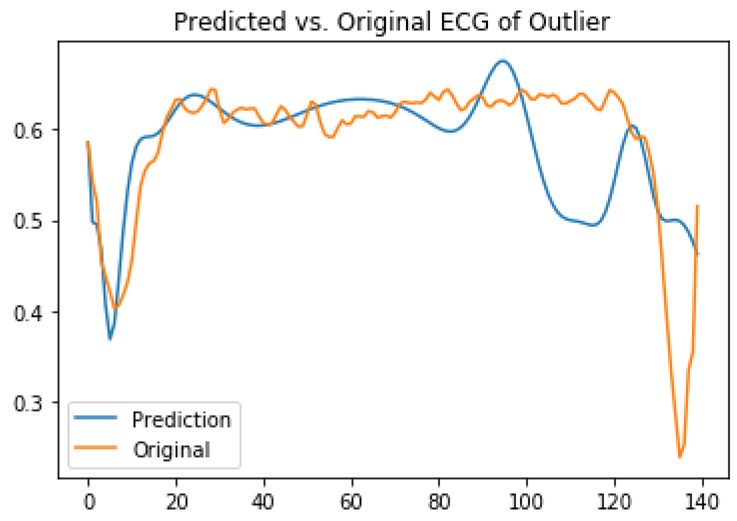
```
[9]: filepath = 'my_path'  # change to (absolute) directory where model is downloaded
if load_outlier_detector:  # load pretrained outlier detector
    detector_type = 'outlier'
    dataset = 'ecg'
    detector_name = 'OutlierSeq2Seq'
    od = fetch_detector(filepath, detector_type, dataset, detector_name)
    filepath = os.path.join(filepath, detector_name)
else:  # define model, initialize, train and save outlier detector

    # initialize outlier detector
    od = OutlierSeq2Seq(1,
                        X_inlier.shape[1],  # sequence length
                        threshold=None,
                        latent_dim=40)

    # train
    od.fit(X_inlier,
           epochs=100,
           verbose=False)

    # save the trained outlier detector
    save_detector(od, filepath)
```

Let's inspect how well the sequence-to-sequence model can predict the ECG's of the inlier and outlier classes. The predictions in the charts below are made on ECG's from the test set:

```
[10]: ecg_pred = od.seq2seq.decode_seq(X_test)[0]
```

```
[11]: i_normal = np.where(y_test == 0)[0][0]
plt.plot(ecg_pred[i_normal], label='Prediction')
plt.plot(X_test[i_normal], label='Original')
plt.title('Predicted vs. Original ECG of Inlier Class 1')
plt.legend()
plt.show()

i_outlier = np.where(y_test == 1)[0][0]
plt.plot(ecg_pred[i_outlier], label='Prediction')
plt.plot(X_test[i_outlier], label='Original')
plt.title('Predicted vs. Original ECG of Outlier')
plt.legend()
plt.show()
```





It is clear that the model can reconstruct the inlier class but struggles with the outliers.

If we trained a model from scratch, the warning thrown when we initialized the model tells us that we need to set the outlier threshold. This can be done with the `infer_threshold` method. We need to pass a time series of instances and specify what percentage of those we consider to be normal via `threshold_perc`, equal to the percentage of *Class 1* in *X_threshold*. The `outlier_perc` parameter defines the percentage of features used to define the outlier threshold. In this example, the number of features considered per instance equals 140 (1 for each timestep). We set the `outlier_perc` at 95, which means that we will use the 95% features with highest reconstruction error, adjusted for by the threshold estimate.

[12]:
```
od.infer_threshold(X_threshold, outlier_perc=95, threshold_perc=perc_inlier)
print('New threshold: {}'.format(od.threshold))
```

```
New threshold: 0.002807901981854227
```

Let’s save the outlier detector with the updated threshold:

[13]:
```
save_detector(od, filepath)
```

We can load the same detector via `load_detector`:

```
[14]: od = load_detector(filepath)
```

36.5 Detect outliers

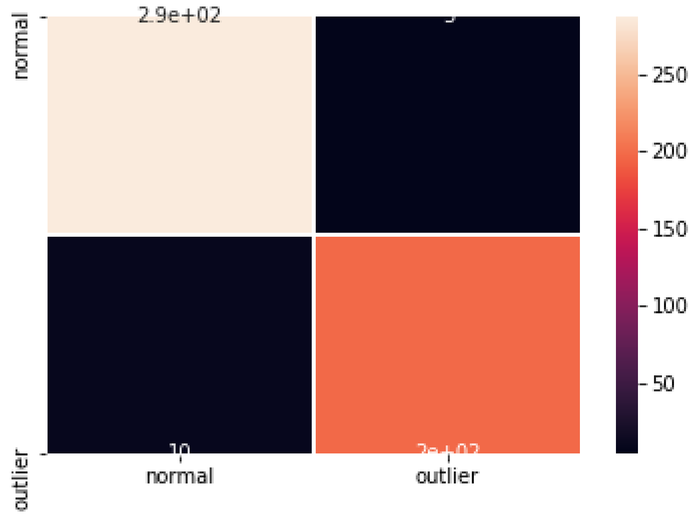
```
[15]: od_preds = od.predict(X_test,
                            outlier_type='instance',    # use 'feature' or 'instance' level
                            return_feature_score=True,  # scores used to determine outliers
                            return_instance_score=True)
```

36.6 Display results

F1 score, accuracy, recall and confusion matrix:

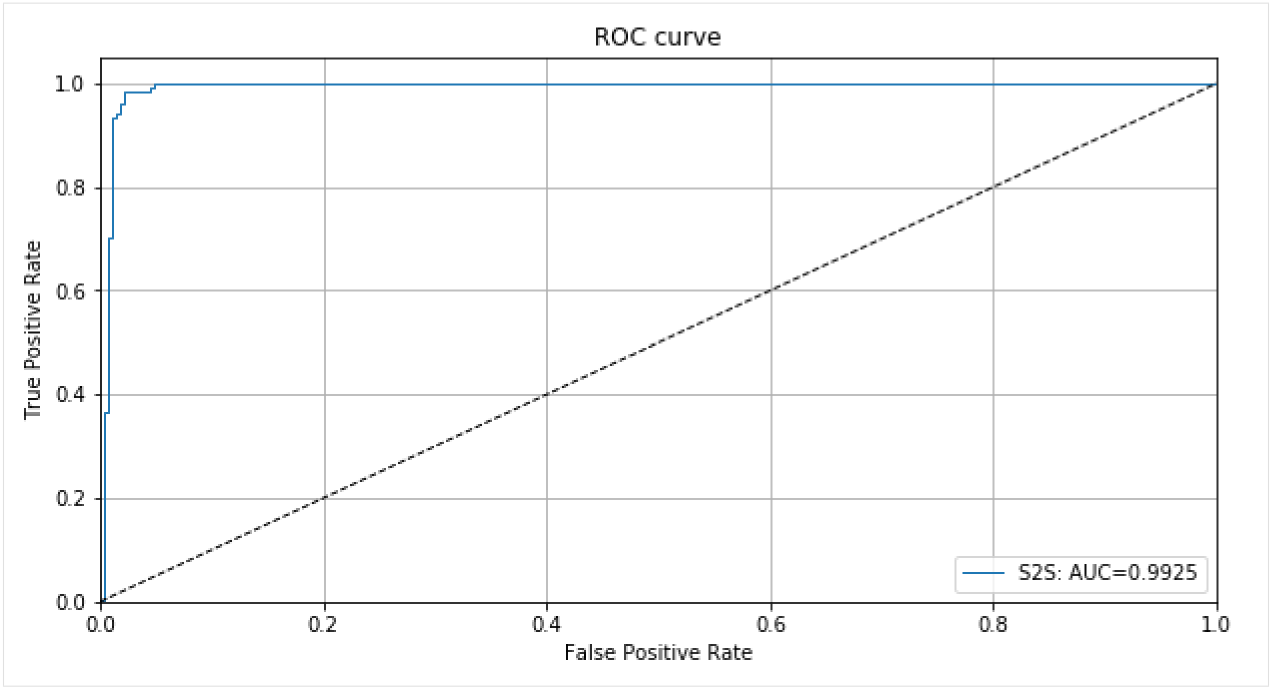
```
[16]: y_pred = od_preds['data']['is_outlier']
labels = ['normal', 'outlier']
f1 = f1_score(y_test, y_pred)
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
print('F1 score: {:.3f} -- Accuracy: {:.3f} -- Precision: {:.3f} -- Recall: {:.3f}'.
↪format(f1, acc, prec, rec))
cm = confusion_matrix(y_test, y_pred)
df_cm = pd.DataFrame(cm, index=labels, columns=labels)
sns.heatmap(df_cm, annot=True, cbar=True, linewidths=.5)
plt.show()
```

```
F1 score: 0.964 -- Accuracy: 0.970 -- Precision: 0.975 -- Recall: 0.952
```



We can also plot the ROC curve based on the instance level outlier scores:

```
[17]: roc_data = {'S2S': {'scores': od_preds['data']['instance_score'], 'labels': y_test}}
plot_roc(roc_data)
```


ROC curve
True Positive Rate
False Positive Rate
S2S: AUC=0.9925
0.0
0.2
0.4
0.6
0.8
1.0

CHAPTER

THIRTYSEVEN

ADVERSARIAL AE DETECTION AND CORRECTION ON CIFAR-10

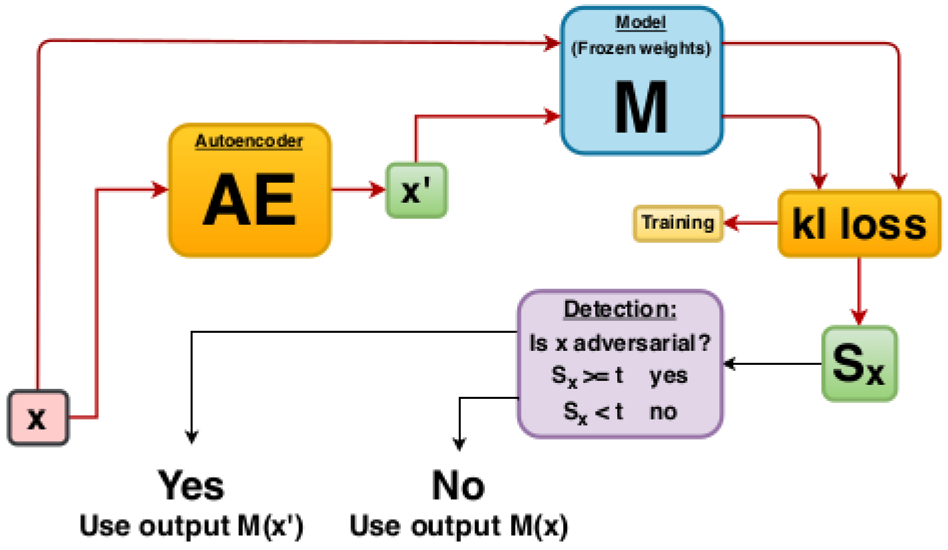
37.1 Method

The adversarial detector is based on Adversarial Detection and Correction by Matching Prediction Distributions. Usually, autoencoders are trained to find a transformation T that reconstructs the input instance x as accurately as possible with loss functions that are suited to capture the similarities between x and x' such as the mean squared reconstruction error. The novelty of the adversarial autoencoder (AE) detector relies on the use of a classification model-dependent loss function based on a distance metric in the output space of the model to train the autoencoder network. Given a classification model M we optimise the weights of the autoencoder such that the KL-divergence between the model predictions on x and on x' is minimised. Without the presence of a reconstruction loss term x' simply tries to make sure that the prediction probabilities $M(x')$ and $M(x)$ match without caring about the proximity of x' to x. As a result, x' is allowed to live in different areas of the input feature space than x with different decision boundary shapes with respect to the model M. The carefully crafted adversarial perturbation which is effective around x does not transfer to the new location of x' in the feature space, and the attack is therefore neutralised. Training of the autoencoder is unsupervised since we only need access to the model prediction probabilities and the normal training instances. We do not require any knowledge about the underlying adversarial attack and the classifier weights are frozen during training.

The detector can be used as follows:

- An adversarial score S is computed. S equals the K-L divergence between the model predictions on x and x'.
- If S is above a threshold (explicitly defined or inferred from training data), the instance is flagged as adversarial.
- For adversarial instances, the model M uses the reconstructed instance x' to make a prediction. If the adversarial score is below the threshold, the model makes a prediction on the original instance x.

This procedure is illustrated in the diagram below:



The method is very flexible and can also be used to detect common data corruptions and perturbations which negatively impact the model performance.

37.1.1 Dataset

CIFAR10 consists of 60,000 32 by 32 RGB images equally distributed over 10 classes.

Note: in order to run this notebook, it is adviced to use **Python 3.7** and have a **GPU** enabled.

```
[1]: import matplotlib.pyplot as plt
import numpy as np
import os
from sklearn.metrics import roc_curve, auc
import tensorflow as tf
from tensorflow.keras.layers import (Conv2D, Conv2DTranspose, Dense, Flatten,
                                     InputLayer, Reshape)
from tensorflow.keras.regularizers import l1

from alibi_detect.ad import AdversarialAE
from alibi_detect.utils.fetching import fetch_detector, fetch_tf_model
from alibi_detect.utils.prediction import predict_batch
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.datasets import fetch_attack, fetch_cifar10c, corruption_types_
↪cifar10c
```

```
ERROR:fbprophet:Importing plotly failed. Interactive plots will not work.
```

37.2 Utility functions

```
[2]: def scale_by_instance(X: np.ndarray) -> np.ndarray:
         mean_ = X.mean(axis=(1, 2, 3)).reshape(-1, 1, 1, 1)
         std_ = X.std(axis=(1, 2, 3)).reshape(-1, 1, 1, 1)
         return (X - mean_) / std_, mean_, std_


     def accuracy(y_true: np.ndarray, y_pred: np.ndarray) -> float:
         return (y_true == y_pred).astype(int).sum() / y_true.shape[0]


     def plot_adversarial(idx: list,
                          X: np.ndarray,
                          y: np.ndarray,
                          X_adv: np.ndarray,
                          y_adv: np.ndarray,
                          mean: np.ndarray,
                          std: np.ndarray,
                          score_x: np.ndarray = None,
                          score_x_adv: np.ndarray = None,
                          X_recon: np.ndarray = None,
                          y_recon: np.ndarray = None,
                          figsize: tuple = (10, 5)) -> None:

         # category map from class numbers to names
         cifar10_map = {0: 'airplane', 1: 'automobile', 2: 'bird', 3: 'cat', 4: 'deer', 5:
     ↪'dog',
                        6: 'frog', 7: 'horse', 8: 'ship', 9: 'truck'}

         nrows = len(idx)
         ncols = 3 if isinstance(X_recon, np.ndarray) else 2
         fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)

         n_subplot = 1
         for i in idx:

             # rescale images in [0, 1]
             X_adj = (X[i] * std[i] + mean[i]) / 255
             X_adv_adj = (X_adv[i] * std[i] + mean[i]) / 255
             if isinstance(X_recon, np.ndarray):
                 X_recon_adj = (X_recon[i] * std[i] + mean[i]) / 255

             # original image
             plt.subplot(nrows, ncols, n_subplot)
             plt.axis('off')
             if i == idx[0]:
                 if isinstance(score_x, np.ndarray):
                     plt.title('CIFAR-10 Image \n{}: {:.3f}'.format(cifar10_map[y[i]],␣
     ↪score_x[i]))
                 else:
                     plt.title('CIFAR-10 Image \n{}'.format(cifar10_map[y[i]]))
             else:
                 if isinstance(score_x, np.ndarray):
                     plt.title('{}: {:.3f}'.format(cifar10_map[y[i]], score_x[i]))
                 else:
                     plt.title('{}'.format(cifar10_map[y[i]]))
```

(continues on next page)

(continued from previous page)

```
        plt.imshow(X_adj)
        n_subplot += 1

        # adversarial image
        plt.subplot(nrows, ncols, n_subplot)
        plt.axis('off')
        if i == idx[0]:
            if isinstance(score_x_adv, np.ndarray):
                plt.title('Adversarial \n{}: {:.3f}'.format(cifar10_map[y_adv[i]],␣
↪score_x_adv[i]))
            else:
                plt.title('Adversarial \n{}'.format(cifar10_map[y_adv[i]]))
        else:
            if isinstance(score_x_adv, np.ndarray):
                plt.title('{}: {:.3f}'.format(cifar10_map[y_adv[i]], score_x_adv[i]))
            else:
                plt.title('{}'.format(cifar10_map[y_adv[i]]))
        plt.imshow(X_adv_adj)
        n_subplot += 1

        # reconstructed image
        if isinstance(X_recon, np.ndarray):
            plt.subplot(nrows, ncols, n_subplot)
            plt.axis('off')
            if i == idx[0]:
                plt.title('AE Reconstruction \n{}'.format(cifar10_map[y_recon[i]]))
            else:
                plt.title('{}'.format(cifar10_map[y_recon[i]]))
            plt.imshow(X_recon_adj)
            n_subplot += 1

    plt.show()


def plot_roc(roc_data: dict, figsize: tuple = (10,5)):
    plot_labels = []
    scores_attacks = []
    labels_attacks = []
    for k, v in roc_data.items():
        if 'original' in k:
            continue
        score_x = roc_data[v['normal']]['scores']
        y_pred = roc_data[v['normal']]['predictions']
        score_v = v['scores']
        y_pred_v = v['predictions']
        labels_v = np.ones(score_x.shape[0])
        idx_remove = np.where(y_pred == y_pred_v)[0]
        labels_v = np.delete(labels_v, idx_remove)
        score_v = np.delete(score_v, idx_remove)
        scores = np.concatenate([score_x, score_v])
        labels = np.concatenate([np.zeros(y_pred.shape[0]), labels_v]).astype(int)
        scores_attacks.append(scores)
        labels_attacks.append(labels)
        plot_labels.append(k)

    for sc_att, la_att, plt_la in zip(scores_attacks, labels_attacks, plot_labels):
        fpr, tpr, thresholds = roc_curve(la_att, sc_att)
```

(continues on next page)

(continued from previous page)

```
        roc_auc = auc(fpr, tpr)
        label = str('{}: AUC = {:.2f}'.format(plt_la, roc_auc))
        plt.plot(fpr, tpr, lw=1, label='{}: AUC={:.4f}'.format(plt_la, roc_auc))

    plt.plot([0, 1], [0, 1], color='black', lw=1, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('{}'.format('ROC curve'))
    plt.legend(loc="lower right", ncol=1)
    plt.grid()
    plt.show()
```

37.3 Load data

[3]:
```
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
y_train = y_train.astype('int64').reshape(-1,)
y_test = y_test.astype('int64').reshape(-1,)
```

Standardise the dataset by instance:

[4]:
```
X_train, mean_train, std_train = scale_by_instance(X_train)
X_test, mean_test, std_test = scale_by_instance(X_test)
scale = (mean_train, std_train), (mean_test, std_test)
```

37.4 Load classifier

[5]:
```
dataset = 'cifar10'
model = 'resnet56'
clf = fetch_tf_model(dataset, model)
```

Check that the predictions on the test set reach 93.15% accuracy:

[6]:
```
y_pred = predict_batch(clf, X_test, batch_size=32, return_class=True)
acc_y_pred = accuracy(y_test, y_pred)
print('Accuracy: {:.4f}'.format(acc_y_pred))
```

```
Accuracy: 0.9315
```

37.5 Adversarial Attack

We investigate both Carlini-Wagner (C&W) and SLIDE attacks. You can simply load previously found adversarial instances on the pretrained ResNet-56 model. The attacks are generated by using Foolbox:

[7]:
```
# C&W attack
data_cw = fetch_attack(dataset, model, 'cw')
X_train_cw, X_test_cw = data_cw['data_train'], data_cw['data_test']
meta_cw = data_cw['meta'] # metadata with hyperparameters of the attack
# SLIDE attack
data_slide = fetch_attack(dataset, model, 'slide')
X_train_slide, X_test_slide = data_slide['data_train'], data_slide['data_test']
meta_slide = data_slide['meta']
```

[8]:
```
print(X_test_cw.shape, X_test_slide.shape)
```

```
(10000, 32, 32, 3) (10000, 32, 32, 3)
```

Check if the prediction accuracy of the model on the adversarial instances is close to 0%.

[9]:
```
y_pred_cw = predict_batch(clf, X_test_cw, batch_size=32, return_class=True)
y_pred_slide = predict_batch(clf, X_test_slide, batch_size=32, return_class=True)
```

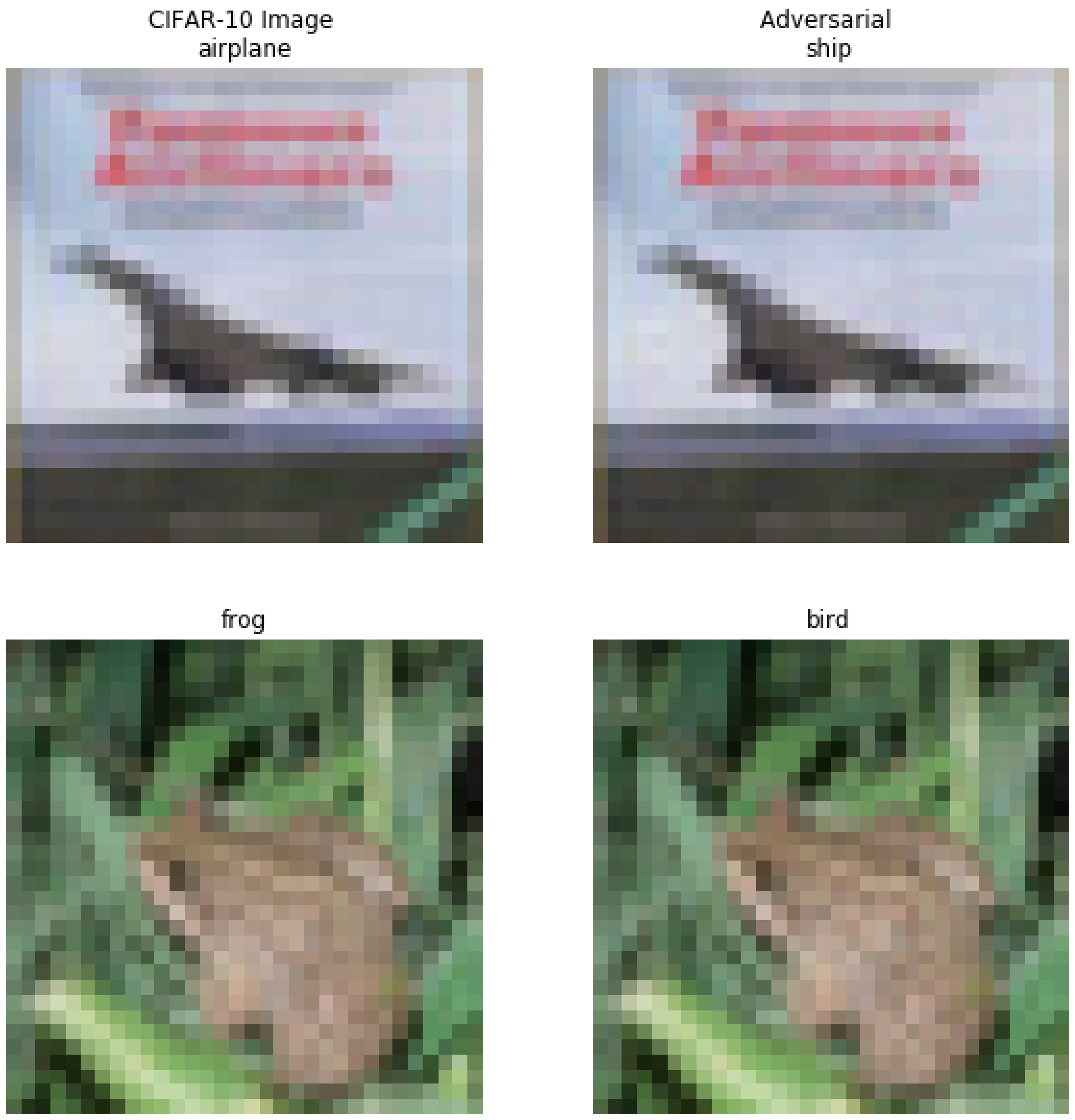
[10]:
```
acc_y_pred_cw = accuracy(y_test, y_pred_cw)
acc_y_pred_slide = accuracy(y_test, y_pred_slide)
print('Accuracy: cw {:.4f} -- SLIDE {:.4f}'.format(acc_y_pred_cw, acc_y_pred_slide))
```

```
Accuracy: cw 0.0000 -- SLIDE 0.0002
```

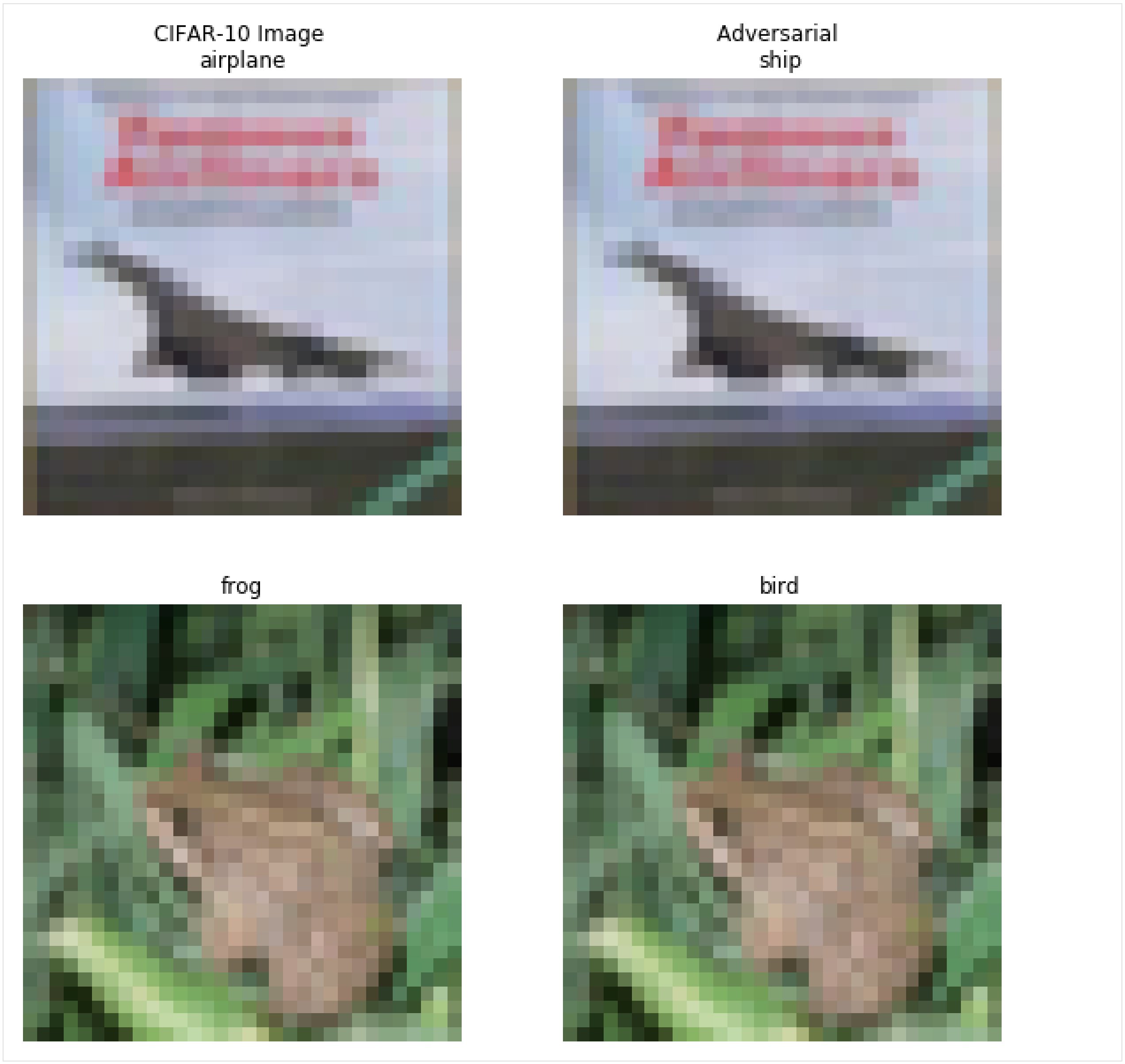
Let's visualise some adversarial instances:

[11]:
```
idx = [3, 4]
print('C&W attack...')
plot_adversarial(idx, X_test, y_pred, X_test_cw, y_pred_cw,
                 mean_test, std_test, figsize=(10, 10))
print('SLIDE attack...')
plot_adversarial(idx, X_test, y_pred, X_test_slide, y_pred_slide,
                 mean_test, std_test, figsize=(10, 10))
```

```
C&W attack...
```



```
SLIDE attack...
```



37.6 Load or train and evaluate the adversarial detectors

We can again either fetch the pretrained detector from a Google Cloud Bucket or train one from scratch:

[12]:
```
load_pretrained = True
```

[13]:
```
filepath = 'my_path'  # change to (absolute) directory where model is downloaded
if load_pretrained:
    detector_type = 'adversarial'
    detector_name = 'base'
    ad = fetch_detector(filepath, detector_type, dataset, detector_name, model=model)
    filepath = os.path.join(filepath, detector_name)
else:  # train detector from scratch
```

(continues on next page)

(continued from previous page)

```
    # define encoder and decoder networks
    encoder_net = tf.keras.Sequential(
            [
                InputLayer(input_shape=(32, 32, 3)),
                Conv2D(32, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2D(64, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2D(256, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Flatten(),
                Dense(40)
            ]
        )

    decoder_net = tf.keras.Sequential(
        [
                InputLayer(input_shape=(40,)),
                Dense(4 * 4 * 128, activation=tf.nn.relu),
                Reshape(target_shape=(4, 4, 128)),
                Conv2DTranspose(256, 4, strides=2, padding='same',
                                activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2DTranspose(64, 4, strides=2, padding='same',
                                activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2DTranspose(3, 4, strides=2, padding='same',
                                activation=None, kernel_regularizer=l1(1e-5))
            ]
        )

    # initialise and train detector
    ad = AdversarialAE(
        encoder_net=encoder_net,
        decoder_net=decoder_net,
        model=clf
    )
    ad.fit(X_train, epochs=40, batch_size=64, verbose=True)

    # save the trained adversarial detector
    save_detector(ad, filepath)
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
WARNING:alibi_detect.ad.adversarialae:No threshold level set. Need to infer threshold␣
↪using `infer_threshold`.
```

The detector first reconstructs the input instances which can be adversarial. The reconstructed input is then fed to the classifier if the adversarial score for the instance is above the threshold. Let's investigate what happens when we reconstruct attacked instances and make predictions on them:

```
[14]: X_recon_cw = predict_batch(ad.ae, X_test_cw, batch_size=32)
      X_recon_slide = predict_batch(ad.ae, X_test_slide, batch_size=32)
```

```
[15]: y_recon_cw = predict_batch(clf, X_recon_cw, batch_size=32, return_class=True)
      y_recon_slide = predict_batch(clf, X_recon_slide, batch_size=32, return_class=True)
```

Accuracy on attacked vs. reconstructed instances:

```
[16]: acc_y_recon_cw = accuracy(y_test, y_recon_cw)
      acc_y_recon_slide = accuracy(y_test, y_recon_slide)
      print('Accuracy after C&W attack {:.4f} -- reconstruction {:.4f}'.format(acc_y_pred_
       ↪cw, acc_y_recon_cw))
      print('Accuracy after SLIDE attack {:.4f} -- reconstruction {:.4f}'.format(acc_y_pred_
       ↪slide, acc_y_recon_slide))
```

```
Accuracy after C&W attack 0.0000 -- reconstruction 0.8048
Accuracy after SLIDE attack 0.0002 -- reconstruction 0.8159
```

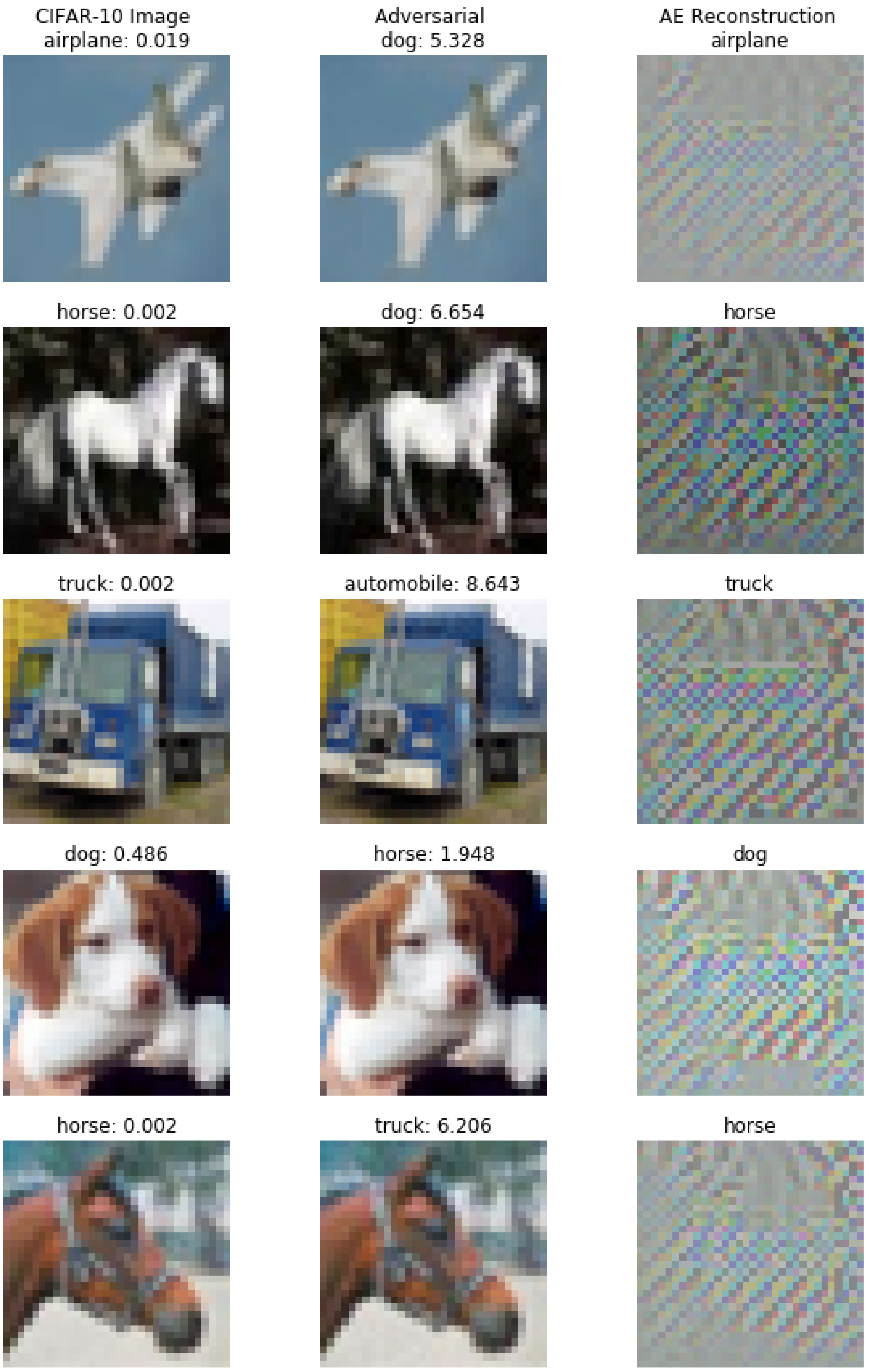
The detector restores the accuracy after the attacks from almost 0% to well over 80%! We can compute the adversarial scores and inspect some of the reconstructed instances:

```
[17]: score_x = ad.score(X_test, batch_size=32)
      score_cw = ad.score(X_test_cw, batch_size=32)
      score_slide = ad.score(X_test_slide, batch_size=32)
```

```
[18]: print('C&W attack...')
      idx = [10, 13, 14, 16, 17]
      plot_adversarial(idx, X_test, y_pred, X_test_cw, y_pred_cw, mean_test, std_test,
                       score_x=score_x, score_x_adv=score_cw, X_recon=X_recon_cw,
                       y_recon=y_recon_cw, figsize=(10, 15))
      print('SLIDE attack...')
      idx = [23, 25, 27, 29, 34]
      plot_adversarial(idx, X_test, y_pred, X_test_slide, y_pred_slide, mean_test, std_test,
                       score_x=score_x, score_x_adv=score_slide, X_recon=X_recon_slide,
                       y_recon=y_recon_slide, figsize=(10, 15))
```

```
C&W attack...
```

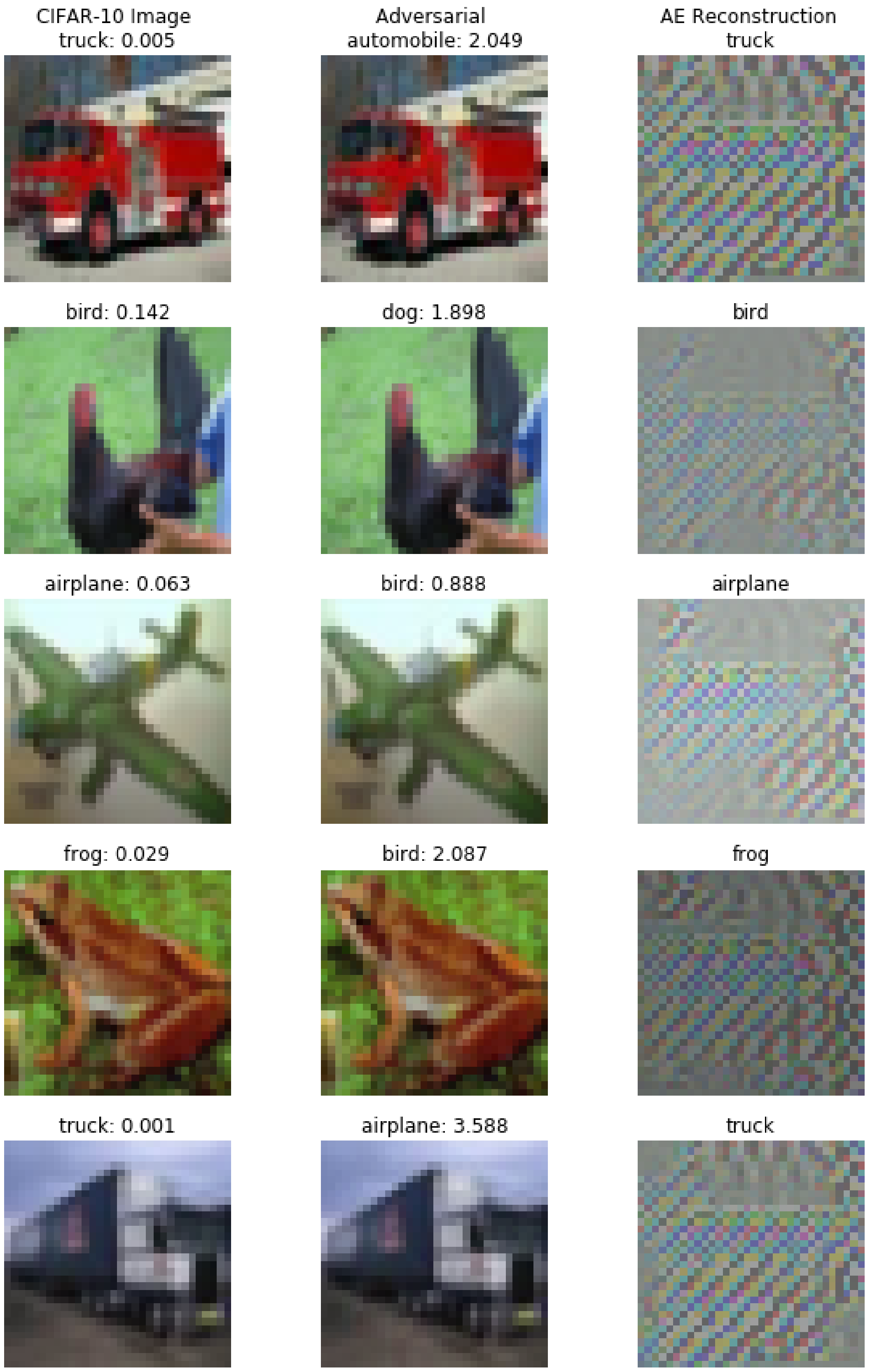
```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB␣
 ↪data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB␣
 ↪data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB␣
 ↪data ([0..1] for floats or [0..255] for integers).
```


CIFAR-10 Image
airplane: 0.019
Adversarial
dog: 5.328
AE Reconstruction
airplane
horse: 0.002
dog: 6.654
horse
truck: 0.002
automobile: 8.643
truck
dog: 0.486
horse: 1.948
dog
horse: 0.002
truck: 6.206
horse

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB␣
↪data ([0..1] for floats or [0..255] for integers).
```

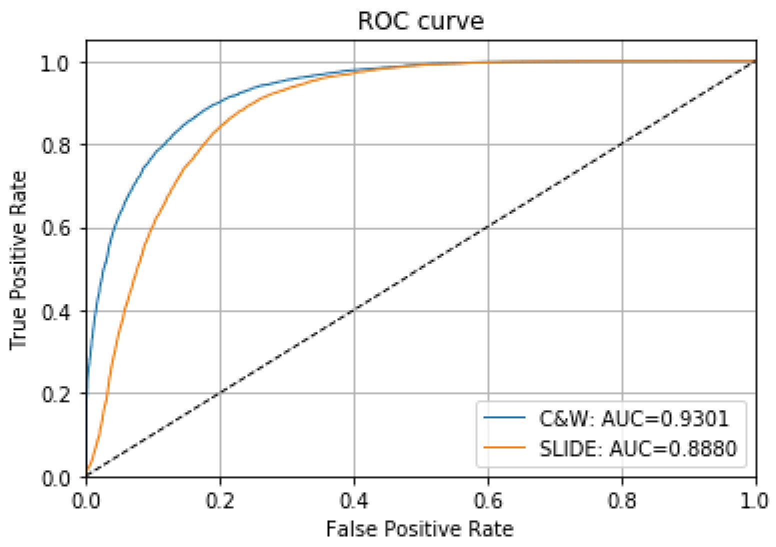
```
SLIDE attack...
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB␣
↪data ([0..1] for floats or [0..255] for integers).
```


CIFAR-10 Image
truck: 0.005
Adversarial
automobile: 2.049
AE Reconstruction
truck
bird: 0.142
dog: 1.898
bird
airplane: 0.063
bird: 0.888
airplane
frog: 0.029
bird: 2.087
frog
truck: 0.001
airplane: 3.588
truck

The ROC curves and AUC values show the effectiveness of the adversarial score to detect adversarial instances:

```
[19]: roc_data = {
          'original': {'scores': score_x, 'predictions': y_pred},
          'C&W': {'scores': score_cw, 'predictions': y_pred_cw, 'normal': 'original'},
          'SLIDE': {'scores': score_slide, 'predictions': y_pred_slide, 'normal': 'original
      ↪'}
      }

      plot_roc(roc_data)
```



The threshold for the adversarial score can be set via `infer_threshold`. We need to pass a batch of instances X and specify what percentage of those we consider to be normal via `threshold_perc`. Assume we have only normal instances some of which the model has misclassified leading to a higher score if the reconstruction picked up features from the correct class or some might look adversarial in the first place. As a result, we set our threshold at 95%:

```
[20]: ad.infer_threshold(X_test, threshold_perc=95, margin=0., batch_size=32)
      print('Adversarial threshold: {:.4f}'.format(ad.threshold))
```

```
Adversarial threshold: 2.6722
```

Let's save the updated detector:

```
[21]: save_detector(ad, filepath)
```

We can also load it easily as follows:

```
[22]: ad = load_detector(filepath)
```

The `correct` method of the detector executes the diagram in Figure 1. First the adversarial scores is computed. For instances where the score is above the threshold, the classifier prediction on the reconstructed instance is returned. Otherwise the original prediction is kept. The method returns a dictionary containing the metadata of the detector, whether the instances in the batch are adversarial (above the threshold) or not, the classifier predictions using the correction mechanism and both the original and reconstructed predictions. Let's illustrate this on a batch containing some adversarial (C&W) and original test set instances:

```
[23]: n_test = X_test.shape[0]
      np.random.seed(0)
      idx_normal = np.random.choice(n_test, size=1600, replace=False)
      idx_cw = np.random.choice(n_test, size=400, replace=False)

      X_mix = np.concatenate([X_test[idx_normal], X_test_cw[idx_cw]])
      y_mix = np.concatenate([y_test[idx_normal], y_test[idx_cw]])
      print(X_mix.shape, y_mix.shape)
```

```
(2000, 32, 32, 3) (2000,)
```

Let's check the model performance:

```
[24]: y_pred_mix = predict_batch(clf, X_mix, batch_size=32, return_class=True)
      acc_y_pred_mix = accuracy(y_mix, y_pred_mix)
      print('Accuracy {:.4f}'.format(acc_y_pred_mix))
```

```
Accuracy 0.7380
```

This can be improved with the correction mechanism:

```
[25]: preds = ad.correct(X_mix, batch_size=32)
      acc_y_corr_mix = accuracy(y_mix, preds['data']['corrected'])
      print('Accuracy {:.4f}'.format(acc_y_corr_mix))
```

```
Accuracy 0.8205
```

37.7 Temperature Scaling

We can further improve the correction performance by applying temperature scaling on the original model predictions $M(x)$ during both training and inference when computing the adversarial scores. We can again load a pretrained detector or train one from scratch:

```
[26]: load_pretrained = True
```

```
[27]: filepath = 'my_path'  # change to (absolute) directory where model is downloaded
      if load_pretrained:
          detector_name = 'temperature'
          ad_t = fetch_detector(filepath, detector_type, dataset, detector_name,␣
      ↪model=model)
          filepath = os.path.join(filepath, detector_name)
      else:  # train detector from scratch
          # define encoder and decoder networks
          encoder_net = tf.keras.Sequential(
                  [
                      InputLayer(input_shape=(32, 32, 3)),
                      Conv2D(32, 4, strides=2, padding='same',
                             activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                      Conv2D(64, 4, strides=2, padding='same',
                             activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                      Conv2D(256, 4, strides=2, padding='same',
                             activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                      Flatten(),
                      Dense(40)
                  ]
```

(continues on next page)

(continued from previous page)

```
        )

    decoder_net = tf.keras.Sequential(
        [
                InputLayer(input_shape=(40,)),
                Dense(4 * 4 * 128, activation=tf.nn.relu),
                Reshape(target_shape=(4, 4, 128)),
                Conv2DTranspose(256, 4, strides=2, padding='same',
                                activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2DTranspose(64, 4, strides=2, padding='same',
                                activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2DTranspose(3, 4, strides=2, padding='same',
                                activation=None, kernel_regularizer=l1(1e-5))
            ]
        )

    # initialise and train detector
    ad_t = AdversarialAE(
        encoder_net=encoder_net,
        decoder_net=decoder_net,
        model=clf,
        temperature=0.5
    )
    ad_t.fit(X_train, epochs=40, batch_size=64, verbose=True)

    # save the trained adversarial detector
    save_detector(ad_t, filepath)
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
WARNING:alibi_detect.ad.adversarialae:No threshold level set. Need to infer threshold␣
↪using `infer_threshold`.
```

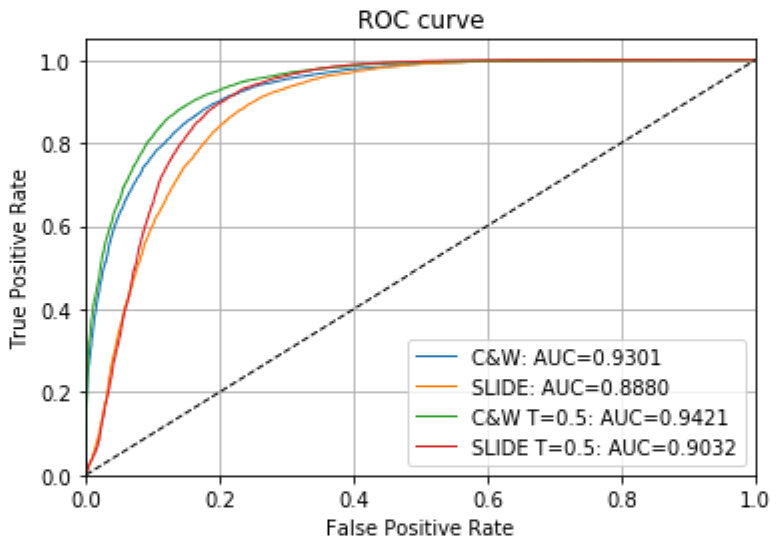
[28]:
```
# reconstructed adversarial instances
X_recon_cw_t = predict_batch(ad_t.ae, X_test_cw, batch_size=32)
X_recon_slide_t = predict_batch(ad_t.ae, X_test_slide, batch_size=32)

# make predictions on reconstructed instances and compute accuracy
y_recon_cw_t = predict_batch(clf, X_recon_cw_t, batch_size=32, return_class=True)
y_recon_slide_t = predict_batch(clf, X_recon_slide_t, batch_size=32, return_
↪class=True)
acc_y_recon_cw_t = accuracy(y_test, y_recon_cw_t)
acc_y_recon_slide_t = accuracy(y_test, y_recon_slide_t)
print('Accuracy after C&W attack {:.4f} -- reconstruction {:.4f}'.format(acc_y_pred_
↪cw, acc_y_recon_cw_t))
print('Accuracy after SLIDE attack {:.4f} -- reconstruction {:.4f}'.format(acc_y_pred_
↪slide,
                                                                          acc_y_
↪recon_slide_t))
```

```
Accuracy after C&W attack 0.0000 -- reconstruction 0.8141
Accuracy after SLIDE attack 0.0002 -- reconstruction 0.8265
```

Applying temperature scaling to CIFAR-10 improves the ROC curve and AUC values.

[29]:
```
score_x_t = ad_t.score(X_test, batch_size=32)
score_cw_t = ad_t.score(X_test_cw, batch_size=32)
score_slide_t = ad_t.score(X_test_slide, batch_size=32)
```

[30]:
```
roc_data['original_t'] = {'scores': score_x_t, 'predictions': y_pred}
roc_data['C&W T=0.5'] = {'scores': score_cw_t, 'predictions': y_pred_cw, 'normal':
↪'original_t'}
roc_data['SLIDE T=0.5'] = {'scores': score_slide_t, 'predictions': y_pred_slide,
↪'normal': 'original_t'}

plot_roc(roc_data)
```



37.8 Hidden Layer K-L Divergence

The performance of the correction mechanism can also be improved by extending the training methodology to one of the hidden layers of the classification model. We extract a flattened feature map from the hidden layer, feed it into a linear layer and apply the softmax function. The K-L divergence between predictions on the hidden layer for x and x' is optimised and included in the adversarial score during inference:

[31]:
```
load_pretrained = True
```

[32]:
```
filepath = 'my_path'  # change to (absolute) directory where model is downloaded
if load_pretrained:
    detector_name = 'hiddenkld'
    ad_hl = fetch_detector(filepath, detector_type, dataset, detector_name,␣
↪model=model)
    filepath = os.path.join(filepath, detector_name)
else:  # train detector from scratch
    # define encoder and decoder networks
```

(continues on next page)

(continued from previous page)

```
    encoder_net = tf.keras.Sequential(
            [
                InputLayer(input_shape=(32, 32, 3)),
                Conv2D(32, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2D(64, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2D(256, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Flatten(),
                Dense(40)
            ]
        )

    decoder_net = tf.keras.Sequential(
        [
                InputLayer(input_shape=(40,)),
                Dense(4 * 4 * 128, activation=tf.nn.relu),
                Reshape(target_shape=(4, 4, 128)),
                Conv2DTranspose(256, 4, strides=2, padding='same',
                                activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2DTranspose(64, 4, strides=2, padding='same',
                                activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2DTranspose(3, 4, strides=2, padding='same',
                                activation=None, kernel_regularizer=l1(1e-5))
            ]
        )

    # initialise and train detector
    ad_hl = AdversarialAE(
        encoder_net=encoder_net,
        decoder_net=decoder_net,
        model=clf,
        hidden_layer_kld={200: 20},  # extract feature map from hidden layer 200
        temperature=1                 # predict softmax with output dim=20
    )
    ad_hl.fit(X_train, epochs=40, batch_size=64, verbose=True)

    # save the trained adversarial detector
    save_detector(ad_hl, filepath)
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
WARNING:alibi_detect.ad.adversarialae:No threshold level set. Need to infer threshold␣
↪using `infer_threshold`.
```

```
[33]: # reconstructed adversarial instances
X_recon_cw_hl = predict_batch(ad_hl.ae, X_test_cw, batch_size=32)
X_recon_slide_hl = predict_batch(ad_hl.ae, X_test_slide, batch_size=32)
```

(continues on next page)

(continued from previous page)

```
# make predictions on reconstructed instances and compute accuracy
y_recon_cw_hl = predict_batch(clf, X_recon_cw_hl, batch_size=32, return_class=True)
y_recon_slide_hl = predict_batch(clf, X_recon_slide_hl, batch_size=32, return_
↪class=True)
acc_y_recon_cw_hl = accuracy(y_test, y_recon_cw_hl)
acc_y_recon_slide_hl = accuracy(y_test, y_recon_slide_hl)
print('Accuracy after C&W attack {:.4f} -- reconstruction {:.4f}'.format(acc_y_pred_
↪cw, acc_y_recon_cw_hl))
print('Accuracy after SLIDE attack {:.4f} -- reconstruction {:.4f}'.format(acc_y_pred_
↪slide,
                                                                           acc_y_
↪recon_slide_hl))
```

```
Accuracy after C&W attack 0.0000 -- reconstruction 0.8153
Accuracy after SLIDE attack 0.0002 -- reconstruction 0.8318
```

37.9 Malicious Data Drift

The adversarial detector proves to be very flexible and can be used to measure the harmfulness of the data drift on the classifier. We evaluate the detector on the CIFAR-10-C dataset (Hendrycks & Dietterich, 2019). The instances in CIFAR-10-C have been corrupted and perturbed by various types of noise, blur, brightness etc. at different levels of severity, leading to a gradual decline in model performance.

We can select from the following corruption types:

[34]:
```
corruptions = corruption_types_cifar10c()
print(corruptions)
```

```
['brightness', 'contrast', 'defocus_blur', 'elastic_transform', 'fog', 'frost',
↪'gaussian_blur', 'gaussian_noise', 'glass_blur', 'impulse_noise', 'jpeg_compression
↪', 'motion_blur', 'pixelate', 'saturate', 'shot_noise', 'snow', 'spatter', 'speckle_
↪noise', 'zoom_blur']
```

Fetch the CIFAR-10-C data for a list of corruptions at each severity level (from 1 to 5), make classifier predictions on the corrupted data, compute adversarial scores and identify which perturbations where malicious or harmful and which weren't. We can then store and visualise the adversarial scores for the harmful and harmless corruption. The score for the harmful perturbations is significantly higher than for the harmless ones. As a result, the adversarial detector also functions as a data drift detector.

[35]:
```
severities = [1,2,3,4,5]

score_drift = {
    1: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
    2: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
    3: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
    4: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
    5: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
}

for s in severities:
    print('\nSeverity: {} of {}'.format(s, len(severities)))

    print('Loading corrupted dataset...')
    X_corr, y_corr = fetch_cifar10c(corruption=corruptions, severity=s, return_X_
↪y=True)
```

(continues on next page)

(continued from previous page)

```
    X_corr = X_corr.astype('float32')

    print('Preprocess data...')
    X_corr, mean_test, std_test = scale_by_instance(X_corr)

    print('Make predictions on corrupted dataset...')
    y_pred_corr = predict_batch(clf, X_corr, batch_size=32, return_class=True)

    print('Compute adversarial scores on corrupted dataset...')
    score_corr = ad_t.score(X_corr, batch_size=32)
    scores = np.concatenate([score_x_t, score_corr])

    print('Get labels for malicious corruptions...')
    labels_corr = np.zeros(score_corr.shape[0])
    repeat = y_corr.shape[0] // y_test.shape[0]
    y_pred_repeat = np.tile(y_pred, (repeat,))
    # malicious/harmful corruption: original prediction correct but
    # prediction on corrupted data incorrect
    idx_orig_right = np.where(y_pred_repeat == y_corr)[0]
    idx_corr_wrong = np.where(y_pred_corr != y_corr)[0]
    idx_harmful = np.intersect1d(idx_orig_right, idx_corr_wrong)
    labels_corr[idx_harmful] = 1
    labels = np.concatenate([np.zeros(X_test.shape[0]), labels_corr]).astype(int)
    # harmless corruption: original prediction correct and prediction
    # on corrupted data correct
    idx_corr_right = np.where(y_pred_corr == y_corr)[0]
    idx_harmless = np.intersect1d(idx_orig_right, idx_corr_right)

    score_drift[s]['all'] = score_corr
    score_drift[s]['harm'] = score_corr[idx_harmful]
    score_drift[s]['noharm'] = score_corr[idx_harmless]
    score_drift[s]['acc'] = accuracy(y_corr, y_pred_corr)
```

```
Severity: 1 of 5
Loading corrupted dataset...
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Get labels for malicious corruptions...

Severity: 2 of 5
Loading corrupted dataset...
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Get labels for malicious corruptions...

Severity: 3 of 5
Loading corrupted dataset...
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Get labels for malicious corruptions...

Severity: 4 of 5
Loading corrupted dataset...
```

(continues on next page)

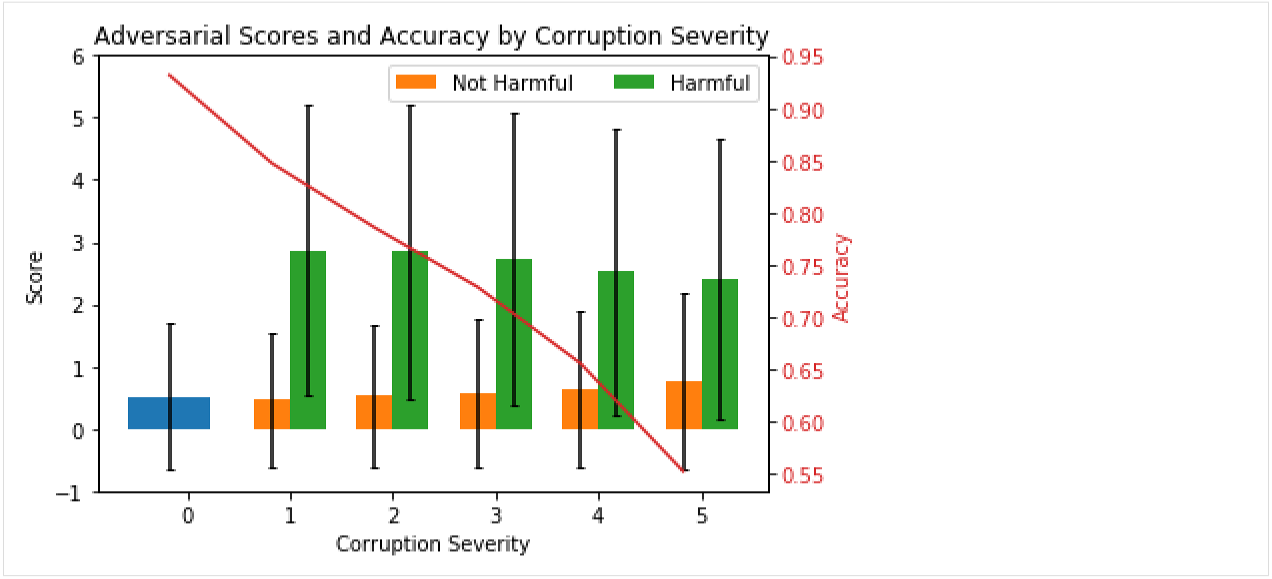
(continued from previous page)

```
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Get labels for malicious corruptions...

Severity: 5 of 5
Loading corrupted dataset...
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Get labels for malicious corruptions...
```

Compute mean scores and standard deviation per severity level and plot:

[36]:
```
mu_noharm, std_noharm = [], []
mu_harm, std_harm = [], []
acc = [acc_y_pred]
for k, v in score_drift.items():
    mu_noharm.append(v['noharm'].mean())
    std_noharm.append(v['noharm'].std())
    mu_harm.append(v['harm'].mean())
    std_harm.append(v['harm'].std())
    acc.append(v['acc'])
```

[37]:
```
plot_labels = ['0', '1', '2', '3', '4', '5']

N = 6
ind = np.arange(N)
width = .35

fig_bar_cd, ax = plt.subplots()
ax2 = ax.twinx()

p0 = ax.bar(ind[0], score_x_t.mean(), yerr=score_x_t.std(), capsize=2)
p1 = ax.bar(ind[1:], mu_noharm, width, yerr=std_noharm, capsize=2)
p2 = ax.bar(ind[1:] + width, mu_harm, width, yerr=std_harm, capsize=2)

ax.set_title('Adversarial Scores and Accuracy by Corruption Severity')
ax.set_xticks(ind + width / 2)
ax.set_xticklabels(plot_labels)
ax.set_ylim((-1,6))
ax.legend((p1[0], p2[0]), ('Not Harmful', 'Harmful'), loc='upper right', ncol=2)
ax.set_ylabel('Score')
ax.set_xlabel('Corruption Severity')

color = 'tab:red'
ax2.set_ylabel('Accuracy', color=color)
ax2.plot(acc, color=color)
ax2.tick_params(axis='y', labelcolor=color)

plt.show()
```


Adversarial Scores and Accuracy by Corruption Severity
Not Harmful
Harmful
Score
Accuracy
Corruption Severity
6
5
4
3
2
1
0
−1
0
1
2
3
4
5
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55

CHAPTER

THIRTYEIGHT

MODEL DISTILLATION DRIFT DETECTOR ON CIFAR-10

38.1 Method

Model distillation is a technique that is used to transfer knowledge from a large network to a smaller network. Typically, it consists of training a second model with a simplified architecture on soft targets (the output distributions or the logits) obtained from the original model.

Here, we apply model distillation to obtain harmfulness scores, by comparing the output distributions of the original model with the output distributions of the distilled model, in order to detect adversarial data, malicious data drift or data corruption. We use the following definition of harmful and harmless data points:

- Harmful data points are defined as inputs for which the model's predictions on the uncorrupted data are correct while the model's predictions on the corrupted data are wrong.
- Harmless data points are defined as inputs for which the model's predictions on the uncorrupted data are correct and the model's predictions on the corrupted data remain correct.

Analogously to the adversarial AE detector, which is also part of the library, the model distillation detector picks up drift that reduces the performance of the classification model.

Moreover, in this example a drift detector that applies two-sample Kolmogorov-Smirnov (K-S) tests to the scores is employed. The p-values obtained are used to assess the harmfulness of the data.

38.2 Dataset

CIFAR10 consists of 60,000 32 by 32 RGB images equally distributed over 10 classes. We evaluate the drift detector on the CIFAR-10-C dataset (Hendrycks & Dietterich, 2019). The instances in CIFAR-10-C have been corrupted and perturbed by various types of noise, blur, brightness etc. at different levels of severity, leading to a gradual decline in the classification model performance.

[2]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import os
import tensorflow as tf
from alibi_detect.cd import KSDrift
from alibi_detect.ad import ModelDistillation

from alibi_detect.models.tensorflow.resnet import scale_by_instance
from alibi_detect.utils.fetching import fetch_tf_model, fetch_detector
from alibi_detect.utils.prediction import predict_batch
from alibi_detect.utils.saving import save_detector
from alibi_detect.datasets import fetch_cifar10c, corruption_types_cifar10c
```

```
Importing plotly failed. Interactive plots will not work.
```

38.3 Load data

Original CIFAR-10 data:

```
[5]: (X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
X_train = X_train.astype('float32') / 255
X_train = scale_by_instance(X_train)
y_train = y_train.astype('int64').reshape(-1,)
X_test = X_test.astype('float32') / 255
y_test = y_test.astype('int64').reshape(-1,)
```

For CIFAR-10-C, we can select from the following corruption types at 5 severity levels:

```
[6]: corruptions = corruption_types_cifar10c()
print(corruptions)
```

```
['brightness', 'contrast', 'defocus_blur', 'elastic_transform', 'fog', 'frost',
↪'gaussian_blur', 'gaussian_noise', 'glass_blur', 'impulse_noise', 'jpeg_compression
↪', 'motion_blur', 'pixelate', 'saturate', 'shot_noise', 'snow', 'spatter', 'speckle_
↪noise', 'zoom_blur']
```

Let's pick a subset of the corruptions at corruption level 5. Each corruption type consists of perturbations on all of the original test set images.

```
[7]: corruption = ['gaussian_noise', 'motion_blur', 'brightness', 'pixelate']
X_corr, y_corr = fetch_cifar10c(corruption=corruption, severity=5, return_X_y=True)
X_corr = X_corr.astype('float32') / 255
```

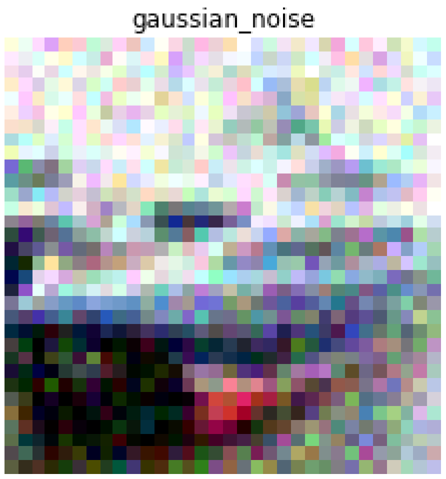
We split the corrupted data by corruption type:

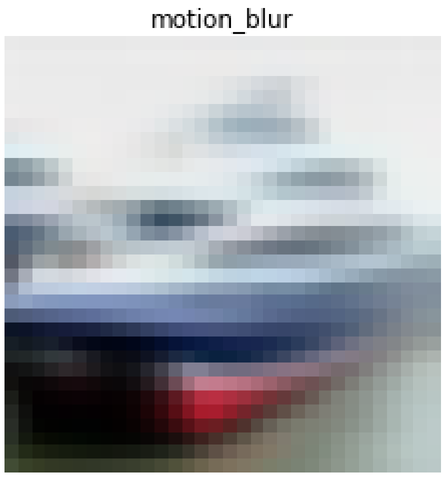
```
[8]: X_c = []
n_corr = len(corruption)
n_test = X_test.shape[0]
for i in range(n_corr):
    X_c.append(X_corr[i * n_test:(i + 1) * n_test])
```

We can visualise the same instance for each corruption type:

```
[9]: i = 1

n_test = X_test.shape[0]
plt.title('Original')
plt.axis('off')
plt.imshow(X_test[i])
plt.show()
for _ in range(len(corruption)):
    plt.title(corruption[_])
    plt.axis('off')
    plt.imshow(X_corr[n_test * _+ i])
    plt.show()
```

Original


gaussian_noise


motion_blur

brightness

pixelate

We can also verify that the performance of a classification model on CIFAR-10 drops significantly on this perturbed dataset:

```
[10]: dataset = 'cifar10'
model = 'resnet32'
clf = fetch_tf_model(dataset, model)
acc = clf.evaluate(scale_by_instance(X_test), y_test, batch_size=128, verbose=0)[1]
print('Test set accuracy:')
print('Original {:.4f}'.format(acc))
clf_accuracy = {'original': acc}
for _ in range(len(corruption)):
    acc = clf.evaluate(scale_by_instance(X_c[_]), y_test, batch_size=128,␣
↪verbose=0)[1]
    clf_accuracy[corruption[_]] = acc
    print('{} {:.4f}'.format(corruption[_], acc))
```

```
Test set accuracy:
Original 0.9278
gaussian_noise 0.2208
motion_blur 0.6339
brightness 0.8913
pixelate 0.3666
```

38.4 Model distillation as a malicious drift detector

Analogously to the adversarial AE detector, which uses an autoencoder to reproduce the output distribution of a classifier and produce adversarial scores, the model distillation detector achieves the same goal by using a simple classifier in place of the autoencoder. This approach is more flexible since it bypasses the instance's generation step, and it can be applied in a straightforward way to a variety of data sets such as text or time series.

We can use the adversarial scores produced by the Model Distillation detector in the context of drift detection. The score function of the detector becomes the preprocessing function for the drift detector. The K-S test is then a simple univariate test between the adversarial scores of the reference batch and the test data. Higher adversarial scores indicate more harmful drift. Importantly, a harmfulness detector flags **malicious data drift**. We can fetch the pretrained model distillation detector from a Google Cloud Bucket or train one from scratch:

Definition and training of the distilled model

[11]:
```
from tensorflow.keras.layers import Conv2D, Dense, Flatten, InputLayer
from tensorflow.keras.regularizers import l1

def distilled_model_cifar10(clf, nb_conv_layers=3, nb_filters1=256, nb_dense=40,
                            kernel1=4, kernel2=4, kernel3=4, ae_arch=False):
    print('Define distilled model')
    nb_filters1 = int(nb_filters1)
    nb_filters2 = int(nb_filters1 / 2)
    nb_filters3 = int(nb_filters1 / 4)
    layers = [InputLayer(input_shape=(32, 32, 3)),
              Conv2D(nb_filters1, kernel1, strides=2, padding='same')]
    if nb_conv_layers > 1:
        layers.append(Conv2D(nb_filters2, kernel2, strides=2, padding='same',
                             activation=tf.nn.relu, kernel_regularizer=l1(1e-5)))
    if nb_conv_layers > 2:
        layers.append(Conv2D(nb_filters3, kernel3, strides=2, padding='same',
                             activation=tf.nn.relu, kernel_regularizer=l1(1e-5)))
    layers.append(Flatten())
    layers.append(Dense(nb_dense))
    layers.append(Dense(clf.output_shape[1], activation='softmax'))
    distilled_model = tf.keras.Sequential(layers)
    return distilled_model
```

[12]:
```
def accuracy(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    return (y_true == y_pred).astype(int).sum() / y_true.shape[0]
```

[13]:
```
load_pretrained = True
```

[14]:
```
filepath = 'my_path' # change to (absolute) directory where model is downloaded
if load_pretrained:
    detector_type = 'adversarial'
    detector_name = 'model_distillation'
    ad = fetch_detector(filepath, detector_type, dataset, detector_name, model=model)
    filepath = os.path.join(filepath, detector_name)
else:
    distilled_model = distilled_model_cifar10(clf)
    print(distilled_model.summary())
    ad = ModelDistillation(distilled_model=distilled_model, model=clf)
    ad.fit(X_train, epochs=50, batch_size=128, verbose=True)
    save_detector(ad, filepath)
```

```
WARNING:tensorflow:No training configuration found in the save file, so the model was␣
↪*not* compiled. Compile it manually.
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your␣
↪model is starting with a freshly initialized optimizer.
```

```
No threshold level set. Need to infer threshold using `infer_threshold`.
```

38.4.1 Scores and p-values calculation

Here we initialize the K-S drift detector using the harmfulness scores as a preprocessing function. The KS test is performed on these scores.

[15]:
```
batch_size = 100
nb_batches = 100
severities = [1, 2, 3, 4, 5]
```

[16]:
```
def sample_batch(x_orig, x_corr, batch_size, p):
    nb_orig = int(batch_size * (1 - p))
    nb_corr = batch_size - nb_orig
    perc = np.round(nb_corr / batch_size, 2)

    idx_orig = np.random.choice(range(x_orig.shape[0]), nb_orig)
    x_sample_orig = x_orig[idx_orig]

    idx_corr = np.random.choice(range(x_corr.shape[0]), nb_corr)
    x_sample_corr = x_corr[idx_corr]

    x_batch = np.concatenate([x_sample_orig, x_sample_corr])
    return x_batch, perc
```

Initialise the drift detector:

[17]:
```
from functools import partial

np.random.seed(0)
n_ref = 1000
idx_ref = np.random.choice(range(X_test.shape[0]), n_ref)
X_test = scale_by_instance(X_test)
X_ref = X_test[idx_ref]
labels = ['No!', 'Yes!']

# adversarial score fn = preprocess step
preprocess_fn = partial(ad.score, batch_size=128)

# initialize the drift detector
cd = KSDrift(X_ref, p_val=.05, preprocess_fn=preprocess_fn)
```

Calculate scores. We split the corrupted data into harmful and harmless data and visualize the harmfulness scores for various values of corruption severity.

[18]:
```
dfs = {}
score_drift = {
    1: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
    2: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
    3: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
```

(continues on next page)

(continued from previous page)

```
    4: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
    5: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
}
y_pred = predict_batch(clf, X_test, batch_size=256, return_class=True)
score_x = ad.score(X_test, batch_size=256)

for s in severities:
    print('Loading corrupted data. Severity = {}'.format(s))
    X_corr, y_corr = fetch_cifar10c(corruption=corruptions, severity=s, return_X_
↪y=True)
    print('Preprocess data...')
    X_corr = X_corr.astype('float32') / 255
    X_corr = scale_by_instance(X_corr)

    print('Make predictions on corrupted dataset...')
    y_pred_corr = predict_batch(clf, X_corr, batch_size=1000, return_class=True)

    print('Compute adversarial scores on corrupted dataset...')
    score_corr = ad.score(X_corr, batch_size=256)

    labels_corr = np.zeros(score_corr.shape[0])
    repeat = y_corr.shape[0] // y_test.shape[0]
    y_pred_repeat = np.tile(y_pred, (repeat,))

    # malicious/harmful corruption: original prediction correct but
    # prediction on corrupted data incorrect
    idx_orig_right = np.where(y_pred_repeat == y_corr)[0]
    idx_corr_wrong = np.where(y_pred_corr != y_corr)[0]
    idx_harmful = np.intersect1d(idx_orig_right, idx_corr_wrong)

    # harmless corruption: original prediction correct and prediction
    # on corrupted data correct
    labels_corr[idx_harmful] = 1
    labels = np.concatenate([np.zeros(X_test.shape[0]), labels_corr]).astype(int)
    idx_corr_right = np.where(y_pred_corr == y_corr)[0]
    idx_harmless = np.intersect1d(idx_orig_right, idx_corr_right)

    # Split corrupted inputs in harmful and harmless
    X_corr_harm = X_corr[idx_harmful]
    X_corr_noharm = X_corr[idx_harmless]

    # Store adversarial scores for harmful and harmless data
    score_drift[s]['all'] = score_corr
    score_drift[s]['harm'] = score_corr[idx_harmful]
    score_drift[s]['noharm'] = score_corr[idx_harmless]
    score_drift[s]['acc'] = accuracy(y_corr, y_pred_corr)

    print('Compute p-values')
    for j in range(nb_batches):
        ps = []
        pvs_harm = []
        pvs_noharm = []
        for p in np.arange(0, 1, 0.1):
            # Sampling a batch of size `batch_size` where a fraction p of the data
            # is corrupted harmful data and a fraction 1 - p is non-corrupted data
            X_batch_harm, _ = sample_batch(X_test, X_corr_harm, batch_size, p)
```

(continues on next page)

(continued from previous page)

```
            # Sampling a batch of size `batch_size` where a fraction p of the data
            # is corrupted harmless data and a fraction 1 - p is non-corrupted data
            X_batch_noharm, perc = sample_batch(X_test, X_corr_noharm, batch_size, p)

            # Calculating p-values for the harmful and harmless data by applying
            # K-S test on the adversarial scores
            pv_harm = cd.score(X_batch_harm)
            pv_noharm = cd.score(X_batch_noharm)
            ps.append(perc * 100)
            pvs_harm.append(pv_harm[0])
            pvs_noharm.append(pv_noharm[0])
        if j == 0:
            df = pd.DataFrame({'p': ps})
        df['pvalue_harm_{}'.format(j)] = pvs_harm
        df['pvalue_noharm_{}'.format(j)] = pvs_noharm

    for name in ['pvalue_harm', 'pvalue_noharm']:
        df[name + '_mean'] = df[[col for col in df.columns if name in col]].
↪mean(axis=1)
        df[name + '_std'] = df[[col for col in df.columns if name in col]].std(axis=1)
        df[name + '_max'] = df[[col for col in df.columns if name in col]].max(axis=1)
        df[name + '_min'] = df[[col for col in df.columns if name in col]].min(axis=1)
    df.set_index('p', inplace=True)
    dfs[s] = df
```

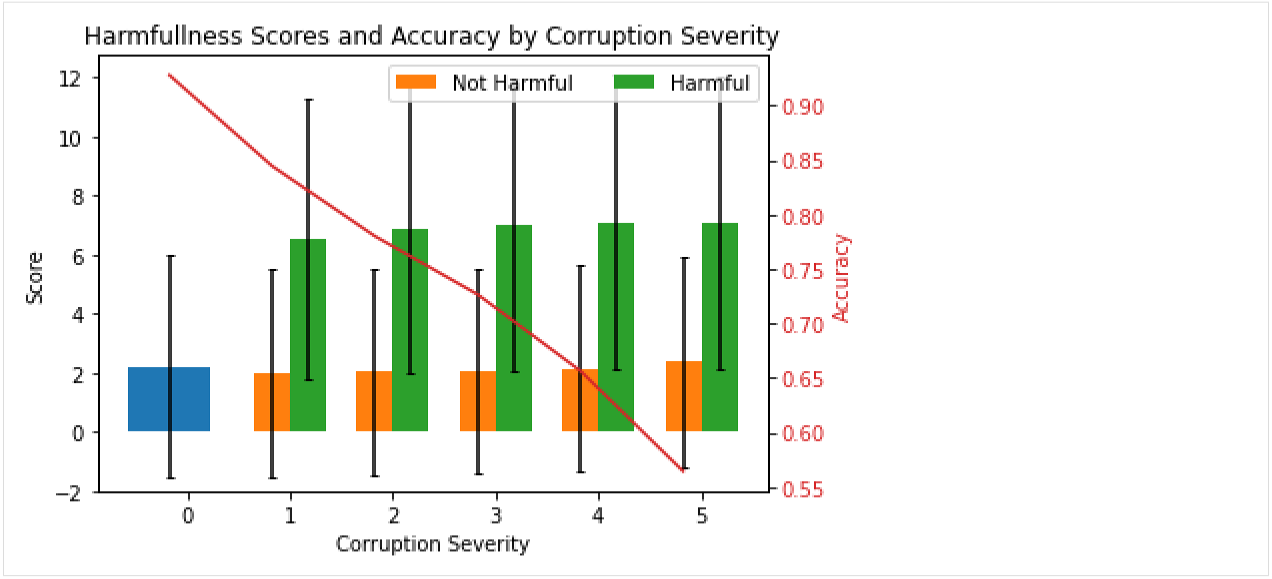
```
Loading corrupted data. Severity = 1
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Compute p-values
Loading corrupted data. Severity = 2
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Compute p-values
Loading corrupted data. Severity = 3
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Compute p-values
Loading corrupted data. Severity = 4
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Compute p-values
Loading corrupted data. Severity = 5
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Compute p-values
```

38.4.2 Plot scores

We now plot the mean scores and standard deviations per severity level. The plot shows the mean harmfulness scores (lhs) and ResNet-32 accuracies (rhs) for increasing data corruption severity levels. Level 0 corresponds to the original test set. Harmful scores are scores from instances which have been flipped from the correct to an incorrect prediction because of the corruption. Not harmful means that a correct prediction was unchanged after the corruption.

[19]:
```python
mu_noharm, std_noharm = [], []
mu_harm, std_harm = [], []
acc = [clf_accuracy['original']]
for k, v in score_drift.items():
    mu_noharm.append(v['noharm'].mean())
    std_noharm.append(v['noharm'].std())
    mu_harm.append(v['harm'].mean())
    std_harm.append(v['harm'].std())
    acc.append(v['acc'])
```

[20]:
```python
plot_labels = ['0', '1', '2', '3', '4', '5']

N = 6
ind = np.arange(N)
width = .35

fig_bar_cd, ax = plt.subplots()
ax2 = ax.twinx()

p0 = ax.bar(ind[0], score_x.mean(), yerr=score_x.std(), capsize=2)
p1 = ax.bar(ind[1:], mu_noharm, width, yerr=std_noharm, capsize=2)
p2 = ax.bar(ind[1:] + width, mu_harm, width, yerr=std_harm, capsize=2)

ax.set_title('Harmfullness Scores and Accuracy by Corruption Severity')
ax.set_xticks(ind + width / 2)
ax.set_xticklabels(plot_labels)
ax.set_ylim((-2))
ax.legend((p1[0], p2[0]), ('Not Harmful', 'Harmful'), loc='upper right', ncol=2)
ax.set_ylabel('Score')
ax.set_xlabel('Corruption Severity')

color = 'tab:red'
ax2.set_ylabel('Accuracy', color=color)
ax2.plot(acc, color=color)
ax2.tick_params(axis='y', labelcolor=color)

plt.show()
```



38.4.3 Plot p-values for contaminated batches

In order to simulate a realistic scenario, we perform a K-S test on batches of instance which are increasingly contaminated with corrupted data. The following steps are implemented:

- We randomly pick `n_ref=1000` samples from the non-currupted test set to be used as a reference set in the initialization of the K-S drift detector.
- We sample batches of data of size `batch_size=100` contaminated with an increasing number of harmful corrupted data and harmless corrupted data.
- The K-S detector predicts whether drift occurs between the contaminated batches and the reference data and returns the p-values of the test.
- We observe that contamination of the batches with harmful data reduces the p-values much faster than contamination with harmless data. In the latter case, the p-values remain above the detection threshold even when the batch is heavily contaminated

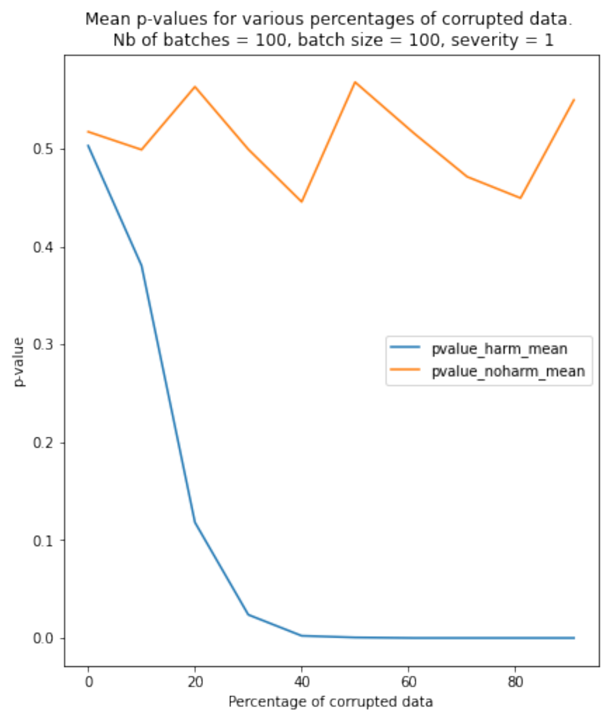
We repeat the test for 100 randomly sampled batches and we plot the mean and the maximum p-values for each level of severity and contamination below. We can see from the plot that the detector is able to clearly detect a batch contaminated with harmful data compared to a batch contaminated with harmless data when the percentage of currupted data reaches 20%-30%.

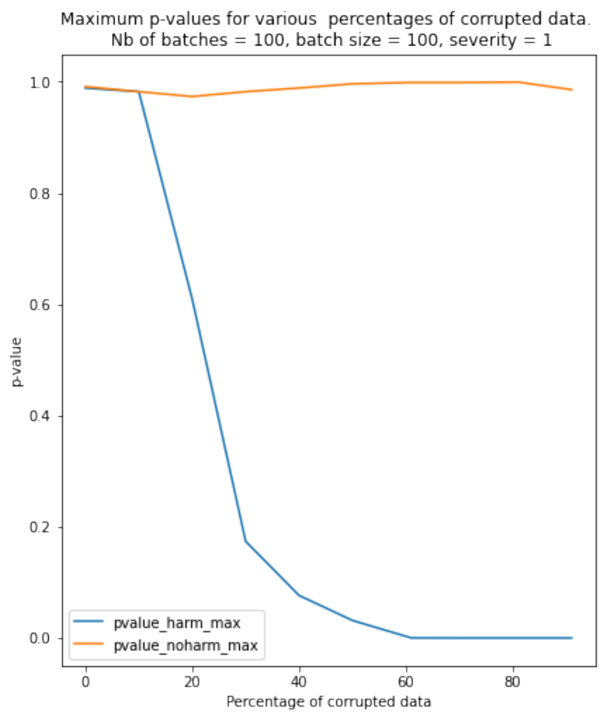
```
[21]: for s in severities:
          nrows = 1
          ncols = 2
          figsize = (15, 8)
          fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)
          title0 = ('Mean p-values for various percentages of corrupted data. \n'
                    ' Nb of batches = {}, batch size = {}, severity = {}'.format(
                        nb_batches, batch_size, s))
          title1 = ('Maximum p-values for various  percentages of corrupted data. \n'
                    ' Nb of batches = {}, batch size = {}, severity = {}'.format(
                        nb_batches, batch_size, s))
          dfs[s][['pvalue_harm_mean', 'pvalue_noharm_mean']].plot(ax=ax[0], title=title0)
          dfs[s][['pvalue_harm_max', 'pvalue_noharm_max']].plot(ax=ax[1], title=title1)
          for a in ax:
```

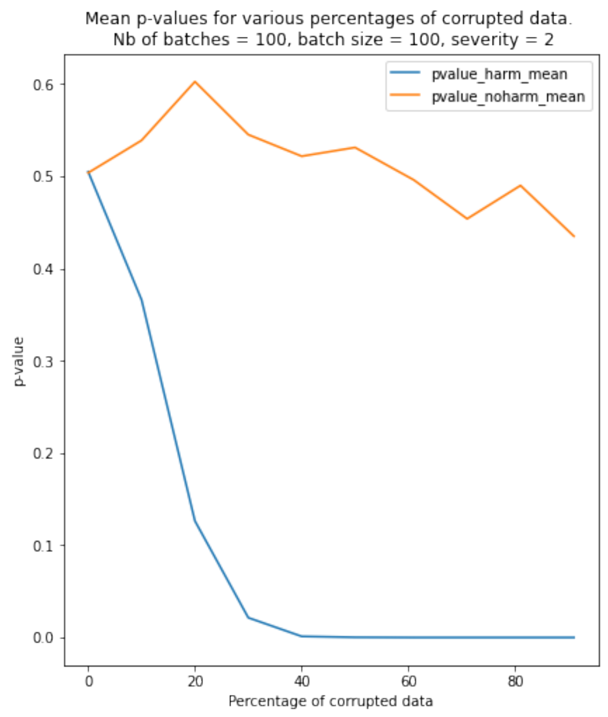
(continues on next page)

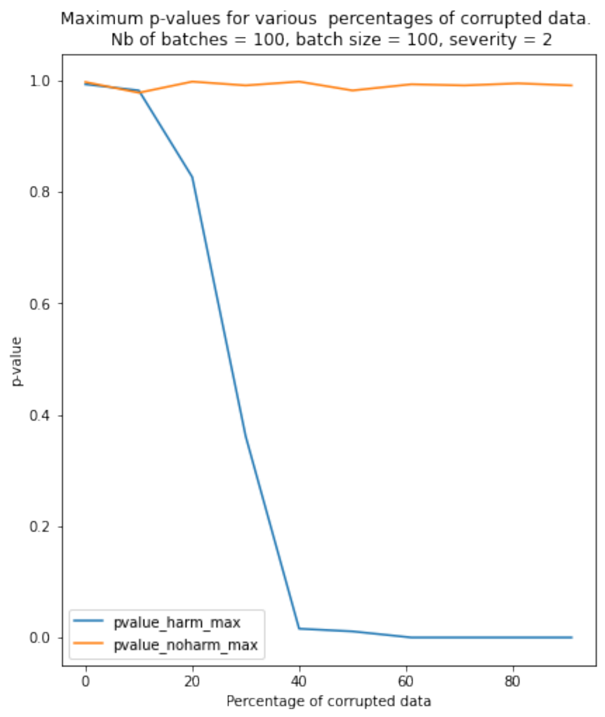
(continued from previous page)

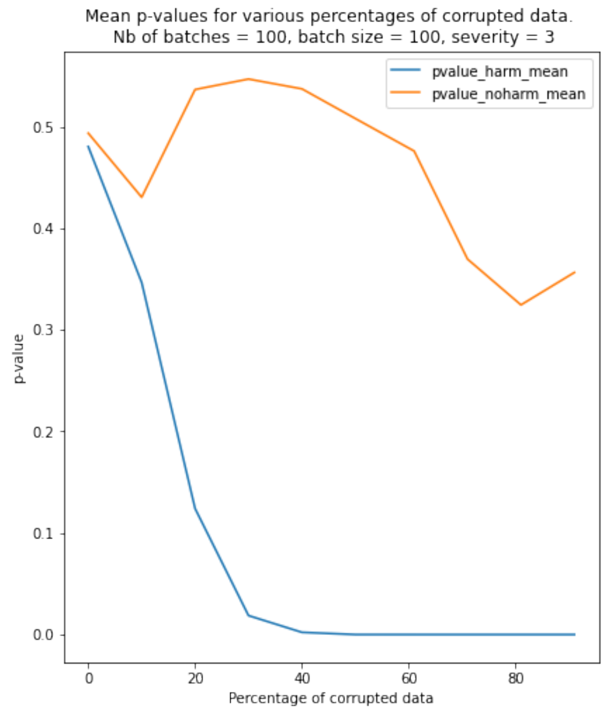
```
        a.set_xlabel('Percentage of corrupted data')
        a.set_ylabel('p-value')
```

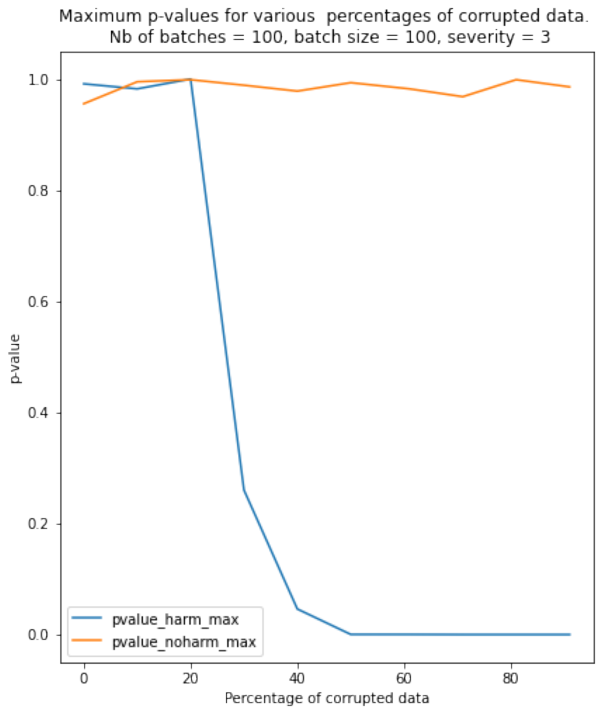


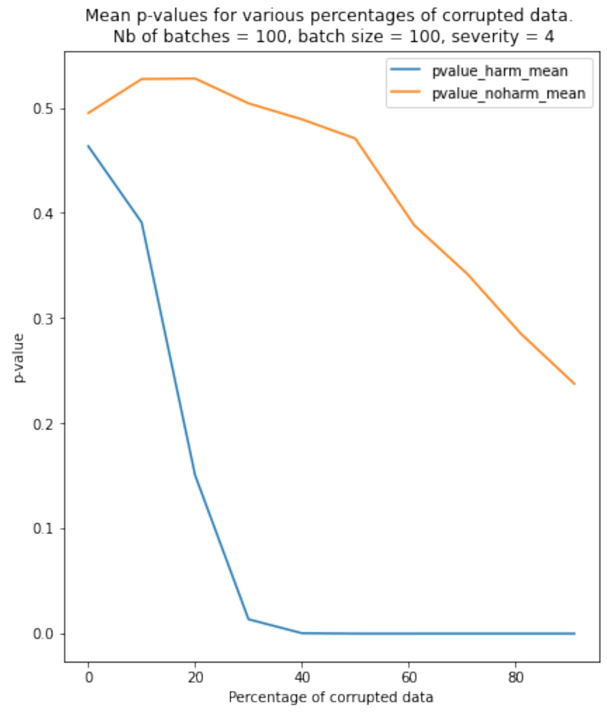


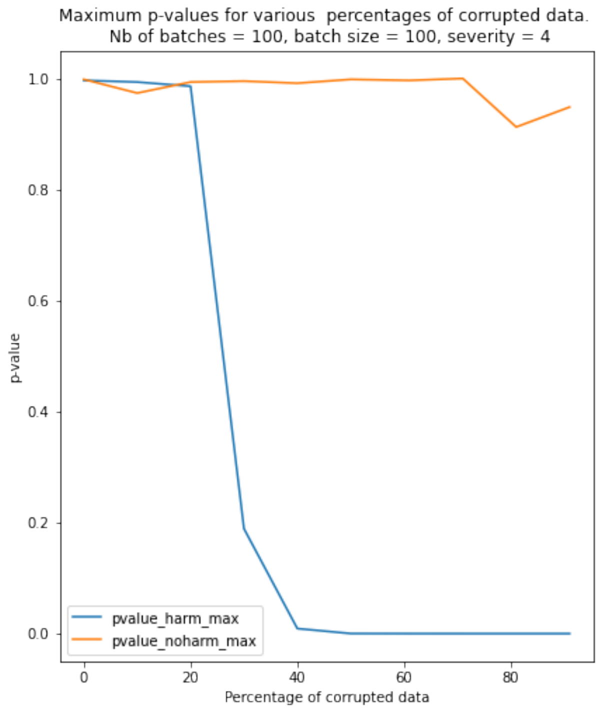


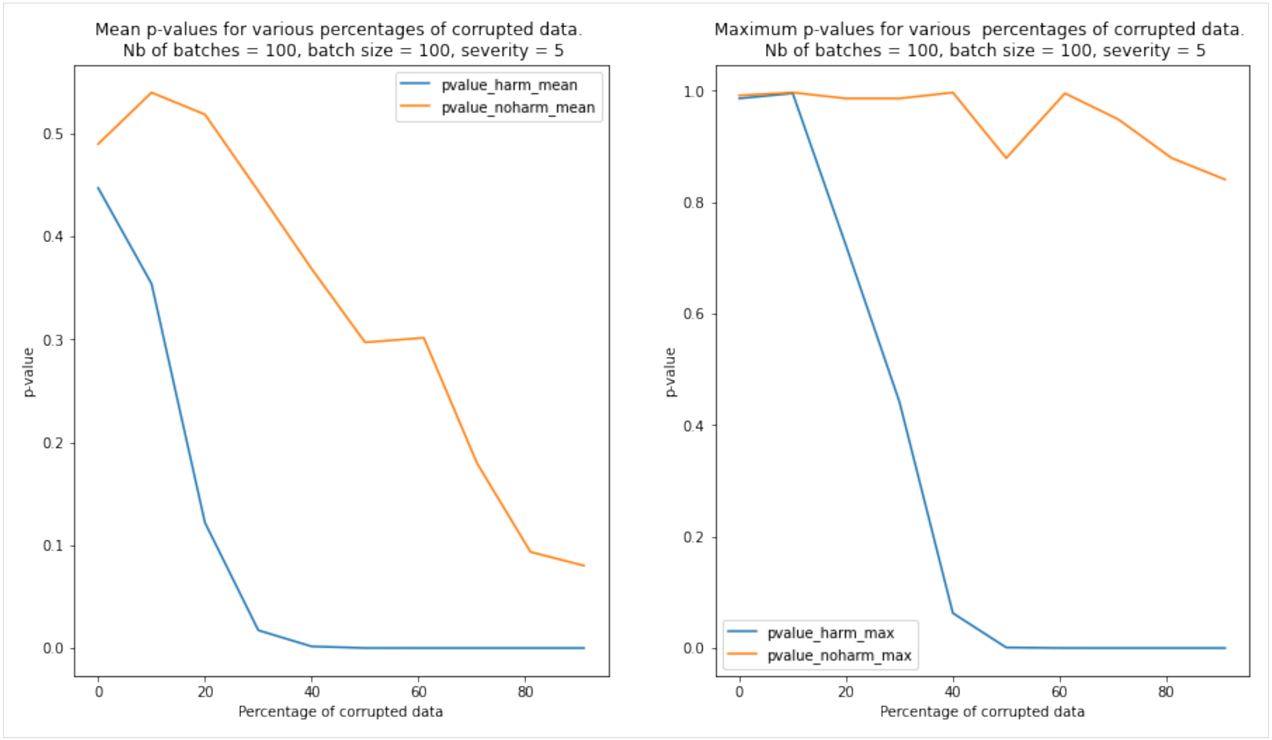



Mean p-values for various percentages of corrupted data.
Nb of batches = 100, batch size = 100, severity = 3
pvalue_harm_mean
pvalue_noharm_mean
p-value
Percentage of corrupted data


Maximum p-values for various percentages of corrupted data.
Nb of batches = 100, batch size = 100, severity = 3
pvalue_harm_max
pvalue_noharm_max
p-value
Percentage of corrupted data


Mean p-values for various percentages of corrupted data.
Nb of batches = 100, batch size = 100, severity = 4
pvalue_harm_mean
pvalue_noharm_mean
p-value
Percentage of corrupted data


Maximum p-values for various percentages of corrupted data.
Nb of batches = 100, batch size = 100, severity = 4
pvalue_harm_max
pvalue_noharm_max
p-value
Percentage of corrupted data


Mean p-values for various percentages of corrupted data.
Nb of batches = 100, batch size = 100, severity = 5
pvalue_harm_mean
pvalue_noharm_mean
p-value
Percentage of corrupted data
Maximum p-values for various percentages of corrupted data.
Nb of batches = 100, batch size = 100, severity = 5
pvalue_harm_max
pvalue_noharm_max
p-value
Percentage of corrupted data

CHAPTER

THIRTYNINE

CATEGORICAL AND MIXED TYPE DATA DRIFT DETECTION ON INCOME PREDICTION

39.1 Method

The drift detector applies feature-wise two-sample Kolmogorov-Smirnov (K-S) tests for the continuous numerical features and Chi-Squared tests for the categorical features. For multivariate data, the obtained p-values for each feature are aggregated either via the Bonferroni or the False Discovery Rate (FDR) correction. The Bonferroni correction is more conservative and controls for the probability of at least one false positive. The FDR correction on the other hand allows for an expected fraction of false positives to occur.

39.2 Dataset

The instances contain a person's characteristics like age, marital status or education while the label represents whether the person makes more or less than $50k per year. The dataset consists of a mixture of numerical and categorical features. It is fetched using the Alibi library, which can be installed with pip:

```
pip install alibi
```

[1]:
```
import alibi
import matplotlib.pyplot as plt
import numpy as np

from alibi_detect.cd import ChiSquareDrift, TabularDrift
from alibi_detect.utils.saving import save_detector, load_detector
```

39.3 Load income prediction dataset

The `fetch_adult` function returns a `Bunch` object containing the instances, the targets, the feature names and a dictionary with as keys the column indices of the categorical features and as values the possible categories for each categorical variable.

[2]:
```
adult = alibi.datasets.fetch_adult()
X, y = adult.data, adult.target
feature_names = adult.feature_names
category_map = adult.category_map
X.shape, y.shape
```

[2]:
```
((32561, 12), (32561,))
```

We split the data in a reference set and 2 test sets on which we test the data drift:

[3]:
```
n_ref = 10000
n_test = 10000

X_ref, X_t0, X_t1 = X[:n_ref], X[n_ref:n_ref + n_test], X[n_ref + n_test:n_ref + 2 *␣
↪n_test]
X_ref.shape, X_t0.shape, X_t1.shape
```

[3]:
```
((10000, 12), (10000, 12), (10000, 12))
```

39.4 Detect drift

We need to provide the drift detector with the columns which contain categorical features so it knows which features require the Chi-Squared and which ones require the K-S univariate test. We can either provide a dict with as keys the column indices and as values the number of possible categories or just set the values to *None* and let the detector infer the number of categories from the reference data as in the example below:

[4]:
```
categories_per_feature = {f: None for f in list(category_map.keys())}
```

Initialize the detector:

[5]:
```
cd = TabularDrift(X_ref, p_val=.05, categories_per_feature=categories_per_feature)
```

We can also save/load an initialised detector:

[6]:
```
filepath = 'my_path'  # change to directory where detector is saved
save_detector(cd, filepath)
cd = load_detector(filepath)
```

```
WARNING:alibi_detect.utils.saving:Directory my_path/model does not exist.
```

Now we can check whether the 2 test sets are drifting from the reference data:

[7]:
```
preds = cd.predict(X_t0)
labels = ['No!', 'Yes!']
print('Drift? {}'.format(labels[preds['data']['is_drift']]))
```

```
Drift? No!
```

Let's take a closer look at each of the features. The `preds` dictionary also returns the K-S or Chi-Squared test statistics and p-value for each feature:

[8]:
```
for f in range(cd.n_features):
    stat = 'Chi2' if f in list(categories_per_feature.keys()) else 'K-S'
    fname = feature_names[f]
    stat_val, p_val = preds['data']['distance'][f], preds['data']['p_val'][f]
    print(f'{fname} -- {stat} {stat_val:.3f} -- p-value {p_val:.3f}')
```

```
Age -- K-S 0.012 -- p-value 0.512
Workclass -- Chi2 8.487 -- p-value 0.387
Education -- Chi2 4.753 -- p-value 0.576
Marital Status -- Chi2 3.160 -- p-value 0.368
Occupation -- Chi2 8.194 -- p-value 0.415
```

(continues on next page)

(continued from previous page)

```
Relationship -- Chi2 0.485 -- p-value 0.993
Race -- Chi2 0.587 -- p-value 0.965
Sex -- Chi2 0.217 -- p-value 0.641
Capital Gain -- K-S 0.002 -- p-value 1.000
Capital Loss -- K-S 0.002 -- p-value 1.000
Hours per week -- K-S 0.012 -- p-value 0.512
Country -- Chi2 9.991 -- p-value 0.441
```

None of the feature-level p-values are below the threshold:

```
[9]: preds['data']['threshold']
```

```
[9]: 0.004166666666666667
```

If you are interested in individual feature-wise drift, this is also possible:

```
[10]: fpreds = cd.predict(X_t0, drift_type='feature')
```

```
[11]: for f in range(cd.n_features):
          stat = 'Chi2' if f in list(categories_per_feature.keys()) else 'K-S'
          fname = feature_names[f]
          is_drift = fpreds['data']['is_drift'][f]
          stat_val, p_val = fpreds['data']['distance'][f], fpreds['data']['p_val'][f]
          print(f'{fname} -- Drift? {labels[is_drift]} -- {stat} {stat_val:.3f} -- p-value
      ↪{p_val:.3f}')
```

```
Age -- Drift? No! -- K-S 0.012 -- p-value 0.512
Workclass -- Drift? No! -- Chi2 8.487 -- p-value 0.387
Education -- Drift? No! -- Chi2 4.753 -- p-value 0.576
Marital Status -- Drift? No! -- Chi2 3.160 -- p-value 0.368
Occupation -- Drift? No! -- Chi2 8.194 -- p-value 0.415
Relationship -- Drift? No! -- Chi2 0.485 -- p-value 0.993
Race -- Drift? No! -- Chi2 0.587 -- p-value 0.965
Sex -- Drift? No! -- Chi2 0.217 -- p-value 0.641
Capital Gain -- Drift? No! -- K-S 0.002 -- p-value 1.000
Capital Loss -- Drift? No! -- K-S 0.002 -- p-value 1.000
Hours per week -- Drift? No! -- K-S 0.012 -- p-value 0.512
Country -- Drift? No! -- Chi2 9.991 -- p-value 0.441
```

What about the second test set?

```
[12]: preds = cd.predict(X_t1)
      labels = ['No!', 'Yes!']
      print('Drift? {}'.format(labels[preds['data']['is_drift']]))
```

```
Drift? No!
```

We can again investigate the individual features:

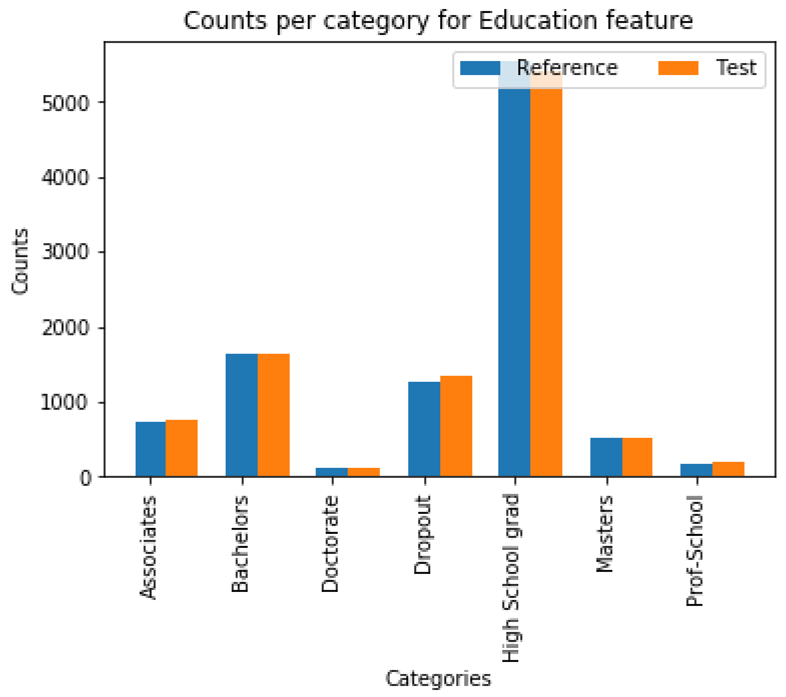
```
[13]: for f in range(cd.n_features):
          stat = 'Chi2' if f in list(categories_per_feature.keys()) else 'K-S'
          fname = feature_names[f]
          is_drift = (preds['data']['p_val'][f] < preds['data']['threshold']).astype(int)
          stat_val, p_val = preds['data']['distance'][f], preds['data']['p_val'][f]
          print(f'{fname} -- Drift? {labels[is_drift]} -- {stat} {stat_val:.3f} -- p-value
      ↪{p_val:.3f}')
```

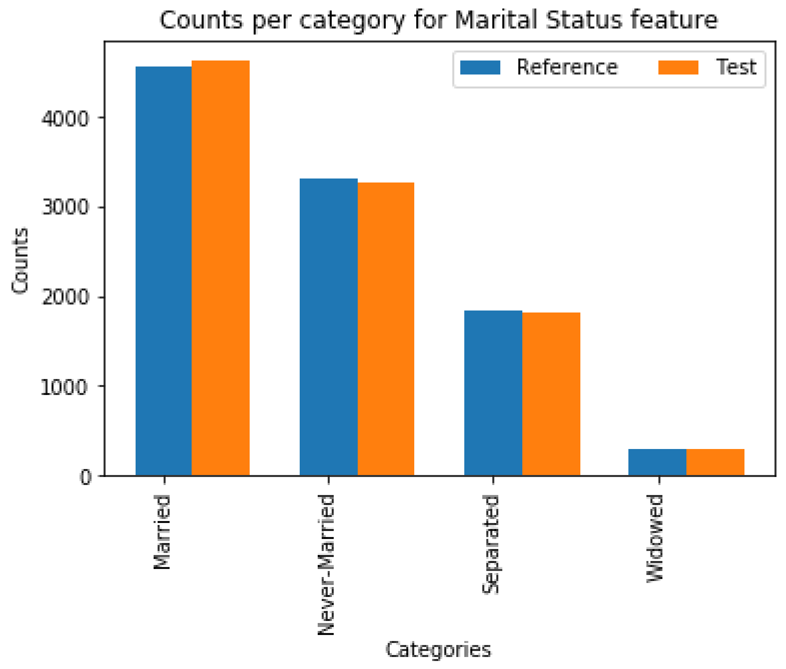
```
Age -- Drift? No! -- K-S 0.007 -- p-value 0.967
Workclass -- Drift? No! -- Chi2 5.800 -- p-value 0.670
Education -- Drift? No! -- Chi2 5.413 -- p-value 0.492
Marital Status -- Drift? No! -- Chi2 1.167 -- p-value 0.761
Occupation -- Drift? No! -- Chi2 12.296 -- p-value 0.138
Relationship -- Drift? No! -- Chi2 6.520 -- p-value 0.259
Race -- Drift? No! -- Chi2 1.417 -- p-value 0.841
Sex -- Drift? No! -- Chi2 0.008 -- p-value 0.928
Capital Gain -- Drift? No! -- K-S 0.005 -- p-value 0.999
Capital Loss -- Drift? No! -- K-S 0.003 -- p-value 1.000
Hours per week -- Drift? No! -- K-S 0.004 -- p-value 1.000
Country -- Drift? No! -- Chi2 11.256 -- p-value 0.338
```

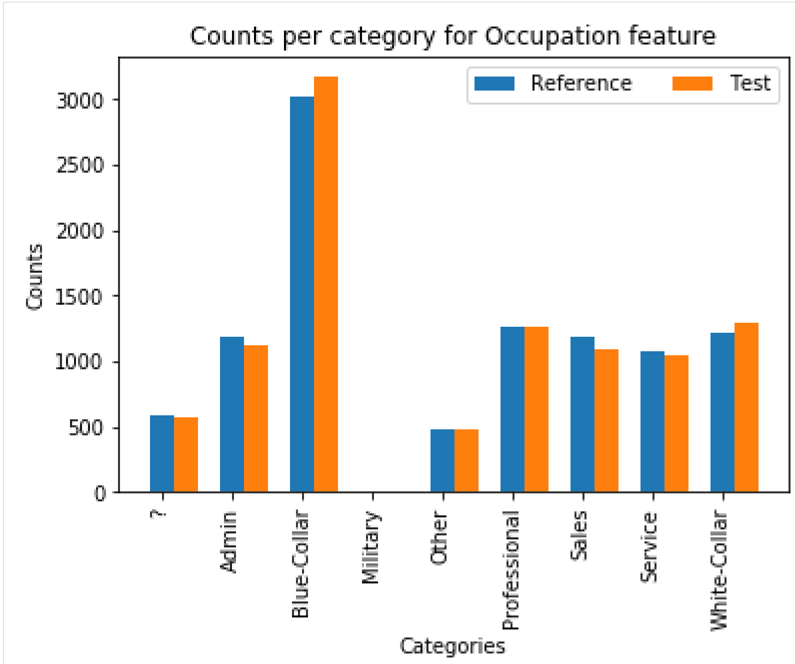
It seems like there is little divergence in the distributions of the features between the reference and test set. Let's visualize this:

```
[14]: def plot_categories(idx: int) -> None:
          # reference data
          x_ref_count = {f: [(X_ref[:, f] == v).sum() for v in vals]
                         for f, vals in cd.x_ref_categories.items()}
          fref_drift = {cat: x_ref_count[idx][i] for i, cat in enumerate(category_map[idx])}

          # test set
          cats = {f: list(np.unique(X_t1[:, f])) for f in categories_per_feature.keys()}
          X_count = {f: [(X_t1[:, f] == v).sum() for v in vals] for f, vals in cats.items()}
          fxt1_drift = {cat: X_count[idx][i] for i, cat in enumerate(category_map[idx])}

          # plot bar chart
          plot_labels = list(fxt1_drift.keys())
          ind = np.arange(len(plot_labels))
          width = .35
          fig, ax = plt.subplots()
          p1 = ax.bar(ind, list(fref_drift.values()), width)
          p2 = ax.bar(ind + width, list(fxt1_drift.values()), width)
          ax.set_title(f'Counts per category for {feature_names[idx]} feature')
          ax.set_xticks(ind + width / 2)
          ax.set_xticklabels(plot_labels)
          ax.legend((p1[0], p2[0]), ('Reference', 'Test'), loc='upper right', ncol=2)
          ax.set_ylabel('Counts')
          ax.set_xlabel('Categories')
          plt.xticks(list(np.arange(len(plot_labels))), plot_labels, rotation='vertical')
          plt.show()
```

```
[15]: plot_categories(2)
      plot_categories(3)
      plot_categories(4)
```


Counts per category for Education feature
Reference
Test
Counts
0
1000
2000
3000
4000
5000
Associates
Bachelors
Doctorate
Dropout
High School grad
Masters
Prof-School
Categories


Counts per category for Marital Status feature
Reference
Test
Counts
0
1000
2000
3000
4000
Married
Never-Married
Separated
Widowed
Categories



39.5 Categorical data drift

While the *TabularDrift* detector works fine with numerical or categorical features only, we can also directly use a categorical drift detector. In this case, we don't need to specify the categorical feature columns. First we construct a categorical-only dataset and then use the *ChiSquareDrift* detector:

[16]:
```
cols = list(category_map.keys())
cat_names = [feature_names[_] for _ in list(category_map.keys())]
X_ref_cat, X_t0_cat = X_ref[:, cols], X_t0[:, cols]
X_ref_cat.shape, X_t0_cat.shape
```

[16]:
```
((10000, 8), (10000, 8))
```

[17]:
```
cd = ChiSquareDrift(X_ref_cat, p_val=.05)
preds = cd.predict(X_t0_cat)
print('Drift? {}'.format(labels[preds['data']['is_drift']]))
```

```
Drift? No!
```

[18]:
```
print(f"Threshold {preds['data']['threshold']}")
for f in range(cd.n_features):
    fname = cat_names[f]
    is_drift = (preds['data']['p_val'][f] < preds['data']['threshold']).astype(int)
    stat_val, p_val = preds['data']['distance'][f], preds['data']['p_val'][f]
    print(f'{fname} -- Drift? {labels[is_drift]} -- {stat} {stat_val:.3f} -- p-value
↪{p_val:.3f}')
```

```
Threshold 0.00625
Workclass -- Drift? No! -- Chi2 8.487 -- p-value 0.387
Education -- Drift? No! -- Chi2 4.753 -- p-value 0.576
Marital Status -- Drift? No! -- Chi2 3.160 -- p-value 0.368
Occupation -- Drift? No! -- Chi2 8.194 -- p-value 0.415
```

(continues on next page)

(continued from previous page)

```
Relationship -- Drift? No! -- Chi2 0.485 -- p-value 0.993
Race -- Drift? No! -- Chi2 0.587 -- p-value 0.965
Sex -- Drift? No! -- Chi2 0.217 -- p-value 0.641
Country -- Drift? No! -- Chi2 9.991 -- p-value 0.441
```

CHAPTER

FORTY

KOLMOGOROV-SMIRNOV DATA DRIFT DETECTOR ON CIFAR-10

40.1 Method

The drift detector applies feature-wise two-sample Kolmogorov-Smirnov (K-S) tests. For multivariate data, the obtained p-values for each feature are aggregated either via the Bonferroni or the False Discovery Rate (FDR) correction. The Bonferroni correction is more conservative and controls for the probability of at least one false positive. The FDR correction on the other hand allows for an expected fraction of false positives to occur.

For high-dimensional data, we typically want to reduce the dimensionality before computing the feature-wise univariate K-S tests and aggregating those via the chosen correction method. Following suggestions in Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift, we incorporate randomly initialized encoders, also called Untrained AutoEncoders (UAE), black-box shift detection using the classifier's softmax outputs (BBSDs) and PCA as out-of-the box preprocessing methods. Preprocessing methods which do not rely on the classifier will usually pick up drift in the input data, while BBSDs focuses on label shift. The adversarial detector which is part of the library can also be transformed into a drift detector picking up drift that reduces the performance of the classification model. We can therefore combine different preprocessing techniques to figure out if there is drift which hurts the model performance, and whether this drift can be classified as input drift or label shift.

40.2 Backend

The method works with both the **PyTorch** and **TensorFlow** frameworks for the optional preprocessing step. Alibi Detect does however not install PyTorch for you. Check the PyTorch docs how to do this.

40.3 Dataset

CIFAR10 consists of 60,000 32 by 32 RGB images equally distributed over 10 classes. We evaluate the drift detector on the CIFAR-10-C dataset (Hendrycks & Dietterich, 2019). The instances in CIFAR-10-C have been corrupted and perturbed by various types of noise, blur, brightness etc. at different levels of severity, leading to a gradual decline in the classification model performance. We also check for drift against the original test set with class imbalances.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import os
import tensorflow as tf

from alibi_detect.cd import KSDrift
from alibi_detect.models.resnet import scale_by_instance
from alibi_detect.utils.fetching import fetch_tf_model, fetch_detector
```

(continues on next page)

(continued from previous page)

```
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.datasets import fetch_cifar10c, corruption_types_cifar10c
```

40.4 Load data

Original CIFAR-10 data:

```
[2]: (X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
y_train = y_train.astype('int64').reshape(-1,)
y_test = y_test.astype('int64').reshape(-1,)
```

For CIFAR-10-C, we can select from the following corruption types at 5 severity levels:

```
[3]: corruptions = corruption_types_cifar10c()
print(corruptions)
```

```
['brightness', 'contrast', 'defocus_blur', 'elastic_transform', 'fog', 'frost',
↪'gaussian_blur', 'gaussian_noise', 'glass_blur', 'impulse_noise', 'jpeg_compression
↪', 'motion_blur', 'pixelate', 'saturate', 'shot_noise', 'snow', 'spatter', 'speckle_
↪noise', 'zoom_blur']
```

Let's pick a subset of the corruptions at corruption level 5. Each corruption type consists of perturbations on all of the original test set images.

```
[4]: corruption = ['gaussian_noise', 'motion_blur', 'brightness', 'pixelate']
X_corr, y_corr = fetch_cifar10c(corruption=corruption, severity=5, return_X_y=True)
X_corr = X_corr.astype('float32') / 255
```

We split the original test set in a reference dataset and a dataset which should not be rejected under the *H0* of the K-S test. We also split the corrupted data by corruption type:

```
[5]: np.random.seed(0)
n_test = X_test.shape[0]
idx = np.random.choice(n_test, size=n_test // 2, replace=False)
idx_h0 = np.delete(np.arange(n_test), idx, axis=0)
X_ref,y_ref = X_test[idx], y_test[idx]
X_h0, y_h0 = X_test[idx_h0], y_test[idx_h0]
print(X_ref.shape, X_h0.shape)
```

```
(5000, 32, 32, 3) (5000, 32, 32, 3)
```

```
[6]: # check that the classes are more or less balanced
classes, counts_ref = np.unique(y_ref, return_counts=True)
counts_h0 = np.unique(y_h0, return_counts=True)[1]
print('Class Ref H0')
for cl, cref, ch0 in zip(classes, counts_ref, counts_h0):
    assert cref + ch0 == n_test // 10
    print('{}      {} {}'.format(cl, cref, ch0))
```

```
Class Ref H0
0     472 528
1     510 490
2     498 502
```

(continues on next page)

(continued from previous page)

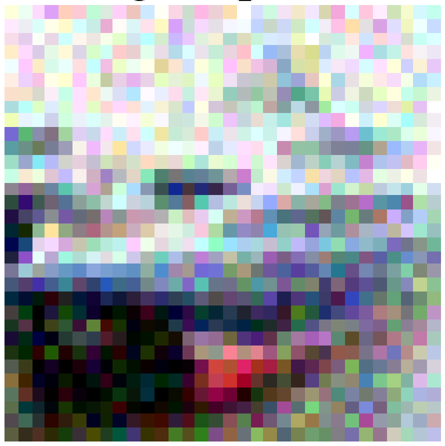
```
3      492 508
4      501 499
5      495 505
6      493 507
7      501 499
8      516 484
9      522 478
```

[7]:
```
n_corr = len(corruption)
X_c = [X_corr[i * n_test:(i + 1) * n_test] for i in range(n_corr)]
```

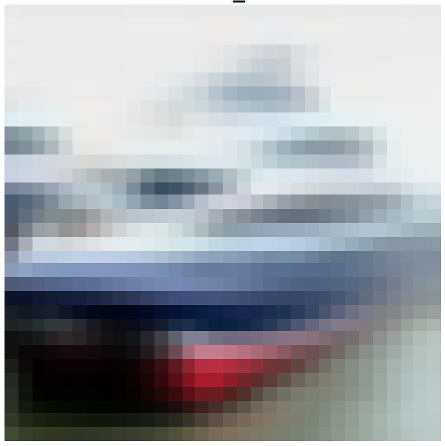
We can visualise the same instance for each corruption type:

[8]:
```
i = 1

n_test = X_test.shape[0]
plt.title('Original')
plt.axis('off')
plt.imshow(X_test[i])
plt.show()
for _ in range(len(corruption)):
    plt.title(corruption[_])
    plt.axis('off')
    plt.imshow(X_corr[n_test * _+ i])
    plt.show()
```

gaussian_noise



motion_blur



brightness

We can also verify that the performance of a classification model on CIFAR-10 drops significantly on this perturbed dataset:

```
[9]: dataset = 'cifar10'
model = 'resnet32'
clf = fetch_tf_model(dataset, model)
acc = clf.evaluate(scale_by_instance(X_test), y_test, batch_size=128, verbose=0)[1]
print('Test set accuracy:')
print('Original {:.4f}'.format(acc))
clf_accuracy = {'original': acc}
for _ in range(len(corruption)):
    acc = clf.evaluate(scale_by_instance(X_c[_]), y_test, batch_size=128,␣
↪verbose=0)[1]
    clf_accuracy[corruption[_]] = acc
    print('{} {:.4f}'.format(corruption[_], acc))
```

```
Test set accuracy:
Original 0.9278
gaussian_noise 0.2208
motion_blur 0.6339
brightness 0.8913
pixelate 0.3666
```

Given the drop in performance, it is important that we detect the harmful data drift!

40.5 Detect drift

First we try a drift detector using the **TensorFlow** framework for the preprocessing step. We are trying to detect data drift on high-dimensional (*32x32x3*) data using feature-wise univariate tests. It therefore makes sense to apply dimensionality reduction first. Some dimensionality reduction methods also used in Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift are readily available: a randomly initialized encoder (**UAE** or Untrained AutoEncoder in the paper), **BBSDs** (black-box shift detection using the classifier's softmax outputs) and **PCA**.

40.5.1 Random encoder

First we try the randomly initialized encoder:

[10]:
```
from functools import partial
from tensorflow.keras.layers import Conv2D, Dense, Flatten, InputLayer, Reshape
from alibi_detect.cd.tensorflow import preprocess_drift

tf.random.set_seed(0)

# define encoder
encoding_dim = 32
encoder_net = tf.keras.Sequential(
  [
      InputLayer(input_shape=(32, 32, 3)),
      Conv2D(64, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(128, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(512, 4, strides=2, padding='same', activation=tf.nn.relu),
      Flatten(),
      Dense(encoding_dim,)
  ]
)

# define preprocessing function
preprocess_fn = partial(preprocess_drift, model=encoder_net, batch_size=512)

# initialise drift detector
p_val = .05
cd = KSDrift(X_ref, p_val=p_val, preprocess_fn=preprocess_fn)

# we can also save/load an initialised detector
filepath = 'my_path'  # change to directory where detector is saved
save_detector(cd, filepath)
cd = load_detector(filepath)
```

The p-value used by the detector for the multivariate data with *encoding_dim* features is equal to *p_val / encoding_dim* because of the Bonferroni correction.

[11]:
```
assert cd.p_val / cd.n_features == p_val / encoding_dim
```

Let's check whether the detector thinks drift occurred on the different test sets and time the prediction calls:

[12]:
```
from timeit import default_timer as timer

labels = ['No!', 'Yes!']

def make_predictions(cd, x_h0, x_corr, corruption):
```

(continues on next page)

(continued from previous page)

```
    t = timer()
    preds = cd.predict(x_h0)
    dt = timer() - t
    print('No corruption')
    print('Drift? {}'.format(labels[preds['data']['is_drift']]))
    print('Feature-wise p-values:')
    print(preds['data']['p_val'])
    print(f'Time (s) {dt:.3f}')

    if isinstance(x_corr, list):
        for x, c in zip(x_corr, corruption):
            t = timer()
            preds = cd.predict(x)
            dt = timer() - t
            print('')
            print(f'Corruption type: {c}')
            print('Drift? {}'.format(labels[preds['data']['is_drift']]))
            print('Feature-wise p-values:')
            print(preds['data']['p_val'])
            print(f'Time (s) {dt:.3f}')
```

[13]:
```
make_predictions(cd, X_h0, X_c, corruption)
```

```
No corruption
Drift? No!
Feature-wise p-values:
[0.94146556 0.14195988 0.6440195  0.05512931 0.37907234 0.25943416
 0.87724036 0.48063537 0.11774229 0.677735   0.48063537 0.7442197
 0.08356539 0.14860499 0.31536394 0.31536394 0.61036026 0.36571503
 0.80732274 0.22020556 0.249175   0.46531922 0.11774229 0.45025542
 0.25943416 0.5936282  0.5604951  0.9571862  0.8642828  0.23921937
 0.90134364 0.6945301 ]
Time (s) 0.098

Corruption type: gaussian_noise
Drift? Yes!
Feature-wise p-values:
[4.95750410e-03 7.34658632e-03 5.52912503e-02 1.03268398e-08
 3.38559955e-01 8.17310661e-02 6.79804944e-03 2.71271050e-01
 1.55009609e-02 1.03484383e-02 1.82668434e-03 1.05715834e-01
 5.14261274e-08 1.42579767e-10 1.07582469e-04 1.49479797e-06
 2.90366083e-01 3.92065112e-06 3.53773794e-04 1.99041501e-01
 1.90719927e-03 6.04502577e-03 4.95599561e-10 6.04502577e-03
 5.12230098e-01 5.46741539e-05 3.83900553e-01 1.14905804e-01
 2.80400272e-02 8.04118258e-07 3.30103422e-03 2.05864832e-02]
Time (s) 0.181

Corruption type: motion_blur
Drift? Yes!
Feature-wise p-values:
[3.5206401e-07 1.1811157e-01 5.1718007e-04 3.8903630e-03 6.3595712e-01
 5.4199551e-04 3.7114436e-04 1.6068090e-02 1.9909975e-03 1.2925932e-02
 4.6739896e-09 7.1598392e-04 3.8931589e-04 6.6549936e-21 1.4105450e-07
 2.0635051e-05 7.2645240e-02 7.4932752e-05 1.0571583e-01 1.1322660e-04
 2.2843637e-04 1.5437353e-01 6.5380293e-03 3.2056447e-02 3.1007592e-02
 9.2208997e-05 6.6406779e-02 2.0782007e-03 1.1780208e-03 8.4564666e-10
 9.8475325e-04 3.5377379e-04]
```

(continues on next page)

(continued from previous page)

```
Time (s) 0.139

Corruption type: brightness
Drift? Yes!
Feature-wise p-values:
[0.0000000e+00 0.0000000e+00 4.9483204e-29 0.0000000e+00 0.0000000e+00
 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
 0.0000000e+00 0.0000000e+00 0.0000000e+00 2.7021131e-21 4.0524192e-38
 0.0000000e+00 0.0000000e+00 0.0000000e+00 9.2606446e-34 0.0000000e+00
 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
 3.9701583e-05 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
 0.0000000e+00 0.0000000e+00]
Time (s) 0.138

Corruption type: pixelate
Drift? Yes!
Feature-wise p-values:
[3.38559955e-01 2.09074125e-01 1.99041501e-01 8.56629852e-03
 5.97174764e-01 9.44178849e-02 1.66843131e-01 5.21439314e-01
 2.44745538e-02 4.58541155e-01 1.96971247e-04 1.35259181e-01
 1.67503790e-03 1.82668434e-03 1.14905804e-01 9.17565823e-02
 8.31667542e-01 6.25065863e-02 6.64067790e-02 9.24271531e-03
 2.41498753e-01 6.94294155e-01 1.08709060e-01 8.41466635e-02
 3.45860541e-01 1.38920277e-01 3.68379533e-01 3.03615063e-01
 3.76089692e-01 9.24271531e-03 5.03090501e-01 9.96722654e-03]
Time (s) 0.139
```

As expected, drift was only detected on the corrupted datasets. The feature-wise p-values for each univariate K-S test per (encoded) feature before multivariate correction show that most of them are well above the 0.05 threshold for *H0* and below for the corrupted datasets.

40.5.2 BBSDs

For **BBSDs**, we use the classifier's softmax outputs for black-box shift detection. This method is based on Detecting and Correcting for Label Shift with Black Box Predictors. The ResNet classifier is trained on data standardised by instance so we need to rescale the data.

```
[14]: X_train = scale_by_instance(X_train)
      X_test = scale_by_instance(X_test)
      X_ref = scale_by_instance(X_ref)
      X_h0 = scale_by_instance(X_h0)
      X_c = [scale_by_instance(X_c[i]) for i in range(n_corr)]
```

Now we initialize the detector. Here we use the output of the softmax layer to detect the drift, but other hidden layers can be extracted as well by setting *'layer'* to the index of the desired hidden layer in the model:

```
[15]: from alibi_detect.cd.tensorflow import HiddenOutput

      # define preprocessing function, we use the
      preprocess_fn = partial(preprocess_drift, model=HiddenOutput(clf, layer=-1), batch_
      ↪size=128)

      cd = KSDrift(X_ref, p_val=p_val, preprocess_fn=preprocess_fn)
```

Again we can see that the p-value used by the detector for the multivariate data with 10 features (number of CIFAR-10 classes) is equal to *p_val / 10* because of the Bonferroni correction.

```
[16]: assert cd.p_val / cd.n_features == p_val / 10
```

There is no drift on the original held out test set:

```
[17]: make_predictions(cd, X_h0, X_c, corruption)
```

```
No corruption
Drift? No!
Feature-wise p-values:
[0.11774229 0.52796143 0.19387017 0.20236294 0.496191   0.72781175
 0.12345381 0.420929   0.8367454  0.7604178 ]
Time (s) 1.003

Corruption type: gaussian_noise
Drift? Yes!
Feature-wise p-values:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Time (s) 1.970

Corruption type: motion_blur
Drift? Yes!
Feature-wise p-values:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Time (s) 1.798

Corruption type: brightness
Drift? Yes!
Feature-wise p-values:
[0.0000000e+00 4.2024049e-15 2.8963613e-33 4.8499879e-07 2.3718185e-15
 1.2473309e-05 2.9714003e-30 1.0611427e-09 4.6048109e-12 4.1857830e-17]
Time (s) 1.975

Corruption type: pixelate
Drift? Yes!
Feature-wise p-values:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Time (s) 1.894
```

40.6 Label drift

We can also check what happens when we introduce class imbalances between the reference data *X_ref* and the tested data *X_imb*. The reference data will use 75% of the instances of the first 5 classes and only 25% of the last 5. The data used for drift testing then uses respectively 25% and 75% of the test instances for the first and last 5 classes.

```
[19]: np.random.seed(0)
# get index for each class in the test set
num_classes = len(np.unique(y_test))
idx_by_class = [np.where(y_test == c)[0] for c in range(num_classes)]
# sample imbalanced data for different classes for X_ref and X_imb
perc_ref = .75
perc_ref_by_class = [perc_ref if c < 5 else 1 - perc_ref for c in range(num_classes)]
n_by_class = n_test // num_classes
X_ref = []
X_imb, y_imb = [], []
for _ in range(num_classes):
```

(continues on next page)

(continued from previous page)

```
    idx_class_ref = np.random.choice(n_by_class, size=int(perc_ref_by_class[_] * n_by_
↪class), replace=False)
    idx_ref = idx_by_class[_][idx_class_ref]
    idx_class_imb = np.delete(np.arange(n_by_class), idx_class_ref, axis=0)
    idx_imb = idx_by_class[_][idx_class_imb]
    assert idx_ref != idx_imb
    X_ref.append(X_test[idx_ref])
    X_imb.append(X_test[idx_imb])
    y_imb.append(y_test[idx_imb])
X_ref = np.concatenate(X_ref)
X_imb = np.concatenate(X_imb)
y_imb = np.concatenate(y_imb)
print(X_ref.shape, X_imb.shape, y_imb.shape)
```

```
(5000, 32, 32, 3) (5000, 32, 32, 3) (5000,)
```

Update reference dataset for the detector and make predictions. Note that we store the preprocessed reference data since the `preprocess_x_ref` kwarg is by default True:

```
[20]: cd.x_ref = cd.preprocess_fn(X_ref)
```

```
[21]: preds_imb = cd.predict(X_imb)
print('Drift? {}'.format(labels[preds_imb['data']['is_drift']]))
print(preds_imb['data']['p_val'])
```

```
Drift? Yes!
[6.10830360e-20 1.32319470e-20 7.62410424e-29 1.05537245e-17
 7.68910424e-23 1.57479264e-15 5.77457112e-19 1.94419707e-20
 5.02102509e-21 4.13147353e-21]
```

40.7 Update reference data

So far we have kept the reference data the same throughout the experiments. It is possible however that we want to test a new batch against the last *N* instances or against a batch of instances of fixed size where we give each instance we have seen up until now the same chance of being in the reference batch (reservoir sampling). The `update_x_ref` argument allows you to change the reference data update rule. It is a Dict which takes as key the update rule (*'last'* for last *N* instances or *'reservoir_sampling'*) and as value the batch size *N* of the reference data. You can also save the detector after the prediction calls to save the updated reference data.

```
[22]: N = 7500
cd = KSDrift(X_ref, p_val=.05, preprocess_fn=preprocess_fn, update_x_ref={'reservoir_
↪sampling': N})
```

The reference data is now updated with each `predict` call. Say we start with our imbalanced reference set and make a prediction on the remaining test set data *X_imb*, then the drift detector will figure out data drift has occurred.

```
[23]: preds_imb = cd.predict(X_imb)
print('Drift? {}'.format(labels[preds_imb['data']['is_drift']]))
```

```
Drift? Yes!
```

We can now see that the reference data consists of *N* instances, obtained through reservoir sampling.

```
[24]: assert cd.x_ref.shape[0] == N
```

We then draw a random sample from the training set and compare it with the updated reference data. This still highlights that there is data drift but will update the reference data again:

```
[25]: np.random.seed(0)
      perc_train = .5
      n_train = X_train.shape[0]
      idx_train = np.random.choice(n_train, size=int(perc_train * n_train), replace=False)
```

```
[26]: preds_train = cd.predict(X_train[idx_train])
      print('Drift? {}'.format(labels[preds_train['data']['is_drift']]))
```

```
Drift? Yes!
```

When we draw a new sample from the training set, it highlights that it is not drifting anymore against the reservoir in *X_ref*.

```
[27]: np.random.seed(1)
      perc_train = .1
      idx_train = np.random.choice(n_train, size=int(perc_train * n_train), replace=False)
      preds_train = cd.predict(X_train[idx_train])
      print('Drift? {}'.format(labels[preds_train['data']['is_drift']]))
```

```
Drift? No!
```

40.8 Multivariate correction mechanism

Instead of the Bonferroni correction for multivariate data, we can also use the less conservative False Discovery Rate (FDR) correction. See here or here for nice explanations. While the Bonferroni correction controls the probability of at least one false positive, the FDR correction controls for an expected amount of false positives. The `p_val` argument at initialisation time can be interpreted as the acceptable q-value when the FDR correction is applied.

```
[28]: cd = KSDrift(X_ref, p_val=.05, preprocess_fn=preprocess_fn, correction='fdr')

      preds_imb = cd.predict(X_imb)
      print('Drift? {}'.format(labels[preds_imb['data']['is_drift']]))
```

```
Drift? Yes!
```

40.9 Adversarial autoencoder as a malicious drift detector

We can leverage the adversarial scores obtained from an adversarial autoencoder trained on normal data and transform it into a data drift detector. The score function of the adversarial autoencoder becomes the preprocessing function for the drift detector. The K-S test is then a simple univariate test on the adversarial scores. Importantly, an adversarial drift detector flags **malicious data drift**. We can fetch the pretrained adversarial detector from a Google Cloud Bucket or train one from scratch:

```
[29]: load_pretrained = True
```

```
[30]: # change filepath to (absolute) directory where model is downloaded
      filepath = os.path.join(os.getcwd(), 'my_path')
      if load_pretrained:
          detector_type = 'adversarial'
```

(continues on next page)

(continued from previous page)

```
    detector_name = 'base'
    ad = fetch_detector(filepath, detector_type, dataset, detector_name, model=model)
    filepath = os.path.join(filepath, detector_name)
else:  # train detector from scratch
    # define encoder and decoder networks
    encoder_net = tf.keras.Sequential(
            [
                InputLayer(input_shape=(32, 32, 3)),
                Conv2D(32, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2D(64, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2D(256, 4, strides=2, padding='same',
                       activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Flatten(),
                Dense(40)
            ]
        )

    decoder_net = tf.keras.Sequential(
        [
                InputLayer(input_shape=(40,)),
                Dense(4 * 4 * 128, activation=tf.nn.relu),
                Reshape(target_shape=(4, 4, 128)),
                Conv2DTranspose(256, 4, strides=2, padding='same',
                                activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2DTranspose(64, 4, strides=2, padding='same',
                                activation=tf.nn.relu, kernel_regularizer=l1(1e-5)),
                Conv2DTranspose(3, 4, strides=2, padding='same',
                                activation=None, kernel_regularizer=l1(1e-5))
            ]
        )

    # initialise and train detector
    ad = AdversarialAE(encoder_net=encoder_net, decoder_net=decoder_net, model=clf)
    ad.fit(X_train, epochs=50, batch_size=128, verbose=True)

    # save the trained adversarial detector
    save_detector(ad, filepath)
```

```
WARNING:alibi_detect.utils.fetching:Directory /home/avl/git/fork-alibi-detect/
↪examples/my_path/base does not exist and is now created.
```

```
Downloading data from https://storage.googleapis.com/seldon-models/alibi-detect/ad/
↪cifar10/resnet32/base/model/encoder_net.h5
1867776/1862024 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/seldon-models/alibi-detect/ad/
↪cifar10/resnet32/base/model/decoder_net.h5
3522560/3514976 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/seldon-models/alibi-detect/ad/
↪cifar10/resnet32/base/model/model.h5
4300800/4295560 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/seldon-models/alibi-detect/ad/
↪cifar10/resnet32/base/model/checkpoint
8192/71␣
↪[==========================================================================================
↪- 0s 0us/step
Downloading data from https://storage.googleapis.com/seldon-models/alibi-detect/ad/
↪cifar10/resnet32/base/model/ae.ckpt.index
```

(continues on next page)

(continued from previous page)

```
8192/1291␣
↪[==========================================================================================
↪- 0s 0us/step
Downloading data from https://storage.googleapis.com/seldon-models/alibi-detect/ad/
↪cifar10/resnet32/base/model/ae.ckpt.data-00000-of-00002
8192/2462␣
↪[==========================================================================================
↪- 0s 0us/step
Downloading data from https://storage.googleapis.com/seldon-models/alibi-detect/ad/
↪cifar10/resnet32/base/model/ae.ckpt.data-00001-of-00002
5341184/5337900 [==============================] - 0s 0us/step
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your␣
↪model is starting with a freshly initialized optimizer.
```

```
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your␣
↪model is starting with a freshly initialized optimizer.
WARNING:alibi_detect.ad.adversarialae:No threshold level set. Need to infer threshold␣
↪using `infer_threshold`.
```

Initialise the drift detector:

```
[34]: np.random.seed(0)
idx = np.random.choice(n_test, size=n_test // 2, replace=False)
X_ref = scale_by_instance(X_test[idx])

# adversarial score fn = preprocess step
preprocess_fn = partial(ad.score, batch_size=128)

cd = KSDrift(X_ref, p_val=.05, preprocess_fn=preprocess_fn)
```

Make drift predictions on the original test set and corrupted data:

```
[35]: clf_accuracy['h0'] = clf.evaluate(X_h0, y_h0, batch_size=128, verbose=0)[1]
preds_h0 = cd.predict(X_h0)
print('H0: Accuracy {:.4f} -- Drift? {}'.format(
    clf_accuracy['h0'], labels[preds_h0['data']['is_drift']]))
clf_accuracy['imb'] = clf.evaluate(X_imb, y_imb, batch_size=128, verbose=0)[1]
preds_imb = cd.predict(X_imb)
print('imbalance: Accuracy {:.4f} -- Drift? {}'.format(
    clf_accuracy['imb'], labels[preds_imb['data']['is_drift']]))
for x, c in zip(X_c, corruption):
    preds = cd.predict(x)
    print('{}: Accuracy {:.4f} -- Drift? {}'.format(
        c, clf_accuracy[c],labels[preds['data']['is_drift']]))
```

```
H0: Accuracy 0.9286 -- Drift? No!
imbalance: Accuracy 0.9282 -- Drift? No!
```

(continues on next page)

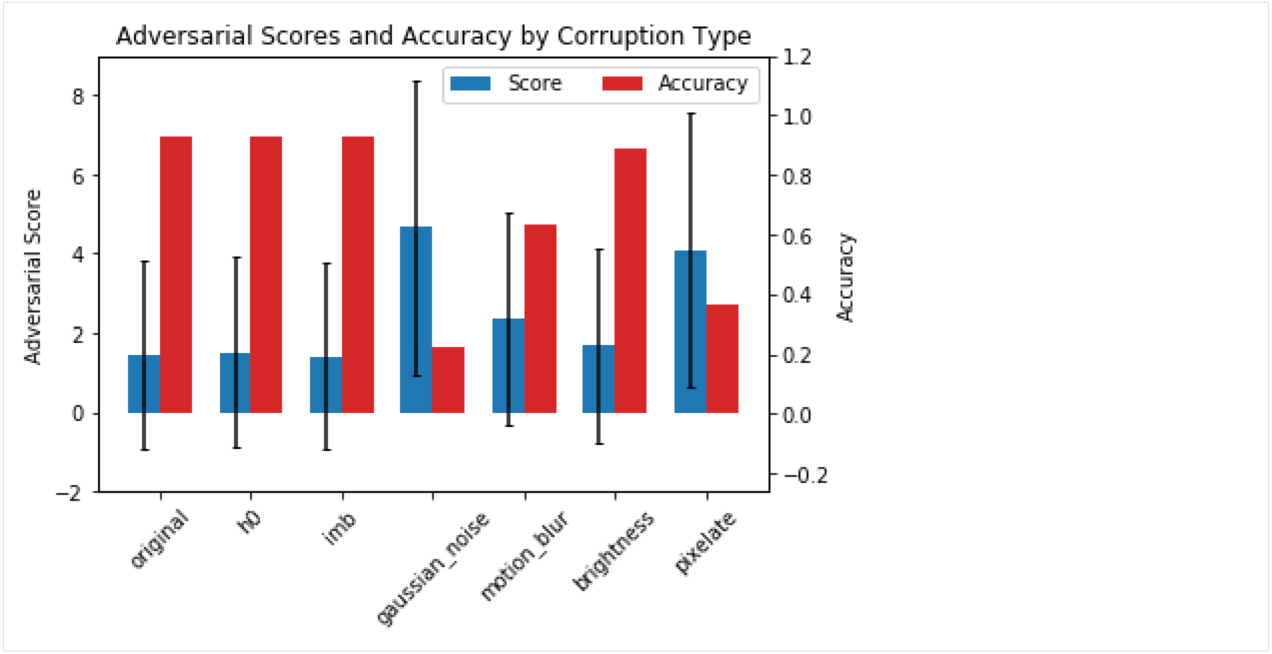
(continued from previous page)

```
gaussian_noise: Accuracy 0.2208 -- Drift? Yes!
motion_blur: Accuracy 0.6339 -- Drift? Yes!
brightness: Accuracy 0.8913 -- Drift? Yes!
pixelate: Accuracy 0.3666 -- Drift? Yes!
```

While *X_imb* clearly exhibits input data drift due to the introduced class imbalances, it is not flagged by the adversarial drift detector since the performance of the classifier is not affected and the drift is not malicious. We can visualise this by plotting the adversarial scores together with the harmfulness of the data corruption as reflected by the drop in classifier accuracy:

[36]:
```
adv_scores = {}
score = ad.score(X_ref, batch_size=128)
adv_scores['original'] = {'mean': score.mean(), 'std': score.std()}
score = ad.score(X_h0, batch_size=128)
adv_scores['h0'] = {'mean': score.mean(), 'std': score.std()}
score = ad.score(X_imb, batch_size=128)
adv_scores['imb'] = {'mean': score.mean(), 'std': score.std()}

for x, c in zip(X_c, corruption):
    score_x = ad.score(x, batch_size=128)
    adv_scores[c] = {'mean': score_x.mean(), 'std': score_x.std()}
```

[37]:
```
mu = [v['mean'] for _, v in adv_scores.items()]
stdev = [v['std'] for _, v in adv_scores.items()]
xlabels = list(adv_scores.keys())
acc = [clf_accuracy[label] for label in xlabels]
xticks = np.arange(len(mu))

width = .35

fig, ax = plt.subplots()
ax2 = ax.twinx()

p1 = ax.bar(xticks, mu, width, yerr=stdev, capsize=2)
color = 'tab:red'
p2 = ax2.bar(xticks + width, acc, width, color=color)

ax.set_title('Adversarial Scores and Accuracy by Corruption Type')
ax.set_xticks(xticks + width / 2)
ax.set_xticklabels(xlabels, rotation=45)
ax.legend((p1[0], p2[0]), ('Score', 'Accuracy'), loc='upper right', ncol=2)
ax.set_ylabel('Adversarial Score')

color = 'tab:red'
ax2.set_ylabel('Accuracy')
ax2.set_ylim((-.26,1.2))
ax.set_ylim((-2,9))

plt.show()
```



We can therefore **use the scores of the detector itself to quantify the harmfulness of the drift**! We can generalise this to all the corruptions at each severity level in CIFAR-10-C:

```
[38]: def accuracy(y_true: np.ndarray, y_pred: np.ndarray) -> float:
          return (y_true == y_pred).astype(int).sum() / y_true.shape[0]
```

```
[40]: from alibi_detect.utils.prediction import predict_batch

      severities = [1, 2, 3, 4, 5]

      score_drift = {
          1: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
          2: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
          3: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
          4: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
          5: {'all': [], 'harm': [], 'noharm': [], 'acc': 0},
      }

      y_pred = predict_batch(clf, X_test, batch_size=256, return_class=True)
      score_x = ad.score(X_test, batch_size=256)

      for s in severities:
          print('\nSeverity: {} of {}'.format(s, len(severities)))

          print('Loading corrupted dataset...')
          X_corr, y_corr = fetch_cifar10c(corruption=corruptions, severity=s, return_X_
      ↪y=True)
          X_corr = X_corr.astype('float32')

          print('Preprocess data...')
          X_corr = scale_by_instance(X_corr)

          print('Make predictions on corrupted dataset...')
          y_pred_corr = predict_batch(clf, X_corr, batch_size=256, return_class=True)
```

(continues on next page)

(continued from previous page)

```
    print('Compute adversarial scores on corrupted dataset...')
    score_corr = ad.score(X_corr, batch_size=256)

    print('Get labels for malicious corruptions...')
    labels_corr = np.zeros(score_corr.shape[0])
    repeat = y_corr.shape[0] // y_test.shape[0]
    y_pred_repeat = np.tile(y_pred, (repeat,))
    # malicious/harmful corruption: original prediction correct but
    # prediction on corrupted data incorrect
    idx_orig_right = np.where(y_pred_repeat == y_corr)[0]
    idx_corr_wrong = np.where(y_pred_corr != y_corr)[0]
    idx_harmful = np.intersect1d(idx_orig_right, idx_corr_wrong)
    labels_corr[idx_harmful] = 1
    labels = np.concatenate([np.zeros(X_test.shape[0]), labels_corr]).astype(int)
    # harmless corruption: original prediction correct and prediction
    # on corrupted data correct
    idx_corr_right = np.where(y_pred_corr == y_corr)[0]
    idx_harmless = np.intersect1d(idx_orig_right, idx_corr_right)

    score_drift[s]['all'] = score_corr
    score_drift[s]['harm'] = score_corr[idx_harmful]
    score_drift[s]['noharm'] = score_corr[idx_harmless]
    score_drift[s]['acc'] = accuracy(y_corr, y_pred_corr)
```

```
Severity: 1 of 5
Loading corrupted dataset...
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Get labels for malicious corruptions...

Severity: 2 of 5
Loading corrupted dataset...
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Get labels for malicious corruptions...

Severity: 3 of 5
Loading corrupted dataset...
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Get labels for malicious corruptions...

Severity: 4 of 5
Loading corrupted dataset...
Preprocess data...
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Get labels for malicious corruptions...

Severity: 5 of 5
Loading corrupted dataset...
Preprocess data...
```

(continues on next page)

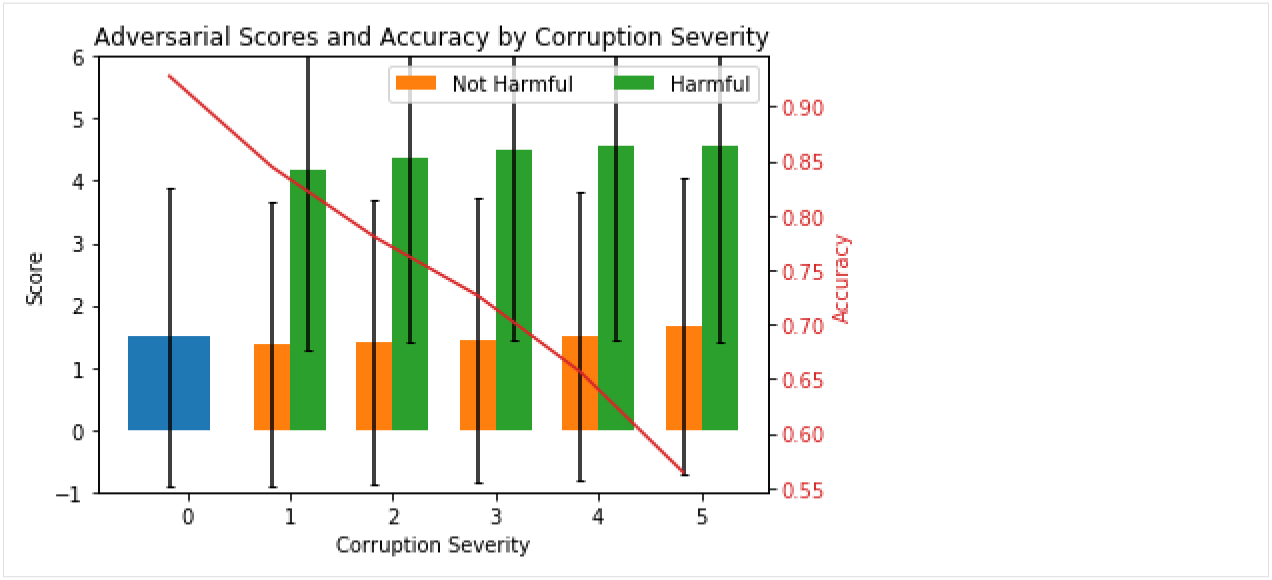
(continued from previous page)

```
Make predictions on corrupted dataset...
Compute adversarial scores on corrupted dataset...
Get labels for malicious corruptions...
```

We now compute mean scores and standard deviations per severity level and plot the results. The plot shows the mean adversarial scores (lhs) and ResNet-32 accuracies (rhs) for increasing data corruption severity levels. Level 0 corresponds to the original test set. Harmful scores are scores from instances which have been flipped from the correct to an incorrect prediction because of the corruption. Not harmful means that the prediction was unchanged after the corruption.

[41]:
```
mu_noharm, std_noharm = [], []
mu_harm, std_harm = [], []
acc = [clf_accuracy['original']]
for k, v in score_drift.items():
    mu_noharm.append(v['noharm'].mean())
    std_noharm.append(v['noharm'].std())
    mu_harm.append(v['harm'].mean())
    std_harm.append(v['harm'].std())
    acc.append(v['acc'])
```

[42]:
```
plot_labels = ['0', '1', '2', '3', '4', '5']

N = 6
ind = np.arange(N)
width = .35

fig_bar_cd, ax = plt.subplots()
ax2 = ax.twinx()

p0 = ax.bar(ind[0], score_x.mean(), yerr=score_x.std(), capsize=2)
p1 = ax.bar(ind[1:], mu_noharm, width, yerr=std_noharm, capsize=2)
p2 = ax.bar(ind[1:] + width, mu_harm, width, yerr=std_harm, capsize=2)

ax.set_title('Adversarial Scores and Accuracy by Corruption Severity')
ax.set_xticks(ind + width / 2)
ax.set_xticklabels(plot_labels)
ax.set_ylim((-1,6))
ax.legend((p1[0], p2[0]), ('Not Harmful', 'Harmful'), loc='upper right', ncol=2)
ax.set_ylabel('Score')
ax.set_xlabel('Corruption Severity')

color = 'tab:red'
ax2.set_ylabel('Accuracy', color=color)
ax2.plot(acc, color=color)
ax2.tick_params(axis='y', labelcolor=color)

plt.show()
```


Adversarial Scores and Accuracy by Corruption Severity
Not Harmful
Harmful
Score
Accuracy
Corruption Severity

CHAPTER

FORTYONE

MAXIMUM MEAN DISCREPANCY DRIFT DETECTOR ON CIFAR-10

41.1 Method

The Maximum Mean Discrepancy (MMD) detector is a kernel-based method for multivariate 2 sample testing. The MMD is a distance-based measure between 2 distributions p and q based on the mean embeddings μ_p and μ_q in a reproducing kernel Hilbert space F:

$$MMD(F, p, q) = ||\mu_p - \mu_q||^2_F \tag{41.1}$$

(41.2)

We can compute unbiased estimates of MMD^2 from the samples of the 2 distributions after applying the kernel trick. We use by default a radial basis function kernel, but users are free to pass their own kernel of preference to the detector. We obtain a p-value via a permutation test on the values of MMD^2. This method is also described in Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift.

41.2 Backend

The method is implemented in both the *PyTorch* and *TensorFlow* frameworks with support for CPU and GPU. Various preprocessing steps are also supported out-of-the box in Alibi Detect for both frameworks and illustrated throughout the notebook. Alibi Detect does however not install PyTorch for you. Check the PyTorch docs how to do this.

41.3 Dataset

CIFAR10 consists of 60,000 32 by 32 RGB images equally distributed over 10 classes. We evaluate the drift detector on the CIFAR-10-C dataset (Hendrycks & Dietterich, 2019). The instances in CIFAR-10-C have been corrupted and perturbed by various types of noise, blur, brightness etc. at different levels of severity, leading to a gradual decline in the classification model performance. We also check for drift against the original test set with class imbalances.

[1]:
```
from functools import partial
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

from alibi_detect.cd import MMDDrift
from alibi_detect.models.tensorflow.resnet import scale_by_instance
from alibi_detect.utils.fetching import fetch_tf_model
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.datasets import fetch_cifar10c, corruption_types_cifar10c
```

41.4 Load data

Original CIFAR-10 data:

```
[2]: (X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
y_train = y_train.astype('int64').reshape(-1,)
y_test = y_test.astype('int64').reshape(-1,)
```

For CIFAR-10-C, we can select from the following corruption types at 5 severity levels:

```
[3]: corruptions = corruption_types_cifar10c()
print(corruptions)
```

```
['brightness', 'contrast', 'defocus_blur', 'elastic_transform', 'fog', 'frost',
↪'gaussian_blur', 'gaussian_noise', 'glass_blur', 'impulse_noise', 'jpeg_compression
↪', 'motion_blur', 'pixelate', 'saturate', 'shot_noise', 'snow', 'spatter', 'speckle_
↪noise', 'zoom_blur']
```

Let's pick a subset of the corruptions at corruption level 5. Each corruption type consists of perturbations on all of the original test set images.

```
[4]: corruption = ['gaussian_noise', 'motion_blur', 'brightness', 'pixelate']
X_corr, y_corr = fetch_cifar10c(corruption=corruption, severity=5, return_X_y=True)
X_corr = X_corr.astype('float32') / 255
```

We split the original test set in a reference dataset and a dataset which should not be rejected under the *H0* of the MMD test. We also split the corrupted data by corruption type:

```
[5]: np.random.seed(0)
n_test = X_test.shape[0]
idx = np.random.choice(n_test, size=n_test // 2, replace=False)
idx_h0 = np.delete(np.arange(n_test), idx, axis=0)
X_ref,y_ref = X_test[idx], y_test[idx]
X_h0, y_h0 = X_test[idx_h0], y_test[idx_h0]
print(X_ref.shape, X_h0.shape)
```

```
(5000, 32, 32, 3) (5000, 32, 32, 3)
```

```
[6]: # check that the classes are more or less balanced
classes, counts_ref = np.unique(y_ref, return_counts=True)
counts_h0 = np.unique(y_h0, return_counts=True)[1]
print('Class Ref H0')
for cl, cref, ch0 in zip(classes, counts_ref, counts_h0):
    assert cref + ch0 == n_test // 10
    print('{}     {} {}'.format(cl, cref, ch0))
```

```
Class Ref H0
0     472 528
1     510 490
2     498 502
3     492 508
4     501 499
5     495 505
6     493 507
7     501 499
```

(continues on next page)

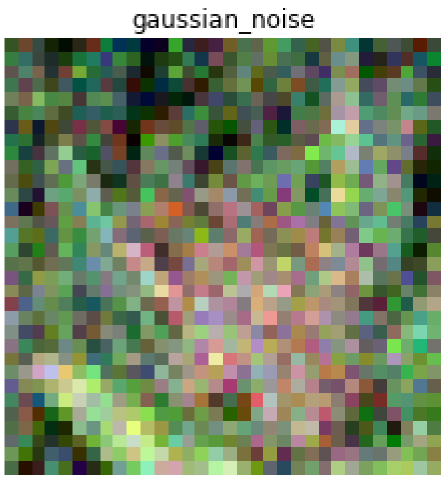
(continued from previous page)

```
8      516 484
9      522 478
```

[7]:
```
n_corr = len(corruption)
X_c = [X_corr[i * n_test:(i + 1) * n_test] for i in range(n_corr)]
```

We can visualise the same instance for each corruption type:

[8]:
```
i = 4

n_test = X_test.shape[0]
plt.title('Original')
plt.axis('off')
plt.imshow(X_test[i])
plt.show()
for _ in range(len(corruption)):
    plt.title(corruption[_])
    plt.axis('off')
    plt.imshow(X_corr[n_test * _+ i])
    plt.show()
```



We can also verify that the performance of a classification model on CIFAR-10 drops significantly on this perturbed dataset:

[9]:
```
dataset = 'cifar10'
```

(continues on next page)

(continued from previous page)

```
model = 'resnet32'
clf = fetch_tf_model(dataset, model)
acc = clf.evaluate(scale_by_instance(X_test), y_test, batch_size=128, verbose=0)[1]
print('Test set accuracy:')
print('Original {:.4f}'.format(acc))
clf_accuracy = {'original': acc}
for _ in range(len(corruption)):
    acc = clf.evaluate(scale_by_instance(X_c[_]), y_test, batch_size=128,␣
↪verbose=0)[1]
    clf_accuracy[corruption[_]] = acc
    print('{} {:.4f}'.format(corruption[_], acc))
```

```
Test set accuracy:
Original 0.9278
gaussian_noise 0.2208
motion_blur 0.6339
brightness 0.8913
pixelate 0.3666
```

Given the drop in performance, it is important that we detect the harmful data drift!

41.5 Detect drift with TensorFlow backend

First we try a drift detector using the *TensorFlow* framework for both the preprocessing and the *MMD* computation steps.

We are trying to detect data drift on high-dimensional (*32x32x3*) data using a multivariate MMD permutation test. It therefore makes sense to apply dimensionality reduction first. Some dimensionality reduction methods also used in Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift are readily available: a randomly initialized encoder (**UAE** or Untrained AutoEncoder in the paper), **BBSDs** (black-box shift detection using the classifier's softmax outputs) and **PCA**.

41.5.1 Random encoder

First we try the randomly initialized encoder:

```
[10]: from tensorflow.keras.layers import Conv2D, Dense, Flatten, InputLayer, Reshape
from alibi_detect.cd.tensorflow import preprocess_drift

tf.random.set_seed(0)

# define encoder
encoding_dim = 32
encoder_net = tf.keras.Sequential(
  [
      InputLayer(input_shape=(32, 32, 3)),
      Conv2D(64, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(128, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(512, 4, strides=2, padding='same', activation=tf.nn.relu),
      Flatten(),
      Dense(encoding_dim,)
  ]
)
```

(continues on next page)

(continued from previous page)

```
# define preprocessing function
preprocess_fn = partial(preprocess_drift, model=encoder_net, batch_size=512)

# initialise drift detector
cd = MMDDrift(X_ref, backend='tensorflow', p_val=.05,
              preprocess_fn=preprocess_fn, n_permutations=100)

# we can also save/load an initialised detector
filepath = 'my_path'  # change to directory where detector is saved
save_detector(cd, filepath)
cd = load_detector(filepath)
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*␣
↪compiled. Compile it manually.
WARNING:alibi_detect.cd.base:`sigma` is specified for the kernel and `configure_
↪kernel_from_x_ref` is set to True. `sigma` argument takes priority over `configure_
↪kernel_from_x_ref` (set to False).
```

Let's check whether the detector thinks drift occurred on the different test sets and time the prediction calls:

[11]:
```
from timeit import default_timer as timer

labels = ['No!', 'Yes!']

def make_predictions(cd, x_h0, x_corr, corruption):
    t = timer()
    preds = cd.predict(x_h0)
    dt = timer() - t
    print('No corruption')
    print('Drift? {}'.format(labels[preds['data']['is_drift']]))
    print(f'p-value: {preds["data"]["p_val"]:.3f}')
    print(f'Time (s) {dt:.3f}')

    if isinstance(x_corr, list):
        for x, c in zip(x_corr, corruption):
            t = timer()
            preds = cd.predict(x)
            dt = timer() - t
            print('')
            print(f'Corruption type: {c}')
            print('Drift? {}'.format(labels[preds['data']['is_drift']]))
            print(f'p-value: {preds["data"]["p_val"]:.3f}')
            print(f'Time (s) {dt:.3f}')
```

[12]:
```
make_predictions(cd, X_h0, X_c, corruption)
```

```
No corruption
Drift? No!
p-value: 0.680
Time (s) 2.217

Corruption type: gaussian_noise
Drift? Yes!
p-value: 0.000
```

(continues on next page)

(continued from previous page)

```
Time (s) 6.074

Corruption type: motion_blur
Drift? Yes!
p-value: 0.000
Time (s) 6.031

Corruption type: brightness
Drift? Yes!
p-value: 0.000
Time (s) 6.019

Corruption type: pixelate
Drift? Yes!
p-value: 0.000
Time (s) 6.010
```

As expected, drift was only detected on the corrupted datasets.

41.5.2 BBSDs

For **BBSDs**, we use the classifier's softmax outputs for black-box shift detection. This method is based on Detecting and Correcting for Label Shift with Black Box Predictors. The ResNet classifier is trained on data standardised by instance so we need to rescale the data.

```
[13]: X_ref_bbsds = scale_by_instance(X_ref)
      X_h0_bbsds = scale_by_instance(X_h0)
      X_c_bbsds = [scale_by_instance(X_c[i]) for i in range(n_corr)]
```

Initialisation of the drift detector. Here we use the output of the softmax layer to detect the drift, but other hidden layers can be extracted as well by setting *'layer'* to the index of the desired hidden layer in the model:

```
[14]: from alibi_detect.cd.tensorflow import HiddenOutput

      # define preprocessing function
      preprocess_fn = partial(preprocess_drift, model=HiddenOutput(clf, layer=-1), batch_
      ↪size=128)

      # initialise drift detector
      cd = MMDDrift(X_ref_bbsds, backend='tensorflow', p_val=.05,
                    preprocess_fn=preprocess_fn, n_permutations=100)
```

```
[15]: make_predictions(cd, X_h0_bbsds, X_c_bbsds, corruption)
```

```
No corruption
Drift? No!
p-value: 0.440
Time (s) 3.072

Corruption type: gaussian_noise
Drift? Yes!
p-value: 0.000
Time (s) 7.701

Corruption type: motion_blur
```

(continues on next page)

(continued from previous page)

```
Drift? Yes!
p-value: 0.000
Time (s) 7.754

Corruption type: brightness
Drift? Yes!
p-value: 0.000
Time (s) 7.760

Corruption type: pixelate
Drift? Yes!
p-value: 0.000
Time (s) 7.732
```

Again drift is only flagged on the perturbed data.

41.6 Detect drift with PyTorch backend

We can do the same thing using the *PyTorch* backend. We illustrate this using the randomly initialized encoder as preprocessing step:

[16]:
```
import torch
import torch.nn as nn

# set random seed and device
seed = 0
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda
```

Since our *PyTorch* encoder expects the images in a *(batch size, channels, height, width)* format, we transpose the data:

[17]:
```
def permute_c(x):
    return np.transpose(x.astype(np.float32), (0, 3, 1, 2))

X_ref_pt = permute_c(X_ref)
X_h0_pt = permute_c(X_h0)
X_c_pt = [permute_c(xc) for xc in X_c]
print(X_ref_pt.shape, X_h0_pt.shape, X_c_pt[0].shape)
```

```
(5000, 3, 32, 32) (5000, 3, 32, 32) (10000, 3, 32, 32)
```

[18]:
```
from alibi_detect.cd.pytorch import preprocess_drift

# define encoder
encoder_net = nn.Sequential(
    nn.Conv2d(3, 64, 4, stride=2, padding=0),
    nn.ReLU(),
    nn.Conv2d(64, 128, 4, stride=2, padding=0),
    nn.ReLU(),
    nn.Conv2d(128, 512, 4, stride=2, padding=0),
```

(continues on next page)

(continued from previous page)

```
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(2048, encoding_dim)
).to(device).eval()

# define preprocessing function
preprocess_fn = partial(preprocess_drift, model=encoder_net, device=device, batch_
↪size=512)

# initialise drift detector
cd = MMDDrift(X_ref_pt, backend='pytorch', p_val=.05,
              preprocess_fn=preprocess_fn, n_permutations=100)
```

[19]:
```
make_predictions(cd, X_h0_pt, X_c_pt, corruption)
```

```
No corruption
Drift? No!
p-value: 0.730
Time (s) 0.478

Corruption type: gaussian_noise
Drift? Yes!
p-value: 0.000
Time (s) 1.104

Corruption type: motion_blur
Drift? Yes!
p-value: 0.000
Time (s) 1.066

Corruption type: brightness
Drift? Yes!
p-value: 0.000
Time (s) 1.065

Corruption type: pixelate
Drift? Yes!
p-value: 0.000
Time (s) 1.066
```

The drift detector will attempt to use the GPU if available and otherwise falls back on the CPU. We can also explicitly specify the device. Let's compare the GPU speed up with the CPU implementation:

[20]:
```
device = torch.device('cpu')
preprocess_fn = partial(preprocess_drift, model=encoder_net.to(device),
                        device=device, batch_size=512)

cd = MMDDrift(X_ref_pt, backend='pytorch', preprocess_fn=preprocess_fn, device='cpu')
```

[21]:
```
make_predictions(cd, X_h0_pt, X_c_pt, corruption)
```

```
No corruption
Drift? No!
p-value: 0.670
Time (s) 14.282

Corruption type: gaussian_noise
```

(continues on next page)

(continued from previous page)

```
Drift? Yes!
p-value: 0.000
Time (s) 32.061

Corruption type: motion_blur
Drift? Yes!
p-value: 0.000
Time (s) 32.060

Corruption type: brightness
Drift? Yes!
p-value: 0.000
Time (s) 32.459

Corruption type: pixelate
Drift? Yes!
p-value: 0.000
Time (s) 35.935
```

Notice the over **30x acceleration** provided by the GPU.

Similar to the *TensorFlow* implementation, *PyTorch* can also use the hidden layer output from a pretrained model for the preprocessing step via:

```
from alibi_detect.cd.pytorch import HiddenOutput
```

CHAPTER

FORTYTWO

TEXT DRIFT DETECTION ON IMDB MOVIE REVIEWS

42.1 Method

We detect drift on text data using both the Maximum Mean Discrepancy and Kolmogorov-Smirnov (K-S) detectors. In this example notebook we will focus on detecting covariate shift $\Delta p(x)$ as detecting predicted label distribution drift does not differ from other modalities (check K-S and MMD drift on CIFAR-10).

It becomes however a little bit more involved when we want to pick up input data drift $\Delta p(x)$. When we deal with tabular or image data, we can either directly apply the two sample hypothesis test on the input or do the test after a preprocessing step with for instance a randomly initialized encoder as proposed in Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift (they call it an Untrained AutoEncoder or *UAE*). It is not as straightforward when dealing with text, both in string or tokenized format as they don't directly represent the semantics of the input.

As a result, we extract (contextual) embeddings for the text and detect drift on those. This procedure has a significant impact on the type of drift we detect. Strictly speaking we are not detecting $\Delta p(x)$ anymore since the whole training procedure (objective function, training data etc) for the (pre)trained embeddings has an impact on the embeddings we extract.

The library contains functionality to leverage pre-trained embeddings from HuggingFace's transformer package but also allows you to easily use your own embeddings of choice. Both options are illustrated with examples in this notebook.

42.2 Backend

The method works with both the **PyTorch** and **TensorFlow** frameworks for the statistical tests and preprocessing steps. Alibi Detect does however not install PyTorch for you. Check the PyTorch docs how to do this.

42.3 Dataset

Binary sentiment classification dataset containing $25,000$ movie reviews for training and $25,000$ for testing.

[1]:
```
import nlp
import numpy as np
import os
import tensorflow as tf
from transformers import AutoTokenizer
from alibi_detect.cd import KSDrift, MMDDrift
from alibi_detect.utils.saving import save_detector, load_detector
```

42.3.1 Load tokenizer

```
[2]: model_name = 'bert-base-cased'
     tokenizer = AutoTokenizer.from_pretrained(model_name)
```

```
INFO:transformers.configuration_utils:loading configuration file https://s3.amazonaws.
↪com/models.huggingface.co/bert/bert-base-cased-config.json from cache at /home/avl/.
↪cache/torch/transformers/
↪b945b69218e98b3e2c95acf911789741307dec43c698d35fad11c1ae28bda352.
↪9da767be51e1327499df13488672789394e2ca38b877837e52618a67d7002391
INFO:transformers.configuration_utils:Model config BertConfig {
  "architectures": [
    "BertForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "type_vocab_size": 2,
  "vocab_size": 28996
}

INFO:transformers.tokenization_utils:loading file https://s3.amazonaws.com/models.
↪huggingface.co/bert/bert-base-cased-vocab.txt from cache at /home/avl/.cache/torch/
↪transformers/5e8a2b4893d13790ed4150ca1906be5f7a03d6c4ddf62296c383f6db42814db2.
↪e13dbb970cb325137104fb2e5f36fe865f27746c6b526f6352861b1980eb80b1
```

42.3.2 Load data

```
[3]: def load_dataset(dataset: str, split: str = 'test'):
         data = nlp.load_dataset(dataset)
         X, y = [], []
         for x in data[split]:
             X.append(x['text'])
             y.append(x['label'])
         X = np.array(X)
         y = np.array(y)
         return X, y
```

```
[4]: X, y = load_dataset('imdb', split='train')
     print(X.shape, y.shape)
```

```
INFO:nlp.load:Checking /home/avl/.cache/huggingface/datasets/
↪d3b7716978cb901261e59327d43b04c52d6d29e50eeac39bea0816865a584081.
↪7c39fd6270c5ee55bcf2e4de23af77ef299e0df65be3f3e84454dcef7175844a.py for additional␣
↪imports.
INFO:filelock:Lock 139754813800592 acquired on /home/avl/.cache/huggingface/datasets/
↪d3b7716978cb901261e59327d43b04c52d6d29e50eeac39bea0816865a584081.
↪7c39fd6270c5ee55bcf2e4de23af77ef299e0df65be3f3e84454dcef7175844a.py.lock
```

(continues on next page)

(continued from previous page)

```
INFO:nlp.load:Found main folder for dataset https://s3.amazonaws.com/datasets.
↪huggingface.co/nlp/datasets/imdb/imdb.py at /home/avl/anaconda3/envs/detect/lib/
↪python3.7/site-packages/nlp/datasets/imdb
INFO:nlp.load:Found specific version folder for dataset https://s3.amazonaws.com/
↪datasets.huggingface.co/nlp/datasets/imdb/imdb.py at /home/avl/anaconda3/envs/
↪detect/lib/python3.7/site-packages/nlp/datasets/imdb/
↪76cdbd7249ea3548c928bbf304258dab44d09cd3638d9da8d42480d1d1be3743
INFO:nlp.load:Found script file from https://s3.amazonaws.com/datasets.huggingface.co/
↪nlp/datasets/imdb/imdb.py to /home/avl/anaconda3/envs/detect/lib/python3.7/site-
↪packages/nlp/datasets/imdb/
↪76cdbd7249ea3548c928bbf304258dab44d09cd3638d9da8d42480d1d1be3743/imdb.py
INFO:nlp.load:Found dataset infos file from https://s3.amazonaws.com/datasets.
↪huggingface.co/nlp/datasets/imdb/dataset_infos.json to /home/avl/anaconda3/envs/
↪detect/lib/python3.7/site-packages/nlp/datasets/imdb/
↪76cdbd7249ea3548c928bbf304258dab44d09cd3638d9da8d42480d1d1be3743/dataset_infos.json
INFO:nlp.load:Found metadata file for dataset https://s3.amazonaws.com/datasets.
↪huggingface.co/nlp/datasets/imdb/imdb.py at /home/avl/anaconda3/envs/detect/lib/
↪python3.7/site-packages/nlp/datasets/imdb/
↪76cdbd7249ea3548c928bbf304258dab44d09cd3638d9da8d42480d1d1be3743/imdb.json
INFO:filelock:Lock 139754813800592 released on /home/avl/.cache/huggingface/datasets/
↪d3b7716978cb901261e59327d43b04c52d6d29e50eeac39bea0816865a584081.
↪7c39fd6270c5ee55bcf2e4de23af77ef299e0df65be3f3e84454dcef7175844a.py.lock
INFO:nlp.builder:No config specified, defaulting to first: imdb/plain_text
INFO:nlp.info:Loading Dataset Infos from /home/avl/anaconda3/envs/detect/lib/python3.
↪7/site-packages/nlp/datasets/imdb/
↪76cdbd7249ea3548c928bbf304258dab44d09cd3638d9da8d42480d1d1be3743
INFO:nlp.builder:Overwrite dataset info from restored data version.
INFO:nlp.info:Loading Dataset info from /home/avl/.cache/huggingface/datasets/imdb/
↪plain_text/1.0.0/76cdbd7249ea3548c928bbf304258dab44d09cd3638d9da8d42480d1d1be3743
INFO:nlp.builder:Reusing dataset imdb (/home/avl/.cache/huggingface/datasets/imdb/
↪plain_text/1.0.0/76cdbd7249ea3548c928bbf304258dab44d09cd3638d9da8d42480d1d1be3743)
INFO:nlp.builder:Constructing Dataset for split train, test, unsupervised, from /home/
↪avl/.cache/huggingface/datasets/imdb/plain_text/1.0.0/
↪76cdbd7249ea3548c928bbf304258dab44d09cd3638d9da8d42480d1d1be3743
INFO:nlp.utils.info_utils:All the checksums matched successfully for post processing␣
↪resources
INFO:nlp.utils.info_utils:All the checksums matched successfully for post processing␣
↪resources
INFO:nlp.utils.info_utils:All the checksums matched successfully for post processing␣
↪resources
```

```
(25000,) (25000,)
```

Let's take a look at respectively a negative and positive review:

```
[5]: labels = ['Negative', 'Positive']
print(labels[y[-1]])
print(X[-1])
```

```
Negative
This is one of the dumbest films, I've ever seen. It rips off nearly ever type of␣
↪thriller and manages to make a mess of them all.<br /><br />There's not a single␣
↪good line or character in the whole mess. If there was a plot, it was an␣
↪afterthought and as far as acting goes, there's nothing good to say so Ill say␣
↪nothing. I honestly cant understand how this type of nonsense gets produced and␣
↪actually released, does somebody somewhere not at some stage think, 'Oh my god this␣
↪really is a load of shite' and call it a day. Its crap like this that has people␣
↪downloading illegally, the trailer looks like a completely different film, at least␣
↪if you have download it, you haven't wasted your time or money Don't waste your␣
↪time, this is painful.
```

(continues on next page)

(continued from previous page)

[6]:
```
print(labels[y[2]])
print(X[2])
```

```
Positive
Brilliant over-acting by Lesley Ann Warren. Best dramatic hobo lady I have ever seen,␣
↪and love scenes in clothes warehouse are second to none. The corn on face is a␣
↪classic, as good as anything in Blazing Saddles. The take on lawyers is also superb.
↪ After being accused of being a turncoat, selling out his boss, and being dishonest␣
↪the lawyer of Pepto Bolt shrugs indifferently "I'm a lawyer" he says. Three funny␣
↪words. Jeffrey Tambor, a favorite from the later Larry Sanders show, is fantastic␣
↪here too as a mad millionaire who wants to crush the ghetto. His character is more␣
↪malevolent than usual. The hospital scene, and the scene where the homeless invade␣
↪a demolition site, are all-time classics. Look for the legs scene and the two big␣
↪diggers fighting (one bleeds). This movie gets better each time I see it (which is␣
↪quite often).
```

We split the original test set in a reference dataset and a dataset which should not be rejected under the *H0* of the statistical test. We also create imbalanced datasets and inject selected words in the reference set.

[7]:
```
def random_sample(X: np.ndarray, y: np.ndarray, proba_zero: float, n: int):
    if len(y.shape) == 1:
        idx_0 = np.where(y == 0)[0]
        idx_1 = np.where(y == 1)[0]
    else:
        idx_0 = np.where(y[:, 0] == 1)[0]
        idx_1 = np.where(y[:, 1] == 1)[0]
    n_0, n_1 = int(n * proba_zero), int(n * (1 - proba_zero))
    idx_0_out = np.random.choice(idx_0, n_0, replace=False)
    idx_1_out = np.random.choice(idx_1, n_1, replace=False)
    X_out = np.concatenate([X[idx_0_out], X[idx_1_out]])
    y_out = np.concatenate([y[idx_0_out], y[idx_1_out]])
    return X_out, y_out


def padding_last(x: np.ndarray, seq_len: int) -> np.ndarray:
    try:  # try not to replace padding token
        last_token = np.where(x == 0)[0][0]
    except:  # no padding
        last_token = seq_len - 1
    return 1, last_token


def padding_first(x: np.ndarray, seq_len: int) -> np.ndarray:
    try:  # try not to replace padding token
        first_token = np.where(x == 0)[0][-1] + 2
    except:  # no padding
        first_token = 0
    return first_token, seq_len - 1


def inject_word(token: int, X: np.ndarray, perc_chg: float, padding: str = 'last'):
    seq_len = X.shape[1]
    n_chg = int(perc_chg * .01 * seq_len)
    X_cp = X.copy()
    for _ in range(X.shape[0]):
```

(continues on next page)

(continued from previous page)

```
        if padding == 'last':
            first_token, last_token = padding_last(X_cp[_, :], seq_len)
        else:
            first_token, last_token = padding_first(X_cp[_, :], seq_len)
        if last_token <= n_chg:
            choice_len = seq_len
        else:
            choice_len = last_token
        idx = np.random.choice(np.arange(first_token, choice_len), n_chg,␣
↪replace=False)
        X_cp[_, idx] = token
    return X_cp
```

Reference, *H0* and imbalanced data:

[8]:
```
# proba_zero = fraction with label 0 (=negative sentiment)
n_sample = 1000
X_ref = random_sample(X, y, proba_zero=.5, n=n_sample)[0]
X_h0 = random_sample(X, y, proba_zero=.5, n=n_sample)[0]
n_imb = [.1, .9]
X_imb = {_: random_sample(X, y, proba_zero=_, n=n_sample)[0] for _ in n_imb}
```

Inject words in reference data:

[9]:
```
words = ['fantastic', 'good', 'bad', 'horrible']
perc_chg = [1., 5.]  # % of tokens to change in an instance

words_tf = tokenizer.encode(words, return_tensors='tf')
words_tf = list(words_tf.numpy()[0, 1:-1])
max_len = 100
tokens = tokenizer.batch_encode_plus(X_ref, pad_to_max_length=True,
                                     max_length=max_len, return_tensors='tf')
X_word = {}
for i, w in enumerate(words_tf):
    X_word[words[i]] = {}
    for p in perc_chg:
        x = inject_word(w, tokens['input_ids'].numpy(), p)
        dec = tokenizer.batch_decode(x, **dict(skip_special_tokens=True))
        X_word[words[i]][p] = np.array(dec)
```

42.3.3 Preprocessing

First we need to specify the type of embedding we want to extract from the BERT model. We can extract embeddings from the . . .

- **pooler_output**: Last layer hidden-state of the first token of the sequence (classification token; CLS) further processed by a Linear layer and a Tanh activation function. The Linear layer weights are trained from the next sentence prediction (classification) objective during pre-training. **Note**: this output is usually not a good summary of the semantic content of the input, you're often better with averaging or pooling the sequence of hidden-states for the whole input sequence.
- **last_hidden_state**: Sequence of hidden states at the output of the last layer of the model, averaged over the tokens.
- **hidden_state**: Hidden states of the model at the output of each layer, averaged over the tokens.
- **hidden_state_cls**: See *hidden_state* but use the CLS token output.

If *hidden_state* or *hidden_state_cls* is used as embedding type, you also need to pass the layer numbers used to extract the embedding from. As an example we extract embeddings from the last 8 hidden states.

```
[10]: from alibi_detect.models.tensorflow import TransformerEmbedding

emb_type = 'hidden_state'
n_layers = 8
layers = [-_ for _ in range(1, n_layers + 1)]

embedding = TransformerEmbedding(model_name, emb_type, layers)
```

```
INFO:transformers.configuration_utils:loading configuration file https://s3.amazonaws.
↪com/models.huggingface.co/bert/bert-base-cased-config.json from cache at /home/avl/.
↪cache/torch/transformers/
↪b945b69218e98b3e2c95acf911789741307dec43c698d35fad11c1ae28bda352.
↪9da767be51e1327499df13488672789394e2ca38b877837e52618a67d7002391
INFO:transformers.configuration_utils:Model config BertConfig {
  "architectures": [
    "BertForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "output_hidden_states": true,
  "pad_token_id": 0,
  "type_vocab_size": 2,
  "vocab_size": 28996
}

INFO:transformers.modeling_tf_utils:loading weights file https://cdn.huggingface.co/
↪bert-base-cased-tf_model.h5 from cache at /home/avl/.cache/torch/transformers/
↪17e64dc7dc200314bc70dd8198010773501bcabb65a493c1ae7183b8c9a5b1ff.
↪908e74db1113031d6827eb22808cf370b0aeded6e6ac20d0f07af0a334e195cc.h5
INFO:transformers.modeling_tf_utils:Layers from pretrained model not used in␣
↪TFBertModel: ['mlm___cls', 'nsp___cls']
```

Let's check what an embedding looks like:

```
[11]: tokens = tokenizer.batch_encode_plus(X[:5], pad_to_max_length=True,
                                     max_length=max_len, return_tensors='tf')
x_emb = embedding(tokens)
print(x_emb.shape)
```

```
(5, 768)
```

So the BERT model's embedding space used by the drift detector consists of a 768-dimensional vector for each instance. We will therefore first apply a dimensionality reduction step with an Untrained AutoEncoder (*UAE*) before conducting the statistical hypothesis test. We use the embedding model as the input for the UAE which then projects the embedding on a lower dimensional space.

[12]:
```
tf.random.set_seed(0)
```

[14]:
```
from alibi_detect.cd.tensorflow import UAE

enc_dim = 32
shape = (x_emb.shape[1],)

uae = UAE(input_layer=embedding, shape=shape, enc_dim=enc_dim)
```

Let's test this again:

[15]:
```
emb_uae = uae(tokens)
print(emb_uae.shape)
```

```
(5, 32)
```

42.3.4 K-S detector

42.4 Initialize

We proceed to initialize the drift detector. From here on the detector works the same as for other modalities such as images. Please check the images example or the K-S detector documentation for more information about each of the possible parameters.

[16]:
```
from functools import partial
from alibi_detect.cd.tensorflow import preprocess_drift

# define preprocessing function
preprocess_fn = partial(preprocess_drift, model=uae, tokenizer=tokenizer,
                        max_len=max_len, batch_size=32)

# initialize detector
cd = KSDrift(X_ref, p_val=.05, preprocess_fn=preprocess_fn)

# we can also save/load an initialised detector
filepath = 'my_path'  # change to directory where detector is saved
save_detector(cd, filepath)
cd = load_detector(filepath)
```

42.5 Detect drift

Let's first check if drift occurs on a similar sample from the training set as the reference data.

[17]:
```
preds_h0 = cd.predict(X_h0, return_p_val=True)
labels = ['No!', 'Yes!']
print('Drift? {}'.format(labels[preds_h0['data']['is_drift']]))
print('p-value: {}'.format(preds_h0['data']['p_val']))
```

```
Drift? No!
p-value: [0.5360543  0.60991895 0.64755726 0.02246371 0.14833806 0.46576622
 0.60991895 0.6852314  0.6852314  0.64755726 0.08690542 0.01962691
 0.06155144 0.79439443 0.43243074 0.82795686 0.99870795 0.50035924
```

(continues on next page)

(continued from previous page)

```
 0.21933001 0.3699725  0.60991895 0.96887016 0.99365413 0.28769323
 0.64755726 0.9134755  0.01121108 0.82795686 0.28769323 0.79439443
 0.7590978  0.8879386 ]
```

Detect drift on imbalanced and perturbed datasets:

[18]:
```
for k, v in X_imb.items():
    preds = cd.predict(v, return_p_val=True)
    print('% negative sentiment {}'.format(k * 100))
    print('Drift? {}'.format(labels[preds['data']['is_drift']]))
    print('p-value: {}'.format(preds['data']['p_val']))
    print('')
```

```
% negative sentiment 10.0
Drift? Yes!
p-value: [3.69972497e-01 8.27956855e-01 9.68870163e-01 6.47557259e-01
 1.12110768e-02 1.22740539e-03 3.69972497e-01 2.13202584e-05
 3.40991944e-01 1.08282514e-01 7.22554982e-01 2.40603596e-01
 1.33834302e-01 2.87693232e-01 7.22554982e-01 8.27956855e-01
 8.27956855e-01 6.20218972e-03 7.22554982e-01 1.48338065e-01
 9.13475513e-01 2.87693232e-01 9.96931016e-01 5.72654784e-01
 3.13561678e-01 1.29345525e-02 8.87938619e-01 7.76214674e-02
 1.99518353e-01 3.77843790e-02 5.00359237e-01 9.35580969e-01]

% negative sentiment 90.0
Drift? No!
p-value: [0.12833405 0.5825486  0.32157195 0.9785308  0.07975528 0.12313652
 0.02028842 0.00162954 0.300002   0.17058046 0.63624656 0.43757161
 0.6726954  0.8787614  0.05583185 0.16792585 0.9815915  0.0267063
 0.11585598 0.18343964 0.8995234  0.11339176 0.5072882  0.1507392
 0.32593367 0.0034498  0.43747288 0.29798958 0.14292577 0.72975236
 0.27586988 0.90003824]
```

[19]:
```
for w, probas in X_word.items():
    for p, v in probas.items():
        preds = cd.predict(v, return_p_val=True)
        print('Word: {} -- % perturbed: {}'.format(w, p))
        print('Drift? {}'.format(labels[preds['data']['is_drift']]))
        print('p-value: {}'.format(preds['data']['p_val']))
        print('')
```

```
Word: fantastic -- % perturbed: 1.0
Drift? No!
p-value: [0.9134755  0.02565915 0.2406036  0.722555   0.34099194 0.01489316
 0.05464633 0.43243074 0.21933001 0.9999727  0.8879386  0.31356168
 0.00389581 0.93558097 0.9999727  0.99870795 0.5360543  0.996931
 0.03778438 0.64755726 0.9134755  0.14833806 0.06919032 0.722555
 0.28769323 0.18111965 0.9134755  0.9134755  0.1338343  0.01121108
 0.9134755  0.7590978 ]

Word: fantastic -- % perturbed: 5.0
Drift? Yes!
p-value: [6.07078255e-04 8.23113245e-15 1.84965307e-10 6.85231388e-01
 2.43227714e-08 2.53623026e-18 7.04859247e-08 8.37208051e-03
 2.15581372e-12 1.98871276e-04 6.15514442e-02 7.89124083e-13
 0.00000000e+00 2.40344345e-03 7.22554982e-01 5.46463318e-02
```

(continues on next page)

(continued from previous page)

```
 7.22554982e-01 8.69054198e-02 1.93049129e-25 1.08282514e-01
 2.13202584e-05 7.05219168e-14 1.91063212e-21 1.96269080e-02
 8.36122004e-23 2.87940366e-11 1.33834302e-01 6.47557259e-01
 3.34610052e-37 1.04535818e-26 1.11478073e-06 2.82894098e-03]

Word: good -- % perturbed: 1.0
Drift? Yes!
p-value: [1.8111965e-01 9.6887016e-01 6.1551444e-02 9.9954331e-01 2.1933001e-01
 9.9365413e-01 6.0991895e-01 9.3558097e-01 8.5929435e-01 9.3558097e-01
 9.8016179e-01 8.8793862e-01 4.8418805e-02 9.9954331e-01 9.5405817e-01
 4.0047103e-01 9.9987090e-01 3.6997250e-01 9.9870795e-01 7.7621467e-02
 7.5909781e-01 5.7265478e-01 4.0047103e-01 7.5909781e-01 9.1347551e-01
 9.9693102e-01 9.8826110e-01 9.9987090e-01 4.3243074e-01 9.0799862e-05
 9.9870795e-01 9.1347551e-01]

Word: good -- % perturbed: 5.0
Drift? Yes!
p-value: [4.1416480e-17 6.0991895e-01 3.8025766e-18 9.6887016e-01 1.9746931e-09
 5.0035924e-01 1.1079235e-04 3.4099194e-01 1.1211077e-02 1.8548947e-08
 9.6887016e-01 6.2021897e-03 7.3622059e-26 1.1211077e-02 1.3383430e-01
 2.0971582e-11 6.0991895e-01 1.5243730e-11 8.5929435e-01 7.0521917e-14
 6.0991895e-01 3.1830119e-08 1.7943768e-06 1.2274054e-03 4.2185336e-04
 2.8769323e-01 6.2021897e-03 7.9439443e-01 1.0030026e-10 0.0000000e+00
 6.2021897e-03 6.0991895e-01]

Word: bad -- % perturbed: 1.0
Drift? No!
p-value: [0.60991895 0.5360543  0.1338343  0.9882611  0.6852314  0.82795686
 0.996931   0.93558097 0.722555   0.43243074 0.9801618  0.9540582
 0.00721313 0.96887016 0.9998709  0.79439443 0.26338065 0.02565915
 0.9540582  0.18111965 0.31356168 0.50035924 0.10828251 0.85929435
 0.996931   0.99870795 0.96887016 0.9801618  0.96887016 0.31356168
 0.02565915 0.996931  ]

Word: bad -- % perturbed: 5.0
Drift? Yes!
p-value: [6.14975981e-09 2.63112887e-09 8.23113245e-15 1.48338065e-01
 4.93855441e-05 2.87693232e-01 6.47557259e-01 1.48338065e-01
 1.63965786e-04 2.73716728e-15 4.32430744e-01 6.87054069e-07
 4.91627349e-40 3.27475419e-07 7.76214674e-02 2.92505771e-02
 6.87054069e-07 6.05478631e-20 3.13561678e-01 7.04859247e-08
 5.02173026e-25 1.53302494e-07 3.24872937e-19 1.29345525e-02
 3.13561678e-01 9.80161786e-01 2.63380647e-01 2.03786441e-03
 2.00923371e-13 1.88342838e-17 2.53623026e-18 8.27956855e-01]

Word: horrible -- % perturbed: 1.0
Drift? Yes!
p-value: [0.14833806 0.996931   0.99365413 0.9995433  0.5726548  0.79439443
 0.9998709  0.85929435 0.00721313 0.9995433  0.9999727  0.5360543
 0.06155144 0.9998709  0.9540582  0.996931   0.09710453 0.31356168
 0.7590978  0.26338065 0.40047103 0.93558097 0.19951835 0.9882611
 0.9999727  0.04841881 0.8879386  0.99870795 0.9801618  0.00145631
 0.04841881 0.93558097]

Word: horrible -- % perturbed: 5.0
Drift? Yes!
p-value: [9.03489017e-17 4.65766221e-01 2.19330013e-01 8.27956855e-01
```

(continues on next page)

(continued from previous page)

```
 1.22740539e-03 1.71140861e-02 1.08282514e-01 1.45630504e-03
 3.73327202e-31 1.34916729e-04 3.69972497e-01 1.10581561e-11
 3.37775260e-36 1.08282514e-01 7.26078229e-04 1.81119651e-01
 3.49877549e-09 3.50604125e-04 5.39559682e-11 4.17105196e-12
 7.49975329e-32 1.38413116e-05 1.26629300e-17 6.09918952e-01
 7.04859247e-08 1.18333705e-14 7.22554982e-01 1.72444014e-03
 1.69780876e-14 0.00000000e+00 2.14098059e-19 3.32780443e-02]
```

42.5.1 MMD TensorFlow detector

42.6 Initialize

Again check the images example or the MMD detector documentation for more information about each of the possible parameters.

```
[20]: cd = MMDDrift(X_ref, p_val=.05, preprocess_fn=preprocess_fn, n_permutations=100)
```

42.7 Detect drift

H0:

```
[21]: preds_h0 = cd.predict(X_h0)
      labels = ['No!', 'Yes!']
      print('Drift? {}'.format(labels[preds_h0['data']['is_drift']]))
      print('p-value: {}'.format(preds_h0['data']['p_val']))
```

```
Drift? No!
p-value: 0.38
```

Imbalanced data:

```
[22]: for k, v in X_imb.items():
          preds = cd.predict(v)
          print('% negative sentiment {}'.format(k * 100))
          print('Drift? {}'.format(labels[preds['data']['is_drift']]))
          print('p-value: {}'.format(preds['data']['p_val']))
          print('')
```

```
% negative sentiment 10.0
Drift? Yes!
p-value: 0.0

% negative sentiment 90.0
Drift? Yes!
p-value: 0.0
```

Perturbed data:

[23]:
```
for w, probas in X_word.items():
    for p, v in probas.items():
        preds = cd.predict(v)
        print('Word: {} -- % perturbed: {}'.format(w, p))
        print('Drift? {}'.format(labels[preds['data']['is_drift']]))
        print('p-value: {}'.format(preds['data']['p_val']))
        print('')
```

```
Word: fantastic -- % perturbed: 1.0
Drift? Yes!
p-value: 0.01

Word: fantastic -- % perturbed: 5.0
Drift? Yes!
p-value: 0.0

Word: good -- % perturbed: 1.0
Drift? No!
p-value: 0.53

Word: good -- % perturbed: 5.0
Drift? Yes!
p-value: 0.0

Word: bad -- % perturbed: 1.0
Drift? No!
p-value: 0.44

Word: bad -- % perturbed: 5.0
Drift? Yes!
p-value: 0.0

Word: horrible -- % perturbed: 1.0
Drift? No!
p-value: 0.2

Word: horrible -- % perturbed: 5.0
Drift? Yes!
p-value: 0.0
```

42.7.1 MMD PyTorch detector

42.7.2 Initialize

We can run the same detector with *PyTorch* backend for both the preprocessing step and MMD implementation:

[24]:
```
import torch
import torch.nn as nn

# set random seed and device
seed = 0
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
```

(continues on next page)

(continued from previous page)

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda
```

[49]:
```
from alibi_detect.cd.pytorch import preprocess_drift
from alibi_detect.models.pytorch import TransformerEmbedding

embedding_pt = TransformerEmbedding(model_name, emb_type, layers)

model = nn.Sequential(
    embedding_pt,
    nn.Linear(768, 256),
    nn.ReLU(),
    nn.Linear(256, enc_dim)
).to(device).eval()

# define preprocessing function
preprocess_fn = partial(preprocess_drift, model=model, tokenizer=tokenizer,
                        max_len=max_len, batch_size=32)

# initialise drift detector
cd = MMDDrift(X_ref, backend='pytorch', p_val=.05,
              preprocess_fn=preprocess_fn, n_permutations=100)
```

```
INFO:transformers.configuration_utils:loading configuration file https://s3.amazonaws.
↪com/models.huggingface.co/bert/bert-base-cased-config.json from cache at /home/avl/.
↪cache/torch/transformers/
↪b945b69218e98b3e2c95acf911789741307dec43c698d35fad11c1ae28bda352.
↪9da767be51e1327499df13488672789394e2ca38b877837e52618a67d7002391
INFO:transformers.configuration_utils:Model config BertConfig {
  "architectures": [
    "BertForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "output_hidden_states": true,
  "pad_token_id": 0,
  "type_vocab_size": 2,
  "vocab_size": 28996
}

INFO:transformers.modeling_utils:loading weights file https://cdn.huggingface.co/bert-
↪base-cased-pytorch_model.bin from cache at /home/avl/.cache/torch/transformers/
↪d8f11f061e407be64c4d5d7867ee61d1465263e24085cfa26abf183fdc830569.
↪3fadbea36527ae472139fe84cddaa65454d7429f12d543d80bfc3ad70de55ac2
```

42.8 Detect drift

H0:

```
[50]: preds_h0 = cd.predict(X_h0)
labels = ['No!', 'Yes!']
print('Drift? {}'.format(labels[preds_h0['data']['is_drift']]))
print('p-value: {}'.format(preds_h0['data']['p_val']))
```

```
Drift? No!
p-value: 0.07999999821186066
```

Imbalanced data:

```
[51]: for k, v in X_imb.items():
    preds = cd.predict(v)
    print('% negative sentiment {}'.format(k * 100))
    print('Drift? {}'.format(labels[preds['data']['is_drift']]))
    print('p-value: {}'.format(preds['data']['p_val']))
    print('')
```

```
% negative sentiment 10.0
Drift? Yes!
p-value: 0.0

% negative sentiment 90.0
Drift? Yes!
p-value: 0.0
```

Perturbed data:

```
[52]: for w, probas in X_word.items():
    for p, v in probas.items():
        preds = cd.predict(v)
        print('Word: {} -- % perturbed: {}'.format(w, p))
        print('Drift? {}'.format(labels[preds['data']['is_drift']]))
        print('p-value: {}'.format(preds['data']['p_val']))
        print('')
```

```
Word: fantastic -- % perturbed: 1.0
Drift? Yes!
p-value: 0.0

Word: fantastic -- % perturbed: 5.0
Drift? Yes!
p-value: 0.0

Word: good -- % perturbed: 1.0
Drift? No!
p-value: 0.12999999523162842

Word: good -- % perturbed: 5.0
Drift? Yes!
p-value: 0.0

Word: bad -- % perturbed: 1.0
Drift? Yes!
```

(continues on next page)

(continued from previous page)

```
p-value: 0.0

Word: bad -- % perturbed: 5.0
Drift? Yes!
p-value: 0.0

Word: horrible -- % perturbed: 1.0
Drift? Yes!
p-value: 0.019999999552965164

Word: horrible -- % perturbed: 5.0
Drift? Yes!
p-value: 0.0
```

42.8.1 Train embeddings from scratch

So far we used pre-trained embeddings from a BERT model. We can however also use embeddings from a model trained from scratch. First we define and train a simple classification model consisting of an embedding and LSTM layer in *TensorFlow*.

42.9 Load data and train model

[53]:
```
from tensorflow.keras.datasets import imdb, reuters
from tensorflow.keras.layers import Dense, Embedding, Input, LSTM
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.utils import to_categorical

INDEX_FROM = 3
NUM_WORDS = 10000


def print_sentence(tokenized_sentence: str, id2w: dict):
    print(' '.join(id2w[_] for _ in tokenized_sentence))
    print('')
    print(tokenized_sentence)


def mapping_word_id(data):
    w2id = data.get_word_index()
    w2id = {k: (v + INDEX_FROM) for k, v in w2id.items()}
    w2id["<PAD>"] = 0
    w2id["<START>"] = 1
    w2id["<UNK>"] = 2
    w2id["<UNUSED>"] = 3
    id2w = {v: k for k, v in w2id.items()}
    return w2id, id2w


def get_dataset(dataset: str = 'imdb', max_len: int = 100):
    if dataset == 'imdb':
        data = imdb
```

(continues on next page)

(continued from previous page)

```
    elif dataset == 'reuters':
        data = reuters
    else:
        raise NotImplementedError

    w2id, id2w = mapping_word_id(data)

    (X_train, y_train), (X_test, y_test) = data.load_data(
        num_words=NUM_WORDS, index_from=INDEX_FROM)
    X_train = sequence.pad_sequences(X_train, maxlen=max_len)
    X_test = sequence.pad_sequences(X_test, maxlen=max_len)
    y_train, y_test = to_categorical(y_train), to_categorical(y_test)

    return (X_train, y_train), (X_test, y_test), (w2id, id2w)


def imdb_model(X: np.ndarray, num_words: int = 100, emb_dim: int = 128,
               lstm_dim: int = 128, output_dim: int = 2) -> tf.keras.Model:
    inputs = Input(shape=(X.shape[1:]), dtype=tf.float32)
    x = Embedding(num_words, emb_dim)(inputs)
    x = LSTM(lstm_dim, dropout=.5)(x)
    outputs = Dense(output_dim, activation=tf.nn.softmax)(x)
    model = tf.keras.Model(inputs=inputs, outputs=outputs)
    model.compile(
        loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy']
    )
    return model
```

Load and tokenize data:

```
[54]: (X_train, y_train), (X_test, y_test), (word2token, token2word) = \
          get_dataset(dataset='imdb', max_len=max_len)
```

Let's check out an instance:

```
[55]: print_sentence(X_train[0], token2word)
```

```
cry at a film it must have been good and this definitely was also <UNK> to the two␣
↪little boy's that played the <UNK> of norman and paul they were just brilliant␣
↪children are often left out of the <UNK> list i think because the stars that play␣
↪them all grown up are such a big profile for the whole film but these children are␣
↪amazing and should be praised for what they have done don't you think the whole␣
↪story was so lovely because it was true and was someone's life after all that was␣
↪shared with us all

[1415   33    6   22   12  215   28   77   52    5   14  407   16   82
    2    8    4  107  117 5952   15  256    4    2    7 3766    5  723
   36   71   43  530  476   26  400  317   46    7    4    2 1029   13
  104   88    4  381   15  297   98   32 2071   56   26  141    6  194
 7486   18    4  226   22   21  134  476   26  480    5  144   30 5535
   18   51   36   28  224   92   25  104    4  226   65   16   38 1334
   88   12   16  283    5   16 4472  113  103   32   15   16 5345   19
  178   32]
```

Define and train a simple model:

[56]:
```
model = imdb_model(X=X_train, num_words=NUM_WORDS, emb_dim=256, lstm_dim=128, output_
↪dim=2)
model.fit(X_train, y_train, batch_size=32, epochs=2,
          shuffle=True, validation_data=(X_test, y_test))
```

```
Train on 25000 samples, validate on 25000 samples
Epoch 1/2
25000/25000 [==============================] - 22s 885us/sample - loss: 0.4201 -␣
↪accuracy: 0.8042 - val_loss: 0.3328 - val_accuracy: 0.8525
Epoch 2/2
25000/25000 [==============================] - 20s 818us/sample - loss: 0.2720 -␣
↪accuracy: 0.8890 - val_loss: 0.3373 - val_accuracy: 0.8547
```

[56]:
```
<tensorflow.python.keras.callbacks.History at 0x7f19e867f990>
```

Extract the embedding layer from the trained model and combine with UAE preprocessing step:

[57]:
```
embedding = tf.keras.Model(inputs=model.inputs, outputs=model.layers[1].output)
x_emb = embedding(X_train[:5])
print(x_emb.shape)
```

```
(5, 100, 256)
```

[58]:
```
tf.random.set_seed(0)

shape = tuple(x_emb.shape[1:])
uae = UAE(input_layer=embedding, shape=shape, enc_dim=enc_dim)
```

Again, create reference, *H0* and perturbed datasets. Also test against the *Reuters* news topic classification dataset.

[59]:
```
X_ref, y_ref = random_sample(X_test, y_test, proba_zero=.5, n=n_sample)
X_h0, y_h0 = random_sample(X_test, y_test, proba_zero=.5, n=n_sample)
tokens = [word2token[w] for w in words]
X_word = {}
for i, t in enumerate(tokens):
    X_word[words[i]] = {}
    for p in perc_chg:
        X_word[words[i]][p] = inject_word(t, X_ref, p, padding='first')
```

[60]:
```
# load and tokenize Reuters dataset
(X_reut, y_reut), (w2t_reut, t2w_reut) = \
    get_dataset(dataset='reuters', max_len=max_len)[1:]

# sample random instances
idx = np.random.choice(X_reut.shape[0], n_sample, replace=False)
X_ood = X_reut[idx]
```

42.10 Initialize detector and detect drift

[61]:
```
from alibi_detect.cd.tensorflow import preprocess_drift

# define preprocessing function
preprocess_fn = partial(preprocess_drift, model=uae, batch_size=128)

# initialize detector
cd = KSDrift(X_ref, p_val=.05, preprocess_fn=preprocess_fn)
```

H0:

[62]:
```
preds_h0 = cd.predict(X_h0)
labels = ['No!', 'Yes!']
print('Drift? {}'.format(labels[preds_h0['data']['is_drift']]))
print('p-value: {}'.format(preds_h0['data']['p_val']))
```

```
Drift? No!
p-value: [0.5360543  0.18111965 0.99365413 0.10828251 0.64755726 0.9540582
 0.79439443 0.31356168 0.85929435 0.2406036  0.8879386  0.50035924
 0.50035924 0.43243074 0.7590978  0.28769323 0.31356168 0.82795686
 0.40047103 0.09710453 0.04281518 0.8879386  0.5726548  0.8879386
 0.06919032 0.722555   0.43243074 0.07762147 0.18111965 0.60991895
 0.5726548  0.40047103]
```

Perturbed data:

[63]:
```
for w, probas in X_word.items():
    for p, v in probas.items():
        preds = cd.predict(v)
        print('Word: {} -- % perturbed: {}'.format(w, p))
        print('Drift? {}'.format(labels[preds['data']['is_drift']]))
        print('p-value: {}'.format(preds['data']['p_val']))
        print('')
```

```
Word: fantastic -- % perturbed: 1.0
Drift? No!
p-value: [0.9801618  0.96887016 0.9540582  0.82795686 0.996931   0.7590978
 0.85929435 0.96887016 0.31356168 0.8879386  0.722555   0.9134755
 0.82795686 0.93558097 0.9801618  0.6852314  0.5726548  0.9882611
 0.99365413 0.31356168 0.9540582  0.96887016 0.9998709  0.96887016
 0.82795686 0.9801618  0.82795686 0.996931   0.85929435 0.82795686
 0.40047103 0.9801618 ]

Word: fantastic -- % perturbed: 5.0
Drift? Yes!
p-value: [1.08282514e-01 7.94394433e-01 5.00359237e-01 1.08282514e-01
 3.69972497e-01 2.90806405e-04 9.69783217e-03 3.32780443e-02
 2.50427207e-10 8.87938619e-01 9.13475513e-01 7.76214674e-02
 1.96269080e-02 7.94394433e-01 3.89581337e-03 1.81119651e-01
 3.60981949e-06 5.36054313e-01 4.65766221e-01 7.13247118e-06
 7.22554982e-01 4.00471032e-01 9.35580969e-01 9.71045271e-02
 2.87693232e-01 5.00359237e-01 6.15514442e-02 4.00471032e-01
 2.40603596e-01 5.72654784e-01 2.63380647e-01 7.76214674e-02]

Word: good -- % perturbed: 1.0
Drift? No!
p-value: [0.9995433  0.996931   0.9801618  0.7590978  0.96887016 0.9882611
```

(continues on next page)

(continued from previous page)

```
 0.99365413 0.85929435 0.9134755  0.99365413 0.99365413 0.96887016
 0.9801618  0.996931   0.9540582  0.5360543  0.9882611  0.9882611
 0.9882611  0.9995433  0.96887016 0.9540582  0.99365413 0.96887016
 0.9801618  0.9995433  0.99365413 0.99870795 0.82795686 0.9801618
 0.9134755  0.996931  ]

Word: good -- % perturbed: 5.0
Drift? No!
p-value: [0.93558097 0.46576622 0.99365413 0.5360543  0.93558097 0.6852314
 0.82795686 0.9882611  0.64755726 0.6852314  0.96887016 0.7590978
 0.9801618  0.82795686 0.50035924 0.34099194 0.18111965 0.9801618
 0.64755726 0.96887016 0.6852314  0.8879386  0.8879386  0.82795686
 0.3699725  0.64755726 0.50035924 0.5726548  0.28769323 0.96887016
 0.40047103 0.93558097]

Word: bad -- % perturbed: 1.0
Drift? No!
p-value: [0.9134755  0.9134755  0.43243074 0.9134755  0.99870795 0.82795686
 0.99870795 0.9134755  0.85929435 0.5726548  0.82795686 0.82795686
 0.34099194 0.85929435 0.82795686 0.93558097 0.93558097 0.50035924
 0.5726548  0.85929435 0.9882611  0.96887016 0.9134755  0.79439443
 0.6852314  0.99999607 0.996931   0.93558097 0.9882611  0.9995433
 0.85929435 0.9134755 ]

Word: bad -- % perturbed: 5.0
Drift? Yes!
p-value: [1.20504074e-01 5.72654784e-01 5.00359237e-01 6.85231388e-01
 6.85231388e-01 1.99518353e-01 2.40603596e-01 4.00471032e-01
 7.94394433e-01 3.60981949e-06 6.15514442e-02 1.33834302e-01
 3.50604125e-04 6.47557259e-01 1.72444014e-03 1.45630504e-03
 1.29345525e-02 3.89581337e-03 2.87693232e-01 4.32430744e-01
 4.65766221e-01 5.00359237e-01 6.09918952e-01 3.13561678e-01
 9.69783217e-03 3.40991944e-01 7.22554982e-01 2.40603596e-01
 6.91903234e-02 2.40603596e-01 7.22554982e-01 4.32430744e-01]

Word: horrible -- % perturbed: 1.0
Drift? No!
p-value: [0.8879386  0.9134755  0.7590978  0.99365413 0.99365413 0.21933001
 0.79439443 0.82795686 0.79439443 0.1338343  0.82795686 0.722555
 0.1338343  0.6852314  0.64755726 0.2406036  0.64755726 0.5360543
 0.3699725  0.96887016 0.9882611  0.9134755  0.7590978  0.6852314
 0.7590978  0.82795686 0.8879386  0.18111965 0.46576622 0.79439443
 0.5726548  0.5360543 ]

Word: horrible -- % perturbed: 5.0
Drift? Yes!
p-value: [4.28151786e-02 1.11478073e-06 1.33834302e-01 1.29345525e-02
 2.19330013e-01 2.09715821e-11 3.32311448e-03 7.21312594e-03
 5.46463318e-02 6.14106432e-10 1.29345525e-02 1.08282514e-01
 2.53623026e-18 6.07078255e-04 3.25786677e-05 2.86525619e-06
 1.08282514e-01 3.18301190e-08 5.32228360e-03 1.64079204e-01
 2.87693232e-01 1.99518353e-01 6.20218972e-03 1.53302494e-07
 5.37760343e-07 2.03786441e-03 1.18559271e-07 7.05219168e-14
 1.97830971e-07 5.46463318e-02 1.11190266e-05 3.60981949e-06]
```

The detector is not as sensitive as the Transformer-based K-S drift detector. The embeddings trained from scratch only trained on a small dataset and a simple model with cross-entropy loss function for 2 epochs. The pre-trained BERT model on the other hand captures semantics of the data better.

Sample from the Reuters dataset:

[64]:
```
preds_ood = cd.predict(X_ood)
labels = ['No!', 'Yes!']
print('Drift? {}'.format(labels[preds_ood['data']['is_drift']]))
print('p-value: {}'.format(preds_ood['data']['p_val']))
```

```
Drift? Yes!
p-value: [4.56308130e-10 8.24822544e-10 4.01514189e-05 2.43227714e-08
 5.37760343e-07 8.69054198e-02 1.48338065e-01 5.00359237e-01
 3.50604125e-04 6.15514442e-02 1.00300261e-10 1.71140861e-02
 6.15514442e-02 1.79437677e-06 6.47557259e-01 3.32311448e-03
 2.56591532e-02 1.69780876e-14 1.71140861e-02 7.22554982e-01
 2.63380647e-01 8.37208051e-03 6.14975981e-09 8.24822544e-10
 1.81119651e-01 7.21312594e-03 9.69783217e-03 2.19330013e-01
 1.18559271e-07 1.20504074e-01 6.47557259e-01 1.97830971e-07]
```

CHAPTER

FORTYTHREE

CLASSIFIER DRIFT DETECTOR ON CIFAR-10

43.1 Method

The classifier-based drift detector simply tries to correctly distinguish instances from the reference data vs. the test set. The classifier is trained to output the probability that a given instance belongs to the test set. If the probabilities it assigns to unseen tests instances are significantly higher (as determined by a Kolmogorov-Smirnoff test) to those it assigns to unseen reference instances then the test set must differ from the reference set and drift is flagged. To leverage all the available reference and test data, stratified cross-validation can be applied and the out-of-fold predictions are used for the significance test. Note that a new classifier is trained for each test set or even each fold within the test set.

43.2 Backend

The method works with both the **PyTorch** and **TensorFlow** frameworks. Alibi Detect does however not install PyTorch for you. Check the PyTorch docs how to do this.

43.3 Dataset

CIFAR10 consists of 60,000 32 by 32 RGB images equally distributed over 10 classes. We evaluate the drift detector on the CIFAR-10-C dataset (Hendrycks & Dietterich, 2019). The instances in CIFAR-10-C have been corrupted and perturbed by various types of noise, blur, brightness etc. at different levels of severity, leading to a gradual decline in the classification model performance. We also check for drift against the original test set with class imbalances.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

from alibi_detect.cd import ClassifierDrift
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.datasets import fetch_cifar10c, corruption_types_cifar10c
```

```
Importing plotly failed. Interactive plots will not work.
```

43.4 Load data

Original CIFAR-10 data:

```
[2]: (X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
y_train = y_train.astype('int64').reshape(-1,)
y_test = y_test.astype('int64').reshape(-1,)
```

For CIFAR-10-C, we can select from the following corruption types at 5 severity levels:

```
[3]: corruptions = corruption_types_cifar10c()
print(corruptions)
```

```
['brightness', 'contrast', 'defocus_blur', 'elastic_transform', 'fog', 'frost',
↪'gaussian_blur', 'gaussian_noise', 'glass_blur', 'impulse_noise', 'jpeg_compression
↪', 'motion_blur', 'pixelate', 'saturate', 'shot_noise', 'snow', 'spatter', 'speckle_
↪noise', 'zoom_blur']
```

Let's pick a subset of the corruptions at corruption level 5. Each corruption type consists of perturbations on all of the original test set images.

```
[4]: corruption = ['gaussian_noise', 'motion_blur', 'brightness', 'pixelate']
X_corr, y_corr = fetch_cifar10c(corruption=corruption, severity=5, return_X_y=True)
X_corr = X_corr.astype('float32') / 255
```

We split the original test set in a reference dataset and a dataset which should not be flagged as drift. We also split the corrupted data by corruption type:

```
[5]: np.random.seed(0)
n_test = X_test.shape[0]
idx = np.random.choice(n_test, size=n_test // 2, replace=False)
idx_h0 = np.delete(np.arange(n_test), idx, axis=0)
X_ref,y_ref = X_test[idx], y_test[idx]
X_h0, y_h0 = X_test[idx_h0], y_test[idx_h0]
print(X_ref.shape, X_h0.shape)
```

```
(5000, 32, 32, 3) (5000, 32, 32, 3)
```

```
[6]: n_corr = len(corruption)
X_c = [X_corr[i * n_test:(i + 1) * n_test] for i in range(n_corr)]
```

We can visualise the same instance for each corruption type:

```
[7]: i = 6

n_test = X_test.shape[0]
plt.title('Original')
plt.axis('off')
plt.imshow(X_test[i])
plt.show()
for _ in range(len(corruption)):
    plt.title(corruption[_])
    plt.axis('off')
    plt.imshow(X_corr[n_test * _+ i])
    plt.show()
```

Original

gaussian_noise

motion_blur

43.5 Detect drift with a TensorFlow classifier

43.5.1 Single fold

We use a simple classification model and try to distinguish between the reference data and the corrupted test sets. The detector defaults to `binarize=False` which means a Kolmogorov-Smirnoff test will be used to test for significant disparity between continuous model predictions (e.g. probabilities or logits). Initially we'll test at a significance level of $p = 0.05$, use 75% of the shuffled reference and test data for training and evaluate the detector on the remaining 25%. We only train for 1 epoch.

[8]:
```
from tensorflow.keras.layers import Conv2D, Dense, Flatten, Input

tf.random.set_seed(0)

model = tf.keras.Sequential(
  [
      Input(shape=(32, 32, 3)),
      Conv2D(8, 4, strides=2, padding='same', activation=tf.nn.relu),
      Conv2D(16, 4, strides=2, padding='same', activation=tf.nn.relu),
```
(continues on next page)

(continued from previous page)

```
        Conv2D(32, 4, strides=2, padding='same', activation=tf.nn.relu),
        Flatten(),
        Dense(2, activation='softmax')
    ]
)

cd = ClassifierDrift(X_ref, model, p_val=.05, train_size=.75, epochs=1)

# we can also save/load an initialised detector
filepath = 'my_path'  # change to directory where detector is saved
save_detector(cd, filepath)
cd = load_detector(filepath)
```

```
No model found in my_path/model.
WARNING:tensorflow:No training configuration found in the save file, so the model was␣
↪*not* compiled. Compile it manually.
```

Let's check whether the detector thinks drift occurred on the different test sets and time the prediction calls:

[9]:
```
from timeit import default_timer as timer

labels = ['No!', 'Yes!']

def make_predictions(cd, x_h0, x_corr, corruption):
    t = timer()
    preds = cd.predict(x_h0)
    dt = timer() - t
    print('No corruption')
    print('Drift? {}'.format(labels[preds['data']['is_drift']]))
    print(f'p-value: {preds["data"]["p_val"]:.3f}')
    print(f'Time (s) {dt:.3f}')

    if isinstance(x_corr, list):
        for x, c in zip(x_corr, corruption):
            t = timer()
            preds = cd.predict(x)
            dt = timer() - t
            print('')
            print(f'Corruption type: {c}')
            print('Drift? {}'.format(labels[preds['data']['is_drift']]))
            print(f'p-value: {preds["data"]["p_val"]:.3f}')
            print(f'Time (s) {dt:.3f}')
```

[10]:
```
make_predictions(cd, X_h0, X_c, corruption)
```

```
No corruption
Drift? No!
p-value: 0.556
Time (s) 2.685

Corruption type: gaussian_noise
Drift? Yes!
p-value: 0.000
Time (s) 1.847

Corruption type: motion_blur
Drift? Yes!
```

(continues on next page)

(continued from previous page)

```
p-value: 0.000
Time (s) 1.671

Corruption type: brightness
Drift? Yes!
p-value: 0.000
Time (s) 1.919

Corruption type: pixelate
Drift? Yes!
p-value: 0.000
Time (s) 1.640
```

As expected, drift was only detected on the corrupted datasets and the classifier could easily distinguish the corrupted from the reference data.

43.5.2 Use all the available data via cross-validation

So far we've only used 25% of the data to detect the drift since 75% is used for training purposes. At the cost of additional training time we can however leverage all the data via stratified cross-validation. We just need to set the number of folds and keep everything else the same. So for each test set `n_folds` models are trained, and the out-of-fold predictions combined for the significance test:

[11]:
```
cd = ClassifierDrift(X_ref, model, p_val=.05, n_folds=5, epochs=1)
```

```
Both `n_folds` and `train_size` specified. By default `n_folds` is used.
```

[12]:
```
make_predictions(cd, X_h0, X_c, corruption)
```

```
No corruption
Drift? No!
p-value: 0.475
Time (s) 6.485

Corruption type: gaussian_noise
Drift? Yes!
p-value: 0.000
Time (s) 8.901

Corruption type: motion_blur
Drift? Yes!
p-value: 0.000
Time (s) 8.559

Corruption type: brightness
Drift? Yes!
p-value: 0.000
Time (s) 7.764

Corruption type: pixelate
Drift? Yes!
p-value: 0.000
Time (s) 8.476
```

43.6 Detect drift with PyTorch classifier

We can do the same with a *PyTorch* instead of a *TensorFlow* model:

[13]:
```
import torch
import torch.nn as nn

# set random seed and device
seed = 0
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# define classifier model
model = nn.Sequential(
    nn.Conv2d(3, 8, 4, stride=2, padding=0),
    nn.ReLU(),
    nn.Conv2d(8, 16, 4, stride=2, padding=0),
    nn.ReLU(),
    nn.Conv2d(16, 32, 4, stride=2, padding=0),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(128, 2)
)
```

Since our *PyTorch* encoder expects the images in a *(batch size, channels, height, width)* format, we transpose the data. Note that this step could also be passed to the drift detector via the *preprocess_fn* kwarg:

[14]:
```
def permute_c(x):
    return np.transpose(x.astype(np.float32), (0, 3, 1, 2))

X_ref_pt = permute_c(X_ref)
X_h0_pt = permute_c(X_h0)
X_c_pt = [permute_c(xc) for xc in X_c]
print(X_ref_pt.shape, X_h0_pt.shape, X_c_pt[0].shape)
```

```
(5000, 3, 32, 32) (5000, 3, 32, 32) (10000, 3, 32, 32)
```

[15]:
```
# we again use the cross-validation approach
cd = ClassifierDrift(X_ref_pt, model, backend='pytorch', p_val=.05, n_folds=5,␣
↪epochs=1)
```

```
Both `n_folds` and `train_size` specified. By default `n_folds` is used.
No GPU detected, fall back on CPU.
```

[16]:
```
make_predictions(cd, X_h0_pt, X_c_pt, corruption)
```

```
No corruption
Drift? No!
p-value: 1.000
Time (s) 4.715

Corruption type: gaussian_noise
Drift? Yes!
p-value: 0.000
Time (s) 5.863

Corruption type: motion_blur
```

(continues on next page)

(continued from previous page)

```
Drift? Yes!
p-value: 0.000
Time (s) 5.504

Corruption type: brightness
Drift? Yes!
p-value: 0.000
Time (s) 5.186

Corruption type: pixelate
Drift? Yes!
p-value: 0.000
Time (s) 4.817
```

CHAPTER

FORTYFOUR

MODEL UNCERTAINTY BASED DRIFT DETECTION ON CIFAR-10 AND WINE-QUALITY DATASETS

44.1 Method

Model-uncertainty drift detectors aim to directly detect drift that's likely to effect the performance of a model of interest. The approach is to test for change in the number of instances falling into regions of the input space on which the model is uncertain in its predictions. For each instance in the reference set the detector obtains the model's prediction and some associated notion of uncertainty. For example for a classifier this may be the entropy of the predicted label probabilities or for a regressor with dropout layers dropout Monte Carlo can be used to provide a notion of uncertainty. The same is done for the test set and if significant differences in uncertainty are detected (via a Kolmogorov-Smirnoff test) then drift is flagged.

It is important that the detector uses a reference set that is disjoint from the model's training set (on which the model's confidence may be higher).

44.2 Backend

For models that require batch evaluation both **PyTorch** and **TensorFlow** frameworks are supported. Alibi Detect does however not install PyTorch for you. Check the PyTorch docs how to do this.

44.3 Classifier uncertainty based drift detection

We start by demonstrating how to leverage model uncertainty to detect malicious drift when the model of interest is a classifer.

44.3.1 Dataset

CIFAR10 consists of 60,000 32 by 32 RGB images equally distributed over 10 classes. We evaluate the drift detector on the CIFAR-10-C dataset (Hendrycks & Dietterich, 2019). The instances in CIFAR-10-C have been corrupted and perturbed by various types of noise, blur, brightness etc. at different levels of severity, leading to a gradual decline in the classification model performance. We also check for drift against the original test set with class imbalances.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import os
import tensorflow as tf
```

(continues on next page)

(continued from previous page)

```
import torch
from torch import nn

from alibi_detect.cd import ClassifierUncertaintyDrift, RegressorUncertaintyDrift
from alibi_detect.models.tensorflow.resnet import scale_by_instance
from alibi_detect.utils.fetching import fetch_tf_model, fetch_detector
from alibi_detect.utils.saving import save_detector, load_detector
from alibi_detect.datasets import fetch_cifar10c, corruption_types_cifar10c
from alibi_detect.models.pytorch.trainer import trainer
from alibi_detect.cd.utils import encompass_batching
```

Original CIFAR-10 data:

```
[2]: (X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
y_train = y_train.astype('int64').reshape(-1,)
y_test = y_test.astype('int64').reshape(-1,)
```

For CIFAR-10-C, we can select from the following corruption types at 5 severity levels:

```
[3]: corruptions = corruption_types_cifar10c()
print(corruptions)
```

```
['brightness', 'contrast', 'defocus_blur', 'elastic_transform', 'fog', 'frost',
↪'gaussian_blur', 'gaussian_noise', 'glass_blur', 'impulse_noise', 'jpeg_compression
↪', 'motion_blur', 'pixelate', 'saturate', 'shot_noise', 'snow', 'spatter', 'speckle_
↪noise', 'zoom_blur']
```

Let's pick a subset of the corruptions at corruption level 5. Each corruption type consists of perturbations on all of the original test set images.

```
[4]: corruption = ['gaussian_noise', 'motion_blur', 'brightness', 'pixelate']
X_corr, y_corr = fetch_cifar10c(corruption=corruption, severity=5, return_X_y=True)
X_corr = X_corr.astype('float32') / 255
```

We split the original test set in a reference dataset and a dataset which should not be rejected under the no-change null *H0*. We also split the corrupted data by corruption type:

```
[5]: np.random.seed(0)
n_test = X_test.shape[0]
idx = np.random.choice(n_test, size=n_test // 2, replace=False)
idx_h0 = np.delete(np.arange(n_test), idx, axis=0)
X_ref,y_ref = X_test[idx], y_test[idx]
X_h0, y_h0 = X_test[idx_h0], y_test[idx_h0]
print(X_ref.shape, X_h0.shape)
```

```
(5000, 32, 32, 3) (5000, 32, 32, 3)
```

```
[6]: # check that the classes are more or less balanced
classes, counts_ref = np.unique(y_ref, return_counts=True)
counts_h0 = np.unique(y_h0, return_counts=True)[1]
print('Class Ref H0')
for cl, cref, ch0 in zip(classes, counts_ref, counts_h0):
    assert cref + ch0 == n_test // 10
    print('{}     {} {}'.format(cl, cref, ch0))
```

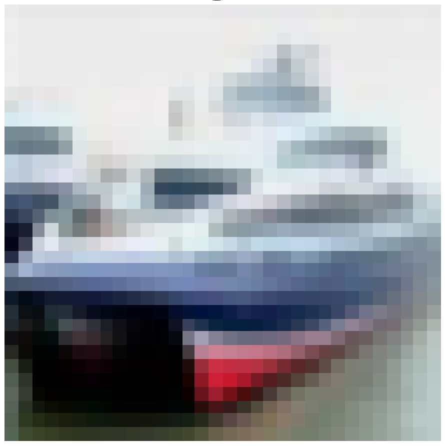
```
Class  Ref  H0
0      472  528
1      510  490
2      498  502
3      492  508
4      501  499
5      495  505
6      493  507
7      501  499
8      516  484
9      522  478
```

```
[7]: n_corr = len(corruption)
X_c = [X_corr[i * n_test:(i + 1) * n_test] for i in range(n_corr)]
```

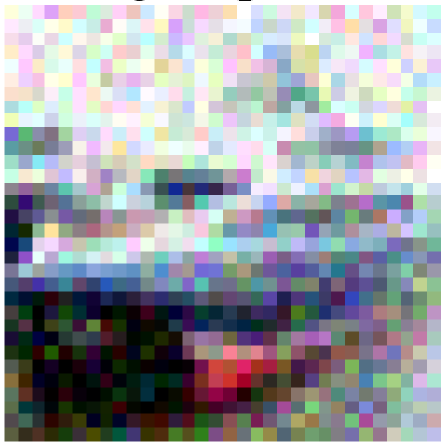
We can visualise the same instance for each corruption type:

```
[8]: i = 1

n_test = X_test.shape[0]
plt.title('Original')
plt.axis('off')
plt.imshow(X_test[i])
plt.show()
for _ in range(len(corruption)):
    plt.title(corruption[_])
    plt.axis('off')
    plt.imshow(X_corr[n_test * _+ i])
    plt.show()
```

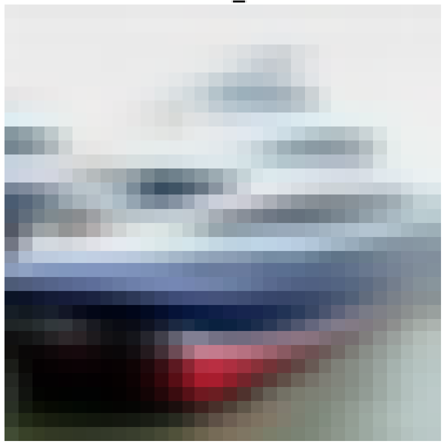
Original



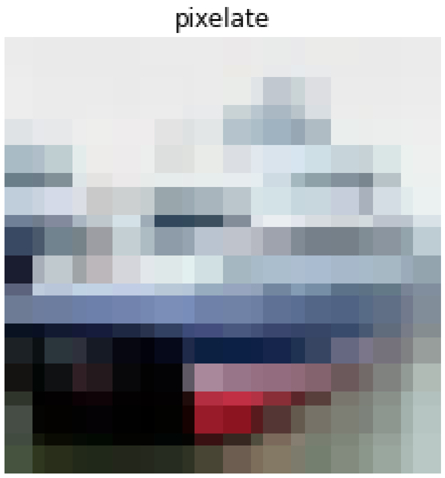
gaussian_noise



motion_blur



brightness



We can also verify that the performance of a classification model on CIFAR-10 drops significantly on this perturbed dataset:

```
[9]: dataset = 'cifar10'
model = 'resnet32'
clf = fetch_tf_model(dataset, model)
acc = clf.evaluate(scale_by_instance(X_test), y_test, batch_size=128, verbose=0)[1]
print('Test set accuracy:')
print('Original {:.4f}'.format(acc))
clf_accuracy = {'original': acc}
for _ in range(len(corruption)):
    acc = clf.evaluate(scale_by_instance(X_c[_]), y_test, batch_size=128,␣
↪verbose=0)[1]
    clf_accuracy[corruption[_]] = acc
    print('{} {:.4f}'.format(corruption[_], acc))
```

```
Test set accuracy:
Original 0.9278
gaussian_noise 0.2208
motion_blur 0.6339
brightness 0.8913
pixelate 0.3666
```

Given the drop in performance, it is important that we detect the harmful data drift!

44.3.2 Detect drift

Unlike many other approaches we needn't specify a dimension-reducing preprocessing step as the detector operates directly on the data as it is input to the model of interest. In fact, the two-stage projection input -> prediction -> uncertainty can be thought of as the projection from the input space onto the real line, ready to perform the test.

We simply pass the model to the detector and inform it that the predictions should be interpreted as 'probs' rather than 'logits' (i.e. a softmax has already been applied). By default `uncertainty_type='entropy'` is used as the notion of uncertainty for classifier predictions, however `uncertainty_type='margin'` can be specified to deem the classifier's prediction uncertain if they fall within a margin (e.g. in [0.45,0.55] for binary classifier probabilities) (similar to Sethi and Kantardzic (2017)).

[10]:
```
cd = ClassifierUncertaintyDrift(
  X_ref, model=clf, backend='tensorflow', p_val=0.05, preds_type='probs'
)
```

Let's check whether the detector thinks drift occurred on the different test sets and time the prediction calls:

[11]:
```
from timeit import default_timer as timer

labels = ['No!', 'Yes!']

def make_predictions(cd, x_h0, x_corr, corruption):
    t = timer()
    preds = cd.predict(x_h0)
    dt = timer() - t
    print('No corruption')
    print('Drift? {}'.format(labels[preds['data']['is_drift']]))
    print('Feature-wise p-values:')
    print(preds['data']['p_val'])
    print(f'Time (s) {dt:.3f}')

    if isinstance(x_corr, list):
        for x, c in zip(x_corr, corruption):
            t = timer()
            preds = cd.predict(x)
            dt = timer() - t
            print('')
            print(f'Corruption type: {c}')
            print('Drift? {}'.format(labels[preds['data']['is_drift']]))
            print('Feature-wise p-values:')
            print(preds['data']['p_val'])
            print(f'Time (s) {dt:.3f}')
```

[12]:
```
make_predictions(cd, X_h0, X_c, corruption)
```

```
No corruption
Drift? No!
Feature-wise p-values:
[0.7868902]
Time (s) 15.574

Corruption type: gaussian_noise
Drift? Yes!
Feature-wise p-values:
[0.]
Time (s) 33.066

Corruption type: motion_blur
Drift? Yes!
Feature-wise p-values:
[0.]
Time (s) 32.637

Corruption type: brightness
Drift? No!
Feature-wise p-values:
[0.1102559]
Time (s) 34.126
```

(continues on next page)

(continued from previous page)

```
Corruption type: pixelate
Drift? Yes!
Feature-wise p-values:
[0.]
Time (s) 34.351
```

Note here how drift is only detected for the corrupted datasets on which the model's performance is significantly degraded. For the 'brightness' corruption, for which the model maintains 89% classification accuracy, the change in model uncertainty is not deemed significant (p-value 0.11, above the 0.05 threshold). For the other corruptions which signficiantly hamper model performance, the malicious drift is detected.

44.4 Regressor uncertainty based drift detection

We now demonstrate how to leverage model uncertainty to detect malicious drift when the model of interest is a regressor. This is a less general approach as regressors often make point-predictions with no associated notion of uncertainty. However, if the model makes its predictions by ensembling the predicitons of sub-models then we can consider the variation in the sub-model predictions as a notion of uncertainty. `RegressorUncertaintyDetector` facilitates models that output a vector of such sub-model predictions (`uncertainty_type='ensemble'`) or deep learning models that include dropout layers and can therefore (as noted by Gal and Ghahramani 2016) be considered as an ensemble (`uncertainty_type='mc_dropout'`, the default option).

44.4.1 Dataset

The Wine Quality Data Set consists of 1599 and 4898 samples of red and white wine respectively. Each sample has an associated quality (as determined by experts) and 11 numeric features indicating its acidity, density, pH etc. We consider the regression problem of tring to predict the quality of red wine sample given these features. We will then consider whether the model remains suitable for predicting the quality of white wine samples or whether the associated change in the underlying distribution should be considered as malicious drift.

First we load in the data.

[13]:
```
red = pd.read_csv(
    "http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/
↪winequality-red.csv", sep=';'
)
white = pd.read_csv(
    "http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/
↪winequality-white.csv", sep=';'
)
red.describe()
```

[13]:
```
       fixed acidity  volatile acidity  citric acid  residual sugar  \
count    1599.000000       1599.000000  1599.000000     1599.000000
mean        8.319637          0.527821     0.270976        2.538806
std         1.741096          0.179060     0.194801        1.409928
min         4.600000          0.120000     0.000000        0.900000
25%         7.100000          0.390000     0.090000        1.900000
50%         7.900000          0.520000     0.260000        2.200000
75%         9.200000          0.640000     0.420000        2.600000
max        15.900000          1.580000     1.000000       15.500000

         chlorides  free sulfur dioxide  total sulfur dioxide      density  \
```

(continues on next page)

(continued from previous page)

```
count  1599.000000          1599.000000           1599.000000  1599.000000
mean      0.087467            15.874922             46.467792     0.996747
std       0.047065            10.460157             32.895324     0.001887
min       0.012000             1.000000              6.000000     0.990070
25%       0.070000             7.000000             22.000000     0.995600
50%       0.079000            14.000000             38.000000     0.996750
75%       0.090000            21.000000             62.000000     0.997835
max       0.611000            72.000000            289.000000     1.003690

                pH    sulphates      alcohol      quality
count  1599.000000  1599.000000  1599.000000  1599.000000
mean      3.311113     0.658149    10.422983     5.636023
std       0.154386     0.169507     1.065668     0.807569
min       2.740000     0.330000     8.400000     3.000000
25%       3.210000     0.550000     9.500000     5.000000
50%       3.310000     0.620000    10.200000     6.000000
75%       3.400000     0.730000    11.100000     6.000000
max       4.010000     2.000000    14.900000     8.000000
```

We can see that the data for both red and white wine samples take the same format.

```
[14]: white.describe()
```

```
[14]:        fixed acidity  volatile acidity  citric acid  residual sugar  \
count    4898.000000       4898.000000  4898.000000     4898.000000
mean        6.854788          0.278241     0.334192        6.391415
std         0.843868          0.100795     0.121020        5.072058
min         3.800000          0.080000     0.000000        0.600000
25%         6.300000          0.210000     0.270000        1.700000
50%         6.800000          0.260000     0.320000        5.200000
75%         7.300000          0.320000     0.390000        9.900000
max        14.200000          1.100000     1.660000       65.800000

         chlorides  free sulfur dioxide  total sulfur dioxide      density  \
count  4898.000000          4898.000000           4898.000000  4898.000000
mean      0.045772            35.308085            138.360657     0.994027
std       0.021848            17.007137             42.498065     0.002991
min       0.009000             2.000000              9.000000     0.987110
25%       0.036000            23.000000            108.000000     0.991723
50%       0.043000            34.000000            134.000000     0.993740
75%       0.050000            46.000000            167.000000     0.996100
max       0.346000           289.000000            440.000000     1.038980

                pH    sulphates      alcohol      quality
count  4898.000000  4898.000000  4898.000000  4898.000000
mean      3.188267     0.489847    10.514267     5.877909
std       0.151001     0.114126     1.230621     0.885639
min       2.720000     0.220000     8.000000     3.000000
25%       3.090000     0.410000     9.500000     5.000000
50%       3.180000     0.470000    10.400000     6.000000
75%       3.280000     0.550000    11.400000     6.000000
max       3.820000     1.080000    14.200000     9.000000
```

We shuffle and normalise the data such that each feature takes a value in [0,1], as does the quality we seek to predict.

```
[15]: red, white = np.asarray(red, np.float32), np.asarray(white, np.float32)
n_red, n_white = red.shape[0], white.shape[0]
```

(continues on next page)

(continued from previous page)

```
col_maxes = red.max(axis=0)
red, white = red / col_maxes, white / col_maxes
red, white = red[np.random.permutation(n_red)], white[np.random.permutation(n_white)]
X, y = red[:, :-1], red[:, -1:]
X_corr, y_corr = white[:, :-1], white[:, -1:]
```

We split the red wine data into a set on which to train the model, a reference set with which to instantiate the detector and a set which the detector should not flag drift. We then instantiate a DataLoader to pass the training data to a PyTorch model in batches.

```
[16]: X_train, y_train = X[:(n_red//2)], y[:(n_red//2)]
X_ref, y_ref = X[(n_red//2):(3*n_red//4)], y[(n_red//2):(3*n_red//4)]
X_h0, y_h0 = X[(3*n_red//4):], y[(3*n_red//4):]

X_train_ds = torch.utils.data.TensorDataset(torch.tensor(X_train), torch.tensor(y_
↪train))
X_train_dl = torch.utils.data.DataLoader(X_train_ds, batch_size=32, shuffle=True,␣
↪drop_last=True)
```

44.4.2 Regression model

We now define the regression model that we'll train to predict the quality from the features. The exact details aren't important other than the presence of at least one dropout layer. We then train the model for 20 epochs to optimise the mean square error on the training data.

```
[17]: reg = nn.Sequential(
    nn.Linear(11, 16),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(16, 32),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(32, 1)
)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
trainer(reg, nn.MSELoss(), X_train_dl, device, torch.optim.Adam, learning_rate=0.001,␣
↪epochs=30)
```

```
/home/oliver/Projects/alibi-detect/.venv/lib/python3.8/site-packages/torch/cuda/__
↪init__.py:52: UserWarning: CUDA initialization: Found no NVIDIA driver on your␣
↪system. Please check that you have an NVIDIA GPU and installed a driver from http://
↪www.nvidia.com/Download/index.aspx (Triggered internally at  /pytorch/c10/cuda/
↪CUDAFunctions.cpp:100.)
  return torch._C._cuda_getDeviceCount() > 0
Epoch 1/30: 100%|| 24/24 [00:00<00:00, 267.60it/s, loss=0.119]
Epoch 2/30: 100%|| 24/24 [00:00<00:00, 267.14it/s, loss=0.0857]
Epoch 3/30: 100%|| 24/24 [00:00<00:00, 266.90it/s, loss=0.043]
Epoch 4/30: 100%|| 24/24 [00:00<00:00, 250.43it/s, loss=0.0553]
Epoch 5/30: 100%|| 24/24 [00:00<00:00, 187.70it/s, loss=0.0365]
Epoch 6/30: 100%|| 24/24 [00:00<00:00, 260.13it/s, loss=0.03]
Epoch 7/30: 100%|| 24/24 [00:00<00:00, 245.26it/s, loss=0.0552]
Epoch 8/30: 100%|| 24/24 [00:00<00:00, 241.64it/s, loss=0.0335]
Epoch 9/30: 100%|| 24/24 [00:00<00:00, 229.60it/s, loss=0.0254]
Epoch 10/30: 100%|| 24/24 [00:00<00:00, 244.06it/s, loss=0.0223]
```

(continues on next page)

(continued from previous page)

```
Epoch 11/30: 100%|| 24/24 [00:00<00:00, 225.92it/s, loss=0.0224]
Epoch 12/30: 100%|| 24/24 [00:00<00:00, 204.65it/s, loss=0.0254]
Epoch 13/30: 100%|| 24/24 [00:00<00:00, 226.55it/s, loss=0.0236]
Epoch 14/30: 100%|| 24/24 [00:00<00:00, 226.15it/s, loss=0.0247]
Epoch 15/30: 100%|| 24/24 [00:00<00:00, 250.90it/s, loss=0.0292]
Epoch 16/30: 100%|| 24/24 [00:00<00:00, 208.73it/s, loss=0.0263]
Epoch 17/30: 100%|| 24/24 [00:00<00:00, 294.98it/s, loss=0.0163]
Epoch 18/30: 100%|| 24/24 [00:00<00:00, 173.03it/s, loss=0.0223]
Epoch 19/30: 100%|| 24/24 [00:00<00:00, 186.12it/s, loss=0.0244]
Epoch 20/30: 100%|| 24/24 [00:00<00:00, 228.75it/s, loss=0.0295]
Epoch 21/30: 100%|| 24/24 [00:00<00:00, 240.64it/s, loss=0.0218]
Epoch 22/30: 100%|| 24/24 [00:00<00:00, 242.33it/s, loss=0.019]
Epoch 23/30: 100%|| 24/24 [00:00<00:00, 275.11it/s, loss=0.0257]
Epoch 24/30: 100%|| 24/24 [00:00<00:00, 267.61it/s, loss=0.0165]
Epoch 25/30: 100%|| 24/24 [00:00<00:00, 290.65it/s, loss=0.0192]
Epoch 26/30: 100%|| 24/24 [00:00<00:00, 259.52it/s, loss=0.0224]
Epoch 27/30: 100%|| 24/24 [00:00<00:00, 244.17it/s, loss=0.0173]
Epoch 28/30: 100%|| 24/24 [00:00<00:00, 261.72it/s, loss=0.0159]
Epoch 29/30: 100%|| 24/24 [00:00<00:00, 293.25it/s, loss=0.012]
Epoch 30/30: 100%|| 24/24 [00:00<00:00, 298.45it/s, loss=0.022]
```

We now evaluate the trained model on both unseen samples of red wine and white wine. We see that, unsurprisingly, the model is better able to predict the quality of unseen red wine samples.

[18]:
```
reg = reg.eval()
reg_fn = encompass_batching(reg, backend='pytorch', batch_size=32)
preds_ref = reg_fn(X_ref)
preds_corr = reg_fn(X_corr)

ref_mse = np.square(preds_ref - y_ref).mean()
corr_mse = np.square(preds_corr - y_corr).mean()

print(f'MSE when predicting the quality of unseen red wine samples: {ref_mse}')
print(f'MSE when predicting the quality of unseen white wine samples: {corr_mse}')
```

```
MSE when predicting the quality of unseen red wine samples: 0.008570569567382336
MSE when predicting the quality of unseen white wine samples: 0.014613097533583641
```

44.4.3 Detect drift

We now look at whether a regressor-uncertainty detector would have picked up on this malicious drift. We instantiate the detector and obtain drift predictions on both the held-out red-wine samples and the white-wine samples. We specify `uncertainty_type='mc_dropout'` in this case, but alternatively we could have trained an ensemble model that for each instance outputs a vector of multiple independent predictions and specified `uncertainty_type='ensemble'`.

[19]:
```
cd = RegressorUncertaintyDrift(
    X_ref, model=reg, backend='pytorch', p_val=0.05, uncertainty_type='mc_dropout', n_
↪evals=100
)
preds_h0 = cd.predict(X_h0)
preds_h1 = cd.predict(X_corr)

print(f"Drift detected on unseen red wine samples? {'yes' if preds_h0['data']['is_
↪drift']==1 else 'no'}")
```

(continues on next page)

(continued from previous page)

```
print(f"Drift detected on white wine samples? {'yes' if preds_h1['data']['is_drift
↪']==1 else 'no'}")

print(f"p-value on unseen red wine samples? {preds_h0['data']['p_val']}")
print(f"p-value on white wine samples? {preds_h1['data']['p_val']}")
```

```
Drift detected on unseen red wine samples? no
Drift detected on white wine samples? yes
p-value on unseen red wine samples? [0.23237702]
p-value on white wine samples? [1.7934791e-10]
```

source

CHAPTER

FORTYFIVE

OVERVIEW

The package also contains functionality in `alibi_detect.datasets` to easily fetch a number of datasets for different modalities. For each dataset either the data and labels or a *Bunch* object with the data, labels and optional metadata are returned. Example:

```
from alibi_detect.datasets import fetch_ecg

(X_train, y_train), (X_test, y_test) = fetch_ecg(return_X_y=True)
```

45.1 Sequential Data and Time Series

Genome Dataset: `fetch_genome`

- Bacteria genomics dataset for out-of-distribution detection, released as part of Likelihood Ratios for Out-of-Distribution Detection. From the original *TL;DR*: *The dataset contains genomic sequences of 250 base pairs from 10 in-distribution bacteria classes for training, 60 OOD bacteria classes for validation, and another 60 different OOD bacteria classes for test*. There are respectively 1, 7 and again 7 million sequences in the training, validation and test sets. For detailed info on the dataset check the README.

```
from alibi_detect.datasets import fetch_genome

(X_train, y_train), (X_val, y_val), (X_test, y_test) = fetch_genome(return_X_y=True)
```

ECG 5000: `fetch_ecg`

- 5000 ECG's, originally obtained from Physionet.

NAB: `fetch_nab`

- Any univariate time series in a DataFrame from the Numenta Anomaly Benchmark. A list with the available time series can be retrieved using `alibi_detect.datasets.get_list_nab()`.

45.2 Images

CIFAR-10-C: `fetch_cifar10c`

- CIFAR-10-C (Hendrycks & Dietterich, 2019) contains the test set of CIFAR-10, but corrupted and perturbed by various types of noise, blur, brightness etc. at different levels of severity, leading to a gradual decline in a classification model's performance trained on CIFAR-10. `fetch_cifar10c` allows you to pick any severity level or corruption type. The list with available corruption types can be retrieved with `alibi_detect.datasets.corruption_types_cifar10c()`. The dataset can be used in research on robustness and drift. The original data can be found here. Example:

```
from alibi_detect.datasets import fetch_cifar10c

corruption = ['gaussian_noise', 'motion_blur', 'brightness', 'pixelate']
X, y = fetch_cifar10c(corruption=corruption, severity=5, return_X_y=True)
```

Adversarial CIFAR-10: `fetch_attack`

- Load adversarial instances on a ResNet-56 classifier trained on CIFAR-10. Available attacks: Carlini-Wagner ('cw') and SLIDE ('slide'). Example:

```
from alibi_detect.datasets import fetch_attack

(X_train, y_train), (X_test, y_test) = fetch_attack('cifar10', 'resnet56', 'cw',␣
↪return_X_y=True)
```

45.3 Tabular

KDD Cup '99: `fetch_kdd`

- Dataset with different types of computer network intrusions. `fetch_kdd` allows you to select a subset of network intrusions as targets or pick only specified features. The original data can be found here.

source

CHAPTER

FORTYSIX

OVERVIEW

Models and/or building blocks that can be useful outside of outlier, adversarial or drift detection can be found under `alibi_detect.models`. Main implementations:

- PixelCNN++: `from alibi_detect.models.tensorflow import PixelCNN`
- Variational Autoencoder: `from alibi_detect.models.tensorflow import VAE`
- Sequence-to-sequence model: `from alibi_detect.models.tensorflow import Seq2Seq`
- ResNet: `from alibi_detect.models.tensorflow import resnet`

Pre-trained TensorFlow ResNet-20/32/44 models on CIFAR-10 can be found on our Google Cloud Bucket and can be fetched as follows:

```
from alibi_detect.utils.fetching import fetch_tf_model

model = fetch_tf_model('cifar10', 'resnet32')
```

CHAPTER

FORTYSEVEN

ALIBI_DETECT

47.1 alibi_detect package

47.1.1 Subpackages

alibi_detect.ad package

class alibi_detect.ad.**AdversarialAE**(*threshold=None*, *ae=None*, *model=None*, *encoder_net=None*, *decoder_net=None*, *model_hl=None*, *hidden_layer_kld=None*, *w_model_hl=None*, *temperature=1.0*, *data_type=None*)

Bases: `alibi_detect.base.BaseDetector`, `alibi_detect.base.FitMixin`, `alibi_detect.base.ThresholdMixin`

__init__(*threshold=None*, *ae=None*, *model=None*, *encoder_net=None*, *decoder_net=None*, *model_hl=None*, *hidden_layer_kld=None*, *w_model_hl=None*, *temperature=1.0*, *data_type=None*)

Autoencoder (AE) based adversarial detector.

Parameters

- **threshold** (Optional[float]) – Threshold used for adversarial score to determine adversarial instances.
- **ae** (Optional[Model]) – A trained tf.keras autoencoder model if available.
- **model** (Optional[Model]) – A trained tf.keras classification model.
- **encoder_net** (Optional[Sequential]) – Layers for the encoder wrapped in a tf.keras.Sequential class if no 'ae' is specified.
- **decoder_net** (Optional[Sequential]) – Layers for the decoder wrapped in a tf.keras.Sequential class if no 'ae' is specified.
- **model_hl** (Optional[List[Model]]) – List with tf.keras models for the hidden layer K-L divergence computation.
- **hidden_layer_kld** (Optional[dict]) – Dictionary with as keys the hidden layer(s) of the model which are extracted and used during training of the AE, and as values the output dimension for the hidden layer.
- **w_model_hl** (Optional[list]) – Weights assigned to the loss of each model in model_hl.
- **temperature** (float) – Temperature used for model prediction scaling. Temperature <1 sharpens the prediction probability distribution.

- **data_type** (Optional[str]) – Optionally specifiy the data type (tabular, image or time-series). Added to metadata.

Return type None

correct (*X*, *batch_size=10000000000*, *return_instance_score=True*, *return_all_predictions=True*)
Correct adversarial instances if the adversarial score is above the threshold.

Parameters

- **X** (ndarray) – Batch of instances.
- **batch_size** (int) – Batch size used when computing scores.
- **return_instance_score** (bool) – Whether to return instance level adversarial scores.
- **return_all_predictions** (bool) – Whether to return the predictions on the original and the reconstructed data.

Return type Dict[Dict[str, str], Dict[str, ndarray]]

Returns *Dict with corrected predictions and information whether an instance is adversarial or not.*

fit (*X*, *loss_fn=<function loss_adv_ae>*, *w_model=1.0*, *w_recon=0.0*, *optimizer=tensorflow.keras.optimizers.Adam*, *epochs=20*, *batch_size=128*, *verbose=True*, *log_metric=None*, *callbacks=None*, *preprocess_fn=None*)
Train Adversarial AE model.

Parameters

- **X** – Training batch.
- **loss_fn** – Loss function used for training.
- **w_model** – Weight on model prediction loss term.
- **w_recon** – Weight on MSE reconstruction error loss term.
- **optimizer** – Optimizer used for training.
- **epochs** – Number of training epochs.
- **batch_size** – Batch size used for training.
- **verbose** – Whether to print training progress.
- **log_metric** – Additional metrics whose progress will be displayed if verbose equals True.
- **callbacks** – Callbacks used during training.
- **preprocess_fn** – Preprocessing function applied to each training batch.

infer_threshold (*X*, *threshold_perc=99.0*, *margin=0.0*, *batch_size=10000000000*)
Update threshold by a value inferred from the percentage of instances considered to be adversarial in a sample of the dataset.

Parameters

- **X** (ndarray) – Batch of instances.
- **threshold_perc** (float) – Percentage of X considered to be normal based on the adversarial score.
- **margin** (float) – Add margin to threshold. Useful if adversarial instances have significantly higher scores and there is no adversarial instance in X.

- **batch_size** (int) – Batch size used when computing scores.

Return type None

predict (*X, batch_size=10000000000, return_instance_score=True*)
Predict whether instances are adversarial instances or not.

Parameters

- **X** (ndarray) – Batch of instances.
- **batch_size** (int) – Batch size used when computing scores.
- **return_instance_score** (bool) – Whether to return instance level adversarial scores.

Return type Dict[Dict[str, str], Dict[str, ndarray]]

Returns

- *Dictionary containing 'meta' and 'data' dictionaries.*
- *'meta' has the model's metadata.*
- *'data' contains the adversarial predictions and instance level adversarial scores.*

score (*X, batch_size=10000000000, return_predictions=False*)
Compute adversarial scores.

Parameters

- **X** (ndarray) – Batch of instances to analyze.
- **batch_size** (int) – Batch size used when computing scores.
- **return_predictions** (bool) – Whether to return the predictions of the classifier on the original and reconstructed instances.

Return type Union[ndarray, Tuple[ndarray, ndarray, ndarray]]

Returns *Array with adversarial scores for each instance in the batch.*

class alibi_detect.ad.**ModelDistillation** (*threshold=None, distilled_model=None, model=None, loss_type='kld', temperature=1.0, data_type=None*)
Bases: *alibi_detect.base.BaseDetector*, *alibi_detect.base.FitMixin*, *alibi_detect.base.ThresholdMixin*

__init__ (*threshold=None, distilled_model=None, model=None, loss_type='kld', temperature=1.0, data_type=None*)
Model distillation concept drift and adversarial detector.

Parameters

- **threshold** (Optional[float]) – Threshold used for score to determine adversarial instances.
- **distilled_model** (Optional[Model]) – A tf.keras model to distill.
- **model** (Optional[Model]) – A trained tf.keras classification model.
- **loss_type** (str) – Loss for distillation. Supported: 'kld', 'xent'
- **temperature** (float) – Temperature used for model prediction scaling. Temperature <1 sharpens the prediction probability distribution.
- **data_type** (Optional[str]) – Optionally specifiy the data type (tabular, image or time-series). Added to metadata.

Return type `None`

`fit` (*X*, *loss_fn=<function loss_distillation>*, *optimizer=tensorflow.keras.optimizers.Adam*, *epochs=20*, *batch_size=128*, *verbose=True*, *log_metric=None*, *callbacks=None*, *preprocess_fn=None*)
Train ModelDistillation detector.

Parameters

- **`X`** – Training batch.
- **`loss_fn`** – Loss function used for training.
- **`optimizer`** – Optimizer used for training.
- **`epochs`** – Number of training epochs.
- **`batch_size`** – Batch size used for training.
- **`verbose`** – Whether to print training progress.
- **`log_metric`** – Additional metrics whose progress will be displayed if verbose equals True.
- **`callbacks`** – Callbacks used during training.
- **`preprocess_fn`** – Preprocessing function applied to each training batch.

`infer_threshold` (*X*, *threshold_perc=99.0*, *margin=0.0*, *batch_size=10000000000*)
Update threshold by a value inferred from the percentage of instances considered to be adversarial in a sample of the dataset.

Parameters

- **`X`** (`ndarray`) – Batch of instances.
- **`threshold_perc`** (`float`) – Percentage of X considered to be normal based on the adversarial score.
- **`margin`** (`float`) – Add margin to threshold. Useful if adversarial instances have significantly higher scores and there is no adversarial instance in X.
- **`batch_size`** (`int`) – Batch size used when computing scores.

Return type `None`

`predict` (*X*, *batch_size=10000000000*, *return_instance_score=True*)
Predict whether instances are adversarial instances or not.

Parameters

- **`X`** (`ndarray`) – Batch of instances.
- **`batch_size`** (`int`) – Batch size used when computing scores.
- **`return_instance_score`** (`bool`) – Whether to return instance level adversarial scores.

Return type `Dict`[`Dict`[`str`, `str`], `Dict`[`str`, `ndarray`]]

Returns

- *Dictionary containing 'meta' and 'data' dictionaries.*
- *'meta' has the model's metadata.*
- *'data' contains the adversarial predictions and instance level adversarial scores.*

`score` (*X*, *batch_size=10000000000*, *return_predictions=False*)
Compute adversarial scores.

Parameters

- **X** (ndarray) – Batch of instances to analyze.
- **batch_size** (int) – Batch size used when computing scores.
- **return_predictions** (bool) – Whether to return the predictions of the classifier on the original and reconstructed instances.

Return type Union[ndarray, Tuple[ndarray, ndarray, ndarray]]

Returns *Array with adversarial scores for each instance in the batch.*

Submodules

alibi_detect.ad.adversarialae module

class alibi_detect.ad.adversarialae.**AdversarialAE**(*threshold=None*, *ae=None*, *model=None*, *encoder_net=None*, *decoder_net=None*, *model_hl=None*, *hidden_layer_kld=None*, *w_model_hl=None*, *temperature=1.0*, *data_type=None*)

Bases: *alibi_detect.base.BaseDetector*, *alibi_detect.base.FitMixin*, *alibi_detect.base.ThresholdMixin*

__init__(*threshold=None*, *ae=None*, *model=None*, *encoder_net=None*, *decoder_net=None*, *model_hl=None*, *hidden_layer_kld=None*, *w_model_hl=None*, *temperature=1.0*, *data_type=None*)

Autoencoder (AE) based adversarial detector.

Parameters

- **threshold** (Optional[float]) – Threshold used for adversarial score to determine adversarial instances.
- **ae** (Optional[Model]) – A trained tf.keras autoencoder model if available.
- **model** (Optional[Model]) – A trained tf.keras classification model.
- **encoder_net** (Optional[Sequential]) – Layers for the encoder wrapped in a tf.keras.Sequential class if no 'ae' is specified.
- **decoder_net** (Optional[Sequential]) – Layers for the decoder wrapped in a tf.keras.Sequential class if no 'ae' is specified.
- **model_hl** (Optional[List[Model]]) – List with tf.keras models for the hidden layer K-L divergence computation.
- **hidden_layer_kld** (Optional[dict]) – Dictionary with as keys the hidden layer(s) of the model which are extracted and used during training of the AE, and as values the output dimension for the hidden layer.
- **w_model_hl** (Optional[list]) – Weights assigned to the loss of each model in model_hl.
- **temperature** (float) – Temperature used for model prediction scaling. Temperature <1 sharpens the prediction probability distribution.

- **data_type** (Optional[str]) – Optionally specifiy the data type (tabular, image or time-series). Added to metadata.

Return type None

correct (*X*, *batch_size=10000000000*, *return_instance_score=True*, *return_all_predictions=True*)
Correct adversarial instances if the adversarial score is above the threshold.

Parameters

- **X** (ndarray) – Batch of instances.
- **batch_size** (int) – Batch size used when computing scores.
- **return_instance_score** (bool) – Whether to return instance level adversarial scores.
- **return_all_predictions** (bool) – Whether to return the predictions on the original and the reconstructed data.

Return type Dict[Dict[str, str], Dict[str, ndarray]]

Returns *Dict with corrected predictions and information whether an instance is adversarial or not.*

fit (*X*, *loss_fn=<function loss_adv_ae>*, *w_model=1.0*, *w_recon=0.0*, *optimizer=tensorflow.keras.optimizers.Adam*, *epochs=20*, *batch_size=128*, *verbose=True*, *log_metric=None*, *callbacks=None*, *preprocess_fn=None*)
Train Adversarial AE model.

Parameters

- **X** – Training batch.
- **loss_fn** – Loss function used for training.
- **w_model** – Weight on model prediction loss term.
- **w_recon** – Weight on MSE reconstruction error loss term.
- **optimizer** – Optimizer used for training.
- **epochs** – Number of training epochs.
- **batch_size** – Batch size used for training.
- **verbose** – Whether to print training progress.
- **log_metric** – Additional metrics whose progress will be displayed if verbose equals True.
- **callbacks** – Callbacks used during training.
- **preprocess_fn** – Preprocessing function applied to each training batch.

infer_threshold (*X*, *threshold_perc=99.0*, *margin=0.0*, *batch_size=10000000000*)
Update threshold by a value inferred from the percentage of instances considered to be adversarial in a sample of the dataset.

Parameters

- **X** (ndarray) – Batch of instances.
- **threshold_perc** (float) – Percentage of X considered to be normal based on the adversarial score.
- **margin** (float) – Add margin to threshold. Useful if adversarial instances have significantly higher scores and there is no adversarial instance in X.

- **batch_size** (int) – Batch size used when computing scores.

Return type None

predict (*X*, *batch_size=10000000000*, *return_instance_score=True*)
Predict whether instances are adversarial instances or not.

Parameters

- **X** (ndarray) – Batch of instances.
- **batch_size** (int) – Batch size used when computing scores.
- **return_instance_score** (bool) – Whether to return instance level adversarial scores.

Return type Dict[Dict[str, str], Dict[str, ndarray]]

Returns

- *Dictionary containing 'meta' and 'data' dictionaries.*
- *'meta' has the model's metadata.*
- *'data' contains the adversarial predictions and instance level adversarial scores.*

score (*X*, *batch_size=10000000000*, *return_predictions=False*)
Compute adversarial scores.

Parameters

- **X** (ndarray) – Batch of instances to analyze.
- **batch_size** (int) – Batch size used when computing scores.
- **return_predictions** (bool) – Whether to return the predictions of the classifier on the original and reconstructed instances.

Return type Union[ndarray, Tuple[ndarray, ndarray, ndarray]]

Returns *Array with adversarial scores for each instance in the batch.*

class alibi_detect.ad.adversarialae.**DenseHidden** (*model*, *hidden_layer*, *output_dim*, *hidden_dim=None*)
Bases: tensorflow.keras.Model

__init__ (*model*, *hidden_layer*, *output_dim*, *hidden_dim=None*)
Dense layer that extracts the feature map of a hidden layer in a model and computes output probabilities over that layer.

Parameters

- **model** (Model) – tf.keras classification model.
- **hidden_layer** (int) – Hidden layer from model where feature map is extracted from.
- **output_dim** (int) – Output dimension for softmax layer.
- **hidden_dim** (Optional[int]) – Dimension of optional additional dense layer.

Return type None

call (*x*)

Return type Tensor

alibi_detect.ad.model_distillation module

class alibi_detect.ad.model_distillation.**ModelDistillation**(*threshold=None*, *distilled_model=None*, *model=None*, *loss_type='kld'*, *temperature=1.0*, *data_type=None*)

Bases: *alibi_detect.base.BaseDetector*, *alibi_detect.base.FitMixin*, *alibi_detect.base.ThresholdMixin*

__init__(*threshold=None*, *distilled_model=None*, *model=None*, *loss_type='kld'*, *temperature=1.0*, *data_type=None*)

Model distillation concept drift and adversarial detector.

Parameters

- **threshold** (Optional[float]) – Threshold used for score to determine adversarial instances.
- **distilled_model** (Optional[Model]) – A tf.keras model to distill.
- **model** (Optional[Model]) – A trained tf.keras classification model.
- **loss_type** (str) – Loss for distillation. Supported: 'kld', 'xent'
- **temperature** (float) – Temperature used for model prediction scaling. Temperature <1 sharpens the prediction probability distribution.
- **data_type** (Optional[str]) – Optionally specifiy the data type (tabular, image or time-series). Added to metadata.

Return type None

fit(*X*, *loss_fn=<function loss_distillation>*, *optimizer=tensorflow.keras.optimizers.Adam*, *epochs=20*, *batch_size=128*, *verbose=True*, *log_metric=None*, *callbacks=None*, *preprocess_fn=None*)

Train ModelDistillation detector.

Parameters

- **X** – Training batch.
- **loss_fn** – Loss function used for training.
- **optimizer** – Optimizer used for training.
- **epochs** – Number of training epochs.
- **batch_size** – Batch size used for training.
- **verbose** – Whether to print training progress.
- **log_metric** – Additional metrics whose progress will be displayed if verbose equals True.
- **callbacks** – Callbacks used during training.
- **preprocess_fn** – Preprocessing function applied to each training batch.

infer_threshold(*X*, *threshold_perc=99.0*, *margin=0.0*, *batch_size=10000000000*)

Update threshold by a value inferred from the percentage of instances considered to be adversarial in a sample of the dataset.

Parameters

- **X** (ndarray) – Batch of instances.

- **threshold_perc** (float) – Percentage of X considered to be normal based on the adversarial score.
- **margin** (float) – Add margin to threshold. Useful if adversarial instances have significantly higher scores and there is no adversarial instance in X.
- **batch_size** (int) – Batch size used when computing scores.

Return type None

predict (*X*, *batch_size=10000000000*, *return_instance_score=True*)
Predict whether instances are adversarial instances or not.

Parameters

- **X** (ndarray) – Batch of instances.
- **batch_size** (int) – Batch size used when computing scores.
- **return_instance_score** (bool) – Whether to return instance level adversarial scores.

Return type Dict[Dict[str, str], Dict[str, ndarray]]

Returns

- *Dictionary containing 'meta' and 'data' dictionaries.*
- *'meta' has the model's metadata.*
- *'data' contains the adversarial predictions and instance level adversarial scores.*

score (*X*, *batch_size=10000000000*, *return_predictions=False*)
Compute adversarial scores.

Parameters

- **X** (ndarray) – Batch of instances to analyze.
- **batch_size** (int) – Batch size used when computing scores.
- **return_predictions** (bool) – Whether to return the predictions of the classifier on the original and reconstructed instances.

Return type Union[ndarray, Tuple[ndarray, ndarray, ndarray]]

Returns *Array with adversarial scores for each instance in the batch.*

alibi_detect.cd package

Subpackages

alibi_detect.cd.pytorch package

class alibi_detect.cd.pytorch.**HiddenOutput** (*model*, *layer=-1*)
Bases: torch.nn.Module

forward (*x*)

Return type Tensor

alibi_detect.cd.pytorch.**preprocess_drift**(*x, model, device=None, tokenizer=None, max_len=None, batch_size=10000000000, dtype=numpy.float32*)

Prediction function used for preprocessing step of drift detector.

Parameters

- **x** (ndarray) – Batch of instances.
- **model** (Union[Module, Sequential]) – Model used for preprocessing.
- **device** (Optional[device]) – Device type used. The default None tries to use the GPU and falls back on CPU if needed. Can be specified by passing either torch.device('cuda') or torch.device('cpu').
- **tokenizer** (Optional[Callable]) – Optional tokenizer for text drift.
- **max_len** (Optional[int]) – Optional max token length for text drift.
- **batch_size** (int) – Batch size used during prediction.
- **dtype** (type) – Model output type, e.g. np.float32 or torch.float32.

Return type Union[ndarray, Tensor]

Returns *Numpy array or torch tensor with predictions.*

Submodules

alibi_detect.cd.pytorch.classifier module

alibi_detect.cd.pytorch.mmd module

class alibi_detect.cd.pytorch.mmd.**MMDDriftTorch**(*x_ref, p_val=0.05, preprocess_x_ref=True, update_x_ref=None, preprocess_fn=None, kernel=<class 'alibi_detect.utils.pytorch.kernels.GaussianRBF'>, sigma=None, configure_kernel_from_x_ref=True, n_permutations=100, device=None, input_shape=None, data_type=None*)

Bases: *alibi_detect.cd.base.BaseMMDDrift*

__init__(*x_ref, p_val=0.05, preprocess_x_ref=True, update_x_ref=None, preprocess_fn=None, kernel=<class 'alibi_detect.utils.pytorch.kernels.GaussianRBF'>, sigma=None, configure_kernel_from_x_ref=True, n_permutations=100, device=None, input_shape=None, data_type=None*)

Maximum Mean Discrepancy (MMD) data drift detector using a permutation test.

Parameters

- **x_ref** (ndarray) – Data used as reference distribution.
- **p_val** (float) – p-value used for the significance of the permutation test.
- **preprocess_x_ref** (bool) – Whether to already preprocess and store the reference data.

- **update_x_ref** (Optional[Dict[str, int]]) – Reference data can optionally be updated to the last n instances seen by the detector or via reservoir sampling with size n. For the former, the parameter equals {'last': n} while for reservoir sampling {'reservoir_sampling': n} is passed.
- **preprocess_fn** (Optional[Callable]) – Function to preprocess the data before computing the data drift metrics.
- **kernel** (Callable) – Kernel used for the MMD computation, defaults to Gaussian RBF kernel.
- **sigma** (Optional[ndarray]) – Optionally set the GaussianRBF kernel bandwidth. Can also pass multiple bandwidth values as an array. The kernel evaluation is then averaged over those bandwidths.
- **configure_kernel_from_x_ref** (bool) – Whether to already configure the kernel bandwidth from the reference data.
- **n_permutations** (int) – Number of permutations used in the permutation test.
- **device** (Optional[str]) – Device type used. The default None tries to use the GPU and falls back on CPU if needed. Can be specified by passing either 'cuda', 'gpu' or 'cpu'.
- **input_shape** (Optional[tuple]) – Shape of input data.
- **data_type** (Optional[str]) – Optionally specify the data type (tabular, image or time-series). Added to metadata.

Return type `None`

kernel_matrix (*x*, *y*)
Compute and return full kernel matrix between arrays x and y.

Return type `Tensor`

score (*x*)
Compute the p-value resulting from a permutation test using the maximum mean discrepancy as a distance measure between the reference data and the data to be tested.

Parameters **x** (ndarray) – Batch of instances.

Return type Tuple[float, float, ndarray]

Returns

- *p-value obtained from the permutation test, the MMD^2 between the reference and test set*
- *and the MMD^2 values from the permutation test.*

alibi_detect.cd.pytorch.preprocess module

class alibi_detect.cd.pytorch.preprocess.**HiddenOutput** (*model*, *layer=-1*)
Bases: `torch.nn.Module`

forward (*x*)

Return type `Tensor`

alibi_detect.cd.pytorch.preprocess.**preprocess_drift** (*x*, *model*, *device=None*, *tokenizer=None*, *max_len=None*, *batch_size=10000000000*, *dtype=numpy.float32*)
Prediction function used for preprocessing step of drift detector.

Parameters

- **x** (ndarray) – Batch of instances.
- **model** (Union[Module, Sequential]) – Model used for preprocessing.
- **device** (Optional[device]) – Device type used. The default None tries to use the GPU and falls back on CPU if needed. Can be specified by passing either torch.device('cuda') or torch.device('cpu').
- **tokenizer** (Optional[Callable]) – Optional tokenizer for text drift.
- **max_len** (Optional[int]) – Optional max token length for text drift.
- **batch_size** (int) – Batch size used during prediction.
- **dtype** (type) – Model output type, e.g. np.float32 or torch.float32.

Return type Union[ndarray, Tensor]

Returns *Numpy array or torch tensor with predictions.*

alibi_detect.cd.pytorch.utils module

alibi_detect.cd.pytorch.utils.**activate_train_mode_for_dropout_layers**(*model*)

Return type Callable

alibi_detect.cd.tensorflow package

Submodules

alibi_detect.cd.tensorflow.classifier module

alibi_detect.cd.tensorflow.mmd module

alibi_detect.cd.tensorflow.preprocess module

alibi_detect.cd.tensorflow.utils module

Submodules

alibi_detect.cd.base module

class alibi_detect.cd.base.**BaseClassifierDrift**(*x_ref*, *p_val=0.05*, *preprocess_x_ref=True*, *update_x_ref=None*, *preprocess_fn=None*, *preds_type='probs'*, *binarize_preds=False*, *train_size=0.75*, *n_folds=None*, *seed=0*, *data_type=None*)

Bases: *alibi_detect.base.BaseDetector*

`__init__`(*x_ref*, *p_val=0.05*, *preprocess_x_ref=True*, *update_x_ref=None*, *preprocess_fn=None*, *preds_type='probs'*, *binarize_preds=False*, *train_size=0.75*, *n_folds=None*, *seed=0*, *data_type=None*)

Base class for the classifier-based drift detector.

Parameters

- **`x_ref`** (`ndarray`) – Data used as reference distribution.
- **`p_val`** (`float`) – p-value used for the significance of the test.
- **`preprocess_x_ref`** (`bool`) – Whether to already preprocess and store the reference data.
- **`update_x_ref`** (`Optional[Dict[str, int]]`) – Reference data can optionally be updated to the last n instances seen by the detector or via reservoir sampling with size n. For the former, the parameter equals {'last': n} while for reservoir sampling {'reservoir_sampling': n} is passed.
- **`preprocess_fn`** (`Optional[Callable]`) – Function to preprocess the data before computing the data drift metrics.
- **`preds_type`** (`str`) – Whether the model outputs probabilities or logits
- **`binarize_preds`** (`bool`) – Whether to test for discrepency on soft (e.g. probs/logits) model predictions directly with a K-S test or binarise to 0-1 prediction errors and apply a binomial test.
- **`train_size`** (`Optional[float]`) – Optional fraction (float between 0 and 1) of the dataset used to train the classifier. The drift is detected on *1 - train_size*. Cannot be used in combination with *n_folds*.
- **`n_folds`** (`Optional[int]`) – Optional number of stratified folds used for training. The model preds are then calculated on all the out-of-fold predictions. This allows to leverage all the reference and test data for drift detection at the expense of longer computation. If both *train_size* and *n_folds* are specified, *n_folds* is prioritized.
- **`seed`** (`int`) – Optional random seed for fold selection.
- **`data_type`** (`Optional[str]`) – Optionally specify the data type (tabular, image or time-series). Added to metadata.

Return type `None`

`get_splits`(*x_ref*, *x*)

Split reference and test data in train and test folds used by the classifier.

Parameters

- **`x_ref`** (`ndarray`) – Data used as reference distribution.
- **`x`** (`ndarray`) – Batch of instances.

Return type `Tuple[ndarray, ndarray, List[Tuple[ndarray, ndarray]]]`

Returns *List with tuples of train and test indices for optionally different folds.*

`predict`(*x*, *return_p_val=True*, *return_distance=True*)

Predict whether a batch of data has drifted from the reference data.

Parameters

- **`x`** (`ndarray`) – Batch of instances.
- **`return_p_val`** (`bool`) – Whether to return the p-value of the test.

- **return_distance** (bool) – Whether to return a notion of strength of the drift. K-S test stat if binarize_preds=False, otherwise relative error reduction.

Return type Dict[Dict[str, str], Dict[str, Union[int, float]]]

Returns

- *Dictionary containing 'meta' and 'data' dictionaries.*
- *'meta' has the model's metadata.*
- *'data' contains the drift prediction and optionally the performance of the classifier* – relative to its expectation under the no-change null.

preprocess(*x*)

Data preprocessing before computing the drift scores. :type x: `ndarray` :param x: Batch of instances.

Return type Tuple[ndarray, ndarray]

Returns *Preprocessed reference data and new instances.*

abstract score(*x*)

Return type Tuple[float, float]

test_probs(*y_oof*, *probs_oof*, *n_ref*, *n_cur*)

Perform a statistical test of the probabilities predicted by the model against what we'd expect under the no-change null.

Parameters

- **y_oof** (ndarray) – Out of fold targets (0 ref, 1 cur)
- **probs_oof** (ndarray) – Probabilities predicted by the model
- **n_ref** (int) – Size of reference window used in training model
- **n_cur** (int) – Size of current window used in trianing model

Return type Tuple[float, float]

Returns *p-value and notion of performance of classifier relative to expectation under null*

class alibi_detect.cd.base.**BaseMMDDrift**(*x_ref*, *p_val=0.05*, *preprocess_x_ref=True*, *update_x_ref=None*, *preprocess_fn=None*, *sigma=None*, *configure_kernel_from_x_ref=True*, *n_permutations=100*, *input_shape=None*, *data_type=None*)

Bases: *alibi_detect.base.BaseDetector*

__init__(*x_ref*, *p_val=0.05*, *preprocess_x_ref=True*, *update_x_ref=None*, *preprocess_fn=None*, *sigma=None*, *configure_kernel_from_x_ref=True*, *n_permutations=100*, *input_shape=None*, *data_type=None*)

Maximum Mean Discrepancy (MMD) base data drift detector using a permutation test.

Parameters

- **x_ref** (ndarray) – Data used as reference distribution.
- **p_val** (float) – p-value used for the significance of the permutation test.
- **preprocess_x_ref** (bool) – Whether to already preprocess and store the reference data.
- **update_x_ref** (Optional[Dict[str, int]]) – Reference data can optionally be updated to the last n instances seen by the detector or via reservoir sampling with size

n. For the former, the parameter equals {'last': n} while for reservoir sampling {'reservoir_sampling': n} is passed.

- **preprocess_fn** (Optional[Callable]) – Function to preprocess the data before computing the data drift metrics.
- **sigma** (Optional[ndarray]) – Optionally set the Gaussian RBF kernel bandwidth. Can also pass multiple bandwidth values as an array. The kernel evaluation is then averaged over those bandwidths.
- **configure_kernel_from_x_ref** (bool) – Whether to already configure the kernel bandwidth from the reference data.
- **n_permutations** (int) – Number of permutations used in the permutation test.
- **input_shape** (Optional[tuple]) – Shape of input data.
- **data_type** (Optional[str]) – Optionally specify the data type (tabular, image or time-series). Added to metadata.

Return type None

abstract kernel_matrix (*x*, *y*)

Return type Union[Tensor, Tensor]

predict (*x*, *return_p_val=True*, *return_distance=True*)

Predict whether a batch of data has drifted from the reference data.

Parameters

- **x** (ndarray) – Batch of instances.
- **return_p_val** (bool) – Whether to return the p-value of the permutation test.
- **return_distance** (bool) – Whether to return the MMD metric between the new batch and reference data.

Return type Dict[Dict[str, str], Dict[str, Union[int, float]]]

Returns

- *Dictionary containing 'meta' and 'data' dictionaries.*
- *'meta' has the model's metadata.*
- *'data' contains the drift prediction and optionally the p-value, threshold and MMD metric.*

preprocess (*x*)

Data preprocessing before computing the drift scores. :type x: ndarray :param x: Batch of instances.

Return type Tuple[ndarray, ndarray]

Returns *Preprocessed reference data and new instances.*

abstract score (*x*)

Return type Tuple[float, float, ndarray]

class alibi_detect.cd.base.**BaseUnivariateDrift** (*x_ref*, *p_val=0.05*, *preprocess_x_ref=True*, *update_x_ref=None*, *preprocess_fn=None*, *correction='bonferroni'*, *n_features=None*, *input_shape=None*, *data_type=None*)

Bases: *alibi_detect.base.BaseDetector*

`__init__`(*x_ref*, *p_val=0.05*, *preprocess_x_ref=True*, *update_x_ref=None*, *preprocess_fn=None*, *correction='bonferroni'*, *n_features=None*, *input_shape=None*, *data_type=None*)

Generic drift detector component which serves as a base class for methods using univariate tests with multivariate correction.

Parameters

- **`x_ref`** (`ndarray`) – Data used as reference distribution.
- **`p_val`** (`float`) – p-value used for significance of the statistical test for each feature. If the FDR correction method is used, this corresponds to the acceptable q-value.
- **`preprocess_x_ref`** (`bool`) – Whether to already preprocess and store the reference data.
- **`update_x_ref`** (`Optional`[`Dict`[`str`, `int`]]) – Reference data can optionally be updated to the last n instances seen by the detector or via reservoir sampling with size n. For the former, the parameter equals {'last': n} while for reservoir sampling {'reservoir_sampling': n} is passed.
- **`preprocess_fn`** (`Optional`[`Callable`]) – Function to preprocess the data before computing the data drift metrics. Typically a dimensionality reduction technique.
- **`correction`** (`str`) – Correction type for multivariate data. Either 'bonferroni' or 'fdr' (False Discovery Rate).
- **`n_features`** (`Optional`[`int`]) – Number of features used in the statistical test. No need to pass it if no preprocessing takes place. In case of a preprocessing step, this can also be inferred automatically but could be more expensive to compute.
- **`input_shape`** (`Optional`[`tuple`]) – Shape of input data. Needs to be provided for text data.
- **`data_type`** (`Optional`[`str`]) – Optionally specify the data type (tabular, image or time-series). Added to metadata.

Return type `None`

`abstract feature_score`(*x_ref*, *x*)

Return type `Tuple`[`ndarray`, `ndarray`]

`predict`(*x*, *drift_type='batch'*, *return_p_val=True*, *return_distance=True*)

Predict whether a batch of data has drifted from the reference data.

Parameters

- **`x`** (`ndarray`) – Batch of instances.
- **`drift_type`** (`str`) – Predict drift at the 'feature' or 'batch' level. For 'batch', the test statistics for each feature are aggregated using the Bonferroni or False Discovery Rate correction.
- **`return_p_val`** (`bool`) – Whether to return feature level p-values.
- **`return_distance`** (`bool`) – Whether to return the test statistic between the features of the new batch and reference data.

Return type `Dict`[`Dict`[`str`, `str`], `Dict`[`str`, `Union`[`ndarray`, `int`, `float`]]]

Returns

- *Dictionary containing 'meta' and 'data' dictionaries.*
- *'meta' has the model's metadata.*

- *'data' contains the drift prediction and optionally the feature level p-values,* – threshold after multivariate correction if needed and test statistics.

preprocess (*x*)

Data preprocessing before computing the drift scores.

Parameters **x** (`ndarray`) – Batch of instances.

Return type `Tuple[ndarray, ndarray]`

Returns *Preprocessed reference data and new instances.*

score (*x*)

Compute the feature-wise drift score which is the p-value of the statistical test and the test statistic.

Parameters **x** (`ndarray`) – Batch of instances.

Return type `Tuple[ndarray, ndarray]`

Returns *Feature level p-values and test statistics.*

alibi_detect.cd.chisquare module

class `alibi_detect.cd.chisquare.`**ChiSquareDrift** (*x_ref*, *p_val=0.05*, *categories_per_feature=None*, *preprocess_x_ref=True*, *update_x_ref=None*, *preprocess_fn=None*, *correction='bonferroni'*, *n_features=None*, *input_shape=None*, *data_type=None*)

Bases: *alibi_detect.cd.base.BaseUnivariateDrift*

__init__ (*x_ref*, *p_val=0.05*, *categories_per_feature=None*, *preprocess_x_ref=True*, *update_x_ref=None*, *preprocess_fn=None*, *correction='bonferroni'*, *n_features=None*, *input_shape=None*, *data_type=None*)

Chi-Squared data drift detector with Bonferroni or False Discovery Rate (FDR) correction for multivariate data.

Parameters

- **x_ref** (`ndarray`) – Data used as reference distribution.
- **p_val** (`float`) – p-value used for significance of the Chi-Squared test for each feature. If the FDR correction method is used, this corresponds to the acceptable q-value.
- **categories_per_feature** (`Optional[Dict[int, int]]`) – Optional dictionary with as keys the feature column index and as values the number of possible categorical values for that feature or a list with the possible values. If you know how many categories are present for a given feature you could pass this in the *categories_per_feature* dict in the Dict[int, int] format, e.g. {0: 3, 3: 2}. If you pass N categories this will assume the possible values for the feature are [0, . . . , N-1]. You can also explicitly pass the possible categories in the Dict[int, List[int]] format, e.g. {0: [0, 1, 2], 3: [0, 55]}. Note that the categories can be arbitrary int values. If it is not specified, *categories_per_feature* is inferred from *x_ref*.
- **preprocess_x_ref** (`bool`) – Whether to already preprocess and infer categories and frequencies for reference data.
- **update_x_ref** (`Optional[Dict[str, int]]`) – Reference data can optionally be updated to the last n instances seen by the detector or via reservoir sampling with size n. For the former, the parameter equals {'last': n} while for reservoir sampling {'reservoir_sampling': n} is passed.

- **preprocess_fn** (Optional[Callable]) – Function to preprocess the data before computing the data drift metrics. Typically a dimensionality reduction technique.
- **correction** (str) – Correction type for multivariate data. Either 'bonferroni' or 'fdr' (False Discovery Rate).
- **n_features** (Optional[int]) – Number of features used in the Chi-Squared test. No need to pass it if no preprocessing takes place. In case of a preprocessing step, this can also be inferred automatically but could be more expensive to compute.
- **input_shape** (Optional[tuple]) – Shape of input data.
- **data_type** (Optional[str]) – Optionally specify the data type (tabular, image or time-series). Added to metadata.

Return type `None`

feature_score (*x_ref*, *x*)

Compute Chi-Squared test statistic and p-values per feature.

Parameters

- **x_ref** (ndarray) – Reference instances to compare distribution with.
- **x** (ndarray) – Batch of instances.

Return type Tuple[ndarray, ndarray]

Returns *Feature level p-values and Chi-Squared statistics.*

alibi_detect.cd.classifier module

alibi_detect.cd.ks module

alibi_detect.cd.mmd module

alibi_detect.cd.model_uncertainty module

alibi_detect.cd.preprocess module

alibi_detect.cd.tabular module

alibi_detect.cd.utils module

alibi_detect.cd.utils.**encompass_batching** (*model*, *backend*, *batch_size*, *device=None*, *tokenizer=None*, *max_len=None*)

Takes a function that must be batch evaluated (on tokenized input) and returns a function that handles batching (and tokenization).

Return type `Callable`

alibi_detect.cd.utils.**encompass_shuffling_and_batch_filling** (*model_fn*, *batch_size*)

Takes a function that already handles batching but additionally performing shuffling and ensures instances are evaluated as part of full batches.

Return type `ndarray`

alibi_detect.cd.utils.**update_reference**(*X_ref*, *X*, *n*, *update_method=None*)

Update reference dataset for drift detectors.

Parameters

- **X_ref** (ndarray) – Current reference dataset.
- **X** (ndarray) – New data.
- **n** (int) – Count of the total number of instances that have been used so far.
- **update_method** (Optional[Dict[str, int]]) – Dict with as key *reservoir_sampling* or *last* and as value n. *reservoir_sampling* will apply reservoir sampling with reservoir of size n while *last* will return (at most) the last n instances.

Return type ndarray

Returns *Updated reference dataset.*

alibi_detect.models package

Subpackages

alibi_detect.models.pytorch package

Submodules

alibi_detect.models.pytorch.embedding module

class alibi_detect.models.pytorch.embedding.**TransformerEmbedding**(*model_name_or_path*, *embedding_type*, *layers=None*)

Bases: torch.nn.Module

forward(*tokens*)

Return type Tensor

alibi_detect.models.pytorch.embedding.**hidden_state_embedding**(*hidden_states*, *layers*, *use_cls*, *reduce_mean=True*)

Extract embeddings from hidden attention state layers.

Parameters

- **hidden_states** (Tensor) – Attention hidden states in the transformer model.
- **layers** (List[int]) – List of layers to use for the embedding.
- **use_cls** (bool) – Whether to use the next sentence token (CLS) to extract the embeddings.
- **reduce_mean** (bool) – Whether to take the mean of the output tensor.

Return type Tensor

Returns *Tensor with embeddings.*

alibi_detect.models.pytorch.trainer module

alibi_detect.models.tensorflow package

class alibi_detect.models.tensorflow.**AE**(*encoder_net*, *decoder_net*, *name='ae'*)

Bases: `tensorflow.keras.Model`

__init__(*encoder_net*, *decoder_net*, *name='ae'*)

Combine encoder and decoder in AE.

Parameters

- **encoder_net** (`Sequential`) – Layers for the encoder wrapped in a tf.keras.Sequential class.
- **decoder_net** (`Sequential`) – Layers for the decoder wrapped in a tf.keras.Sequential class.
- **name** (`str`) – Name of autoencoder model.

Return type `None`

call(*x*)

Return type `Tensor`

class alibi_detect.models.tensorflow.**AEGMM**(*encoder_net*, *decoder_net*, *gmm_density_net*, *n_gmm*, *recon_features=<function eucl_cosim_features>*, *name='aegmm'*)

Bases: `tensorflow.keras.Model`

__init__(*encoder_net*, *decoder_net*, *gmm_density_net*, *n_gmm*, *recon_features=<function eucl_cosim_features>*, *name='aegmm'*)

Deep Autoencoding Gaussian Mixture Model.

Parameters

- **encoder_net** (`Sequential`) – Layers for the encoder wrapped in a tf.keras.Sequential class.
- **decoder_net** (`Sequential`) – Layers for the decoder wrapped in a tf.keras.Sequential class.
- **gmm_density_net** (`Sequential`) – Layers for the GMM network wrapped in a tf.keras.Sequential class.
- **n_gmm** (`int`) – Number of components in GMM.
- **recon_features** (`Callable`) – Function to extract features from the reconstructed instance by the decoder.
- **name** (`str`) – Name of the AEGMM model.

Return type `None`

call(*x*)

Return type `Tuple[Tensor, Tensor, Tensor]`

class alibi_detect.models.tensorflow.**Seq2Seq**(*encoder_net*, *decoder_net*, *threshold_net*, *n_features*, *score_fn=tensorflow.math.squared_difference*, *beta=1.0*, *name='seq2seq'*)

Bases: `tensorflow.keras.Model`

__init__(*encoder_net*, *decoder_net*, *threshold_net*, *n_features*, *score_fn=tensorflow.math.squared_difference*, *beta=1.0*, *name='seq2seq'*)

Sequence-to-sequence model.

Parameters

- **encoder_net** (`EncoderLSTM`) – Encoder network.
- **decoder_net** (`DecoderLSTM`) – Decoder network.
- **threshold_net** (`Sequential`) – Regression network used to estimate threshold.
- **n_features** (`int`) – Number of features.
- **score_fn** (`Callable`) – Function used for outlier score.
- **beta** (`float`) – Weight on the threshold estimation loss term.
- **name** (`str`) – Name of the seq2seq model.

Return type `None`

call(*x*)

Forward pass used for teacher-forcing training.

Return type `Tensor`

decode_seq(*x*)

Sequence decoding and threshold estimation used for inference.

class `alibi_detect.models.tensorflow.`**VAE**(*encoder_net*, *decoder_net*, *latent_dim*, *beta=1.0*, *name='vae'*)

Bases: `tensorflow.keras.Model`

__init__(*encoder_net*, *decoder_net*, *latent_dim*, *beta=1.0*, *name='vae'*)

Combine encoder and decoder in VAE.

Parameters

- **encoder_net** (`Sequential`) – Layers for the encoder wrapped in a tf.keras.Sequential class.
- **decoder_net** (`Sequential`) – Layers for the decoder wrapped in a tf.keras.Sequential class.
- **latent_dim** (`int`) – Dimensionality of the latent space.
- **beta** (`float`) – Beta parameter for KL-divergence loss term.
- **name** (`str`) – Name of VAE model.

Return type `None`

call(*x*)

Return type `Tensor`

class `alibi_detect.models.tensorflow.`**VAEGMM**(*encoder_net*, *decoder_net*, *gmm_density_net*, *n_gmm*, *latent_dim*, *recon_features=<function eucl_cosim_features>*, *beta=1.0*, *name='vaegmm'*)

Bases: `tensorflow.keras.Model`

__init__(*encoder_net*, *decoder_net*, *gmm_density_net*, *n_gmm*, *latent_dim*, *recon_features=<function eucl_cosim_features>*, *beta=1.0*, *name='vaegmm'*)

Variational Autoencoding Gaussian Mixture Model.

Parameters

- **encoder_net** (`Sequential`) – Layers for the encoder wrapped in a tf.keras.Sequential class.
- **decoder_net** (`Sequential`) – Layers for the decoder wrapped in a tf.keras.Sequential class.
- **gmm_density_net** (`Sequential`) – Layers for the GMM network wrapped in a tf.keras.Sequential class.
- **n_gmm** (`int`) – Number of components in GMM.
- **latent_dim** (`int`) – Dimensionality of the latent space.
- **recon_features** (`Callable`) – Function to extract features from the reconstructed instance by the decoder.
- **beta** (`float`) – Beta parameter for KL-divergence loss term.
- **name** (`str`) – Name of the VAEGMM model.

Return type `None`

call(*x*)

Return type `Tuple[Tensor, Tensor, Tensor]`

`alibi_detect.models.tensorflow.`**resnet**(*num_blocks*, *classes=10*, *input_shape=(32, 32, 3)*)

Define ResNet.

Parameters

- **num_blocks** (`int`) – Number of ResNet blocks.
- **classes** (`int`) – Number of classification classes.
- **input_shape** (`Tuple[int, int, int]`) – Input shape of an image.

Return type `Model`

Returns *ResNet as a tf.keras.Model.*

class `alibi_detect.models.tensorflow.`**PixelCNN**(*image_shape*, *conditional_shape=None*, *num_resnet=5*, *num_hierarchies=3*, *num_filters=160*, *num_logistic_mix=10*, *receptive_field_dims=(3, 3)*, *dropout_p=0.5*, *resnet_activation='concat_elu'*, *l2_weight=0.0*, *use_weight_norm=True*, *use_data_init=True*, *high=255*, *low=0*, *dtype=tensorflow.compat.v2.float32*, *name='PixelCNN'*)

Bases: `tensorflow_probability.python.distributions.distribution.Distribution`

__init__(*image_shape*, *conditional_shape=None*, *num_resnet=5*, *num_hierarchies=3*, *num_filters=160*, *num_logistic_mix=10*, *receptive_field_dims=(3, 3)*, *dropout_p=0.5*, *resnet_activation='concat_elu'*, *l2_weight=0.0*, *use_weight_norm=True*, *use_data_init=True*, *high=255*, *low=0*, *dtype=tensorflow.compat.v2.float32*, *name='PixelCNN'*)

Construct Pixel CNN++ distribution.

Parameters

- **image_shape** (tuple) – 3D *TensorShape* or tuple for the *[height, width, channels]* dimensions of the image.
- **conditional_shape** (Optional[tuple]) – *TensorShape* or tuple for the shape of the conditional input, or *None* if there is no conditional input.
- **num_resnet** (int) – The number of layers (shown in Figure 2 of [2]) within each highest-level block of Figure 2 of [1].
- **num_hierarchies** (int) – The number of highest-level blocks (separated by expansions/contractions of dimensions in Figure 2 of [1].)
- **num_filters** (int) – The number of convolutional filters.
- **num_logistic_mix** (int) – Number of components in the logistic mixture distribution.
- **receptive_field_dims** (tuple) – Height and width in pixels of the receptive field of the convolutional layers above and to the left of a given pixel. The width (second element of the tuple) should be odd. Figure 1 (middle) of [2] shows a receptive field of (3, 5) (the row containing the current pixel is included in the height). The default of (3, 3) was used to produce the results in [1].
- **dropout_p** (float) – The dropout probability. Should be between 0 and 1.
- **resnet_activation** (str) – The type of activation to use in the resnet blocks. May be 'concat_elu', 'elu', or 'relu'.
- **use_weight_norm** (bool) – If *True* then use weight normalization (works only in Eager mode).
- **use_data_init** (bool) – If *True* then use data-dependent initialization (has no effect if *use_weight_norm* is *False*).
- **high** (int) – The maximum value of the input data (255 for an 8-bit image).
- **low** (int) – The minimum value of the input data.
- **dtype** – Data type of the *Distribution*.
- **name** (str) – The name of the *Distribution*.

Return type None

class alibi_detect.models.tensorflow.**TransformerEmbedding**(*model_name_or_path*, *embedding_type*, *layers=None*)

Bases: `tensorflow.keras.Model`

__init__(*model_name_or_path*, *embedding_type*, *layers=None*)

Extract text embeddings from transformer models.

Parameters

- **model_name_or_path** (str) – Name of or path to the model.
- **embedding_type** (str) – Type of embedding to extract. Needs to be one of pooler_output, last_hidden_state, hidden_state or hidden_state_cls.

  From the HuggingFace documentation: - pooler_output

  Last layer hidden-state of the first token of the sequence (classification token) further processed by a Linear layer and a Tanh activation function. The Linear layer weights are trained from the next sentence prediction (classification) objective during pre-training. This output is usually not a good summary of the semantic content of the input, you're often better with averaging or pooling the sequence of hidden-states for the whole input sequence.

- **last_hidden_state** Sequence of hidden-states at the output of the last layer of the model.
- **hidden_state** Hidden states of the model at the output of each layer.
- **hidden_state_cls** See hidden_state but use the CLS token output.

- **layers** (Optional[List[int]]) – If "hidden_state" or "hidden_state_cls" is used as embedding type, layers has to be a list with int's referring to the hidden layers used to extract the embedding.

Return type None

call(*tokens*)

Return type Tensor

alibi_detect.models.tensorflow.**trainer**(*model*, *loss_fn*, *X_train*, *y_train=None*, *optimizer=tensorflow.keras.optimizers.Adam*, *loss_fn_kwargs=None*, *preprocess_fn=None*, *epochs=20*, *batch_size=64*, *buffer_size=1024*, *verbose=True*, *log_metric=None*, *callbacks=None*)

Train TensorFlow model.

Parameters

- **model** – Model to train.
- **loss_fn** – Loss function used for training.
- **X_train** – Training batch.
- **y_train** – Training labels.
- **optimizer** – Optimizer used for training.
- **loss_fn_kwargs** – Kwargs for loss function.
- **preprocess_fn** – Preprocessing function applied to each training batch.
- **epochs** – Number of training epochs.
- **batch_size** – Batch size used for training.
- **buffer_size** – Maximum number of elements that will be buffered when prefetching.
- **verbose** – Whether to print training progress.
- **log_metric** – Additional metrics whose progress will be displayed if verbose equals True.
- **callbacks** – Callbacks used during training.

Submodules

alibi_detect.models.tensorflow.autoencoder module

class `alibi_detect.models.tensorflow.autoencoder.`**`AE`**(*encoder_net*, *decoder_net*, *name='ae'*)

Bases: `tensorflow.keras.Model`

`__init__`(*encoder_net*, *decoder_net*, *name='ae'*)

Combine encoder and decoder in AE.

Parameters

- **encoder_net** (`Sequential`) – Layers for the encoder wrapped in a tf.keras.Sequential class.
- **decoder_net** (`Sequential`) – Layers for the decoder wrapped in a tf.keras.Sequential class.
- **name** (`str`) – Name of autoencoder model.

Return type `None`

`call`(*x*)

Return type `Tensor`

class `alibi_detect.models.tensorflow.autoencoder.`**`AEGMM`**(*encoder_net*, *decoder_net*, *gmm_density_net*, *n_gmm*, *recon_features=<function eucl_cosim_features>*, *name='aegmm'*)

Bases: `tensorflow.keras.Model`

`__init__`(*encoder_net*, *decoder_net*, *gmm_density_net*, *n_gmm*, *recon_features=<function eucl_cosim_features>*, *name='aegmm'*)

Deep Autoencoding Gaussian Mixture Model.

Parameters

- **encoder_net** (`Sequential`) – Layers for the encoder wrapped in a tf.keras.Sequential class.
- **decoder_net** (`Sequential`) – Layers for the decoder wrapped in a tf.keras.Sequential class.
- **gmm_density_net** (`Sequential`) – Layers for the GMM network wrapped in a tf.keras.Sequential class.
- **n_gmm** (`int`) – Number of components in GMM.
- **recon_features** (`Callable`) – Function to extract features from the reconstructed instance by the decoder.
- **name** (`str`) – Name of the AEGMM model.

Return type `None`

`call`(*x*)

Return type `Tuple[Tensor, Tensor, Tensor]`

class `alibi_detect.models.tensorflow.autoencoder.`**`Decoder`**(*decoder_net*, *name='decoder'*)

Bases: `tensorflow.keras.layers.Layer`

__init__(*decoder_net*, *name='decoder'*)
Decoder of (V)AE.

Parameters

- **decoder_net** (Sequential) – Layers for the decoder wrapped in a tf.keras.Sequential class.
- **name** (str) – Name of decoder.

Return type None

call(*x*)

Return type Tensor

class alibi_detect.models.tensorflow.autoencoder.**DecoderLSTM**(*latent_dim*, *output_dim*, *output_activation=None*, *name='decoder_lstm'*)
Bases: tensorflow.keras.layers.Layer

__init__(*latent_dim*, *output_dim*, *output_activation=None*, *name='decoder_lstm'*)
LSTM decoder.

Parameters

- **latent_dim** (int) – Latent dimension.
- **output_dim** (int) – Decoder output dimension.
- **output_activation** (Optional[str]) – Activation used in the Dense output layer.
- **name** (str) – Name of decoder.

Return type None

call(*x*, *init_state*)

Return type Tuple[Tensor, Tensor, List[Tensor]]

class alibi_detect.models.tensorflow.autoencoder.**EncoderAE**(*encoder_net*, *name='encoder_ae'*)
Bases: tensorflow.keras.layers.Layer

__init__(*encoder_net*, *name='encoder_ae'*)
Encoder of AE.

Parameters

- **encoder_net** (Sequential) – Layers for the encoder wrapped in a tf.keras.Sequential class.
- **name** (str) – Name of encoder.

Return type None

call(*x*)

Return type Tensor

class alibi_detect.models.tensorflow.autoencoder.**EncoderLSTM**(*latent_dim*, *name='encoder_lstm'*)
Bases: tensorflow.keras.layers.Layer

__init__(*latent_dim*, *name='encoder_lstm'*)
Bidirectional LSTM encoder.

Parameters

- **latent_dim** (`int`) – Latent dimension. Must be an even number given the bidirectional encoder.
- **name** (`str`) – Name of encoder.

Return type `None`

call(*x*)

Return type `Tuple[Tensor, List[Tensor]]`

class `alibi_detect.models.tensorflow.autoencoder.`**EncoderVAE**(*encoder_net*, *latent_dim*, *name='encoder_vae'*)

Bases: `tensorflow.keras.layers.Layer`

__init__(*encoder_net*, *latent_dim*, *name='encoder_vae'*)

Encoder of VAE.

Parameters

- **encoder_net** (`Sequential`) – Layers for the encoder wrapped in a tf.keras.Sequential class.
- **latent_dim** (`int`) – Dimensionality of the latent space.
- **name** (`str`) – Name of encoder.

Return type `None`

call(*x*)

Return type `Tuple[Tensor, Tensor, Tensor]`

class `alibi_detect.models.tensorflow.autoencoder.`**Sampling**(**args*, ***kwargs*)

Bases: `tensorflow.keras.layers.Layer`

Reparametrization trick. Uses (z_mean, z_log_var) to sample the latent vector z.

call(*inputs*)

Sample z.

Parameters **inputs** (`Tuple[Tensor, Tensor]`) – Tuple with mean and log variance.

Return type `Tensor`

Returns *Sampled vector z.*

class `alibi_detect.models.tensorflow.autoencoder.`**Seq2Seq**(*encoder_net*, *decoder_net*, *threshold_net*, *n_features*, *score_fn=tensorflow.math.squared_difference*, *beta=1.0*, *name='seq2seq'*)

Bases: `tensorflow.keras.Model`

__init__(*encoder_net*, *decoder_net*, *threshold_net*, *n_features*, *score_fn=tensorflow.math.squared_difference*, *beta=1.0*, *name='seq2seq'*)

Sequence-to-sequence model.

Parameters

- **encoder_net** (*EncoderLSTM*) – Encoder network.
- **decoder_net** (*DecoderLSTM*) – Decoder network.

- **threshold_net** (`Sequential`) – Regression network used to estimate threshold.
- **n_features** (`int`) – Number of features.
- **score_fn** (`Callable`) – Function used for outlier score.
- **beta** (`float`) – Weight on the threshold estimation loss term.
- **name** (`str`) – Name of the seq2seq model.

Return type `None`

call(*x*)
Forward pass used for teacher-forcing training.

Return type `Tensor`

decode_seq(*x*)
Sequence decoding and threshold estimation used for inference.

class `alibi_detect.models.tensorflow.autoencoder.`**VAE**(*encoder_net, decoder_net, latent_dim, beta=1.0, name='vae'*)
Bases: `tensorflow.keras.Model`

__init__(*encoder_net, decoder_net, latent_dim, beta=1.0, name='vae'*)
Combine encoder and decoder in VAE.

Parameters

- **encoder_net** (`Sequential`) – Layers for the encoder wrapped in a tf.keras.Sequential class.
- **decoder_net** (`Sequential`) – Layers for the decoder wrapped in a tf.keras.Sequential class.
- **latent_dim** (`int`) – Dimensionality of the latent space.
- **beta** (`float`) – Beta parameter for KL-divergence loss term.
- **name** (`str`) – Name of VAE model.

Return type `None`

call(*x*)

Return type `Tensor`

class `alibi_detect.models.tensorflow.autoencoder.`**VAEGMM**(*encoder_net, decoder_net, gmm_density_net, n_gmm, latent_dim, recon_features=<function eucl_cosim_features>, beta=1.0, name='vaegmm'*)
Bases: `tensorflow.keras.Model`

__init__(*encoder_net, decoder_net, gmm_density_net, n_gmm, latent_dim, recon_features=<function eucl_cosim_features>, beta=1.0, name='vaegmm'*)
Variational Autoencoding Gaussian Mixture Model.

Parameters

- **encoder_net** (`Sequential`) – Layers for the encoder wrapped in a tf.keras.Sequential class.

- **decoder_net** (Sequential) – Layers for the decoder wrapped in a tf.keras.Sequential class.
- **gmm_density_net** (Sequential) – Layers for the GMM network wrapped in a tf.keras.Sequential class.
- **n_gmm** (int) – Number of components in GMM.
- **latent_dim** (int) – Dimensionality of the latent space.
- **recon_features** (Callable) – Function to extract features from the reconstructed instance by the decoder.
- **beta** (float) – Beta parameter for KL-divergence loss term.
- **name** (str) – Name of the VAEGMM model.

Return type None

call(*x*)

Return type Tuple[Tensor, Tensor, Tensor]

alibi_detect.models.tensorflow.autoencoder.**eucl_cosim_features**(*x*, *y*, *max_eucl=100.0*)

Compute features extracted from the reconstructed instance using the relative Euclidean distance and cosine similarity between 2 tensors.

Parameters

- **x** (Tensor) – Tensor used in feature computation.
- **y** (Tensor) – Tensor used in feature computation.
- **max_eucl** (float) – Maximum value to clip relative Euclidean distance by.

Return type Tensor

Returns

- *Tensor concatenating the relative Euclidean distance and*
- *cosine similarity features.*

alibi_detect.models.tensorflow.embedding module

class alibi_detect.models.tensorflow.embedding.**TransformerEmbedding**(*model_name_or_path*, *embedding_type*, *layers=None*)

Bases: tensorflow.keras.Model

__init__(*model_name_or_path*, *embedding_type*, *layers=None*)

Extract text embeddings from transformer models.

Parameters

- **model_name_or_path** (str) – Name of or path to the model.
- **embedding_type** (str) – Type of embedding to extract. Needs to be one of pooler_output, last_hidden_state, hidden_state or hidden_state_cls.

  From the HuggingFace documentation: - pooler_output

Last layer hidden-state of the first token of the sequence (classification token) further processed by a Linear layer and a Tanh activation function. The Linear layer weights are trained from the next sentence prediction (classification) objective during pre-training. This output is usually not a good summary of the semantic content of the input, you're often better with averaging or pooling the sequence of hidden-states for the whole input sequence.

– **last_hidden_state** Sequence of hidden-states at the output of the last layer of the model.

– **hidden_state** Hidden states of the model at the output of each layer.

– **hidden_state_cls** See hidden_state but use the CLS token output.

- **layers** (`Optional[List[int]]`) – If "hidden_state" or "hidden_state_cls" is used as embedding type, layers has to be a list with int's referring to the hidden layers used to extract the embedding.

Return type `None`

call(*tokens*)

Return type `Tensor`

alibi_detect.models.tensorflow.embedding.**hidden_state_embedding**(*hidden_states*, *layers*, *use_cls*, *reduce_mean=True*)

Extract embeddings from hidden attention state layers.

Parameters

- **hidden_states** (`Tensor`) – Attention hidden states in the transformer model.
- **layers** (`List[int]`) – List of layers to use for the embedding.
- **use_cls** (`bool`) – Whether to use the next sentence token (CLS) to extract the embeddings.
- **reduce_mean** (`bool`) – Whether to take the mean of the output tensor.

Return type `Tensor`

Returns *Tensor with embeddings.*

alibi_detect.models.tensorflow.gmm module

alibi_detect.models.tensorflow.gmm.**gmm_energy**(*z*, *phi*, *mu*, *cov*, *L*, *log_det_cov*, *return_mean=True*)

Compute sample energy from Gaussian Mixture Model.

Parameters

- **z** (`Tensor`) – Observations.
- **phi** (`Tensor`) – Mixture component distribution weights.
- **mu** (`Tensor`) – Mixture means.
- **cov** (`Tensor`) – Mixture covariance.
- **L** (`Tensor`) – Cholesky decomposition of *cov*.

- **log_det_cov** (Tensor) – Log of the determinant of *cov*.
- **return_mean** (bool) – Take mean across all sample energies in a batch.

Return type Tuple[Tensor, Tensor]

Returns

- *sample_energy* – The sample energy of the GMM.
- *cov_diag* – The inverse sum of the diagonal components of the covariance matrix.

alibi_detect.models.tensorflow.gmm.**gmm_params**(*z*, *gamma*)

Compute parameters of Gaussian Mixture Model.

Parameters

- **z** (Tensor) – Observations.
- **gamma** (Tensor) – Mixture probabilities to derive mixture distribution weights from.

Return type Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]

Returns

- *phi* – Mixture component distribution weights.
- *mu* – Mixture means.
- *cov* – Mixture covariance.
- *L* – Cholesky decomposition of *cov*.
- *log_det_cov* – Log of the determinant of *cov*.

alibi_detect.models.tensorflow.losses module

alibi_detect.models.tensorflow.losses.**elbo**(*y_true*, *y_pred*, *cov_full=None*, *cov_diag=None*, *sim=0.05*)

Compute ELBO loss.

Parameters

- **y_true** (Tensor) – Labels.
- **y_pred** (Tensor) – Predictions.
- **cov_full** (Optional[Tensor]) – Full covariance matrix.
- **cov_diag** (Optional[Tensor]) – Diagonal (variance) of covariance matrix.
- **sim** (float) – Scale identity multiplier.

Return type Tensor

Returns *ELBO loss value.*

alibi_detect.models.tensorflow.losses.**loss_adv_ae**(*x_true*, *x_pred*, *model=None*, *model_hl=None*, *w_model=1.0*, *w_recon=0.0*, *w_model_hl=None*, *temperature=1.0*)

Loss function used for AdversarialAE.

Parameters

- **x_true** (Tensor) – Batch of instances.

- **x_pred** (Tensor) – Batch of reconstructed instances by the autoencoder.
- **model** (Optional[Model]) – A trained tf.keras model with frozen layers (layers.trainable = False).
- **model_hl** (Optional[list]) – List with tf.keras models used to extract feature maps and make predictions on hidden layers.
- **w_model** (float) – Weight on model prediction loss term.
- **w_recon** (float) – Weight on MSE reconstruction error loss term.
- **w_model_hl** (Optional[list]) – Weights assigned to the loss of each model in model_hl.
- **temperature** (float) – Temperature used for model prediction scaling. Temperature <1 sharpens the prediction probability distribution.

Return type Tensor

Returns *Loss value.*

alibi_detect.models.tensorflow.losses.**loss_aegmm**(*x_true*, *x_pred*, *z*, *gamma*, *w_energy=0.1*, *w_cov_diag=0.005*)

Loss function used for OutlierAEGMM.

Parameters

- **x_true** (Tensor) – Batch of instances.
- **x_pred** (Tensor) – Batch of reconstructed instances by the autoencoder.
- **z** (Tensor) – Latent space values.
- **gamma** (Tensor) – Membership prediction for mixture model components.
- **w_energy** (float) – Weight on sample energy loss term.
- **w_cov_diag** (float) – Weight on covariance regularizing loss term.

Return type Tensor

Returns *Loss value.*

alibi_detect.models.tensorflow.losses.**loss_distillation**(*x_true*, *y_pred*, *model=None*, *loss_type='kld'*, *temperature=1.0*)

Loss function used for Model Distillation.

Parameters

- **x_true** (Tensor) – Batch of data points.
- **y_pred** (Tensor) – Batch of prediction from the distilled model.
- **model** (Optional[Model]) – tf.keras model.
- **loss_type** (str) – Type of loss for distillation. Supported 'kld', 'xent.
- **temperature** (float) – Temperature used for model prediction scaling. Temperature <1 sharpens the prediction probability distribution.

Return type Tensor

Returns *Loss value.*

alibi_detect.models.tensorflow.losses.**loss_vaegmm**(*x_true*, *x_pred*, *z*, *gamma*, *w_recon=1e-07*, *w_energy=0.1*, *w_cov_diag=0.005*, *cov_full=None*, *cov_diag=None*, *sim=0.05*)

Loss function used for OutlierVAEGMM.

Parameters

- **x_true** (Tensor) – Batch of instances.
- **x_pred** (Tensor) – Batch of reconstructed instances by the variational autoencoder.
- **z** (Tensor) – Latent space values.
- **gamma** (Tensor) – Membership prediction for mixture model components.
- **w_recon** (float) – Weight on elbo loss term.
- **w_energy** (float) – Weight on sample energy loss term.
- **w_cov_diag** (float) – Weight on covariance regularizing loss term.
- **cov_full** (Optional[Tensor]) – Full covariance matrix.
- **cov_diag** (Optional[Tensor]) – Diagonal (variance) of covariance matrix.
- **sim** (float) – Scale identity multiplier.

Return type Tensor

Returns *Loss value.*

alibi_detect.models.tensorflow.pixelcnn module

class alibi_detect.models.tensorflow.pixelcnn.**Shift**(*shift*, *validate_args=False*, *name='shift'*)

Bases: tensorflow_probability.python.bijectors.bijector.Bijector

__init__(*shift*, *validate_args=False*, *name='shift'*)

Instantiates the *Shift* bijector which computes *Y = g(X; shift) = X + shift* where *shift* is a numeric *Tensor*. :param shift: Floating-point *Tensor*. :param validate_args: Python *bool* indicating whether arguments should be

checked for correctness.

Parameters **name** – Python *str* name given to ops managed by this object.

property shift

The *shift Tensor* in *Y = X + shift*.

alibi_detect.models.tensorflow.resnet module

class alibi_detect.models.tensorflow.resnet.**LearningRateBatchScheduler**(*schedule*, *batch_size*, *steps_per_epoch*)

Bases: tensorflow.keras.callbacks.Callback

__init__(*schedule*, *batch_size*, *steps_per_epoch*)

Callback to update learning rate on every batch instead of epoch.

Parameters

- **schedule** (Callable) – Function taking the epoch and batch index as input which returns the new learning rate as output.
- **batch_size** (int) – Batch size.
- **steps_per_epoch** (int) – Number of batches or steps per epoch.

on_batch_begin(*batch*, *logs=None*)
Executes before step begins.

on_epoch_begin(*epoch*, *logs=None*)

alibi_detect.models.tensorflow.resnet.**conv_block**(*x_in*, *filters*, *kernel_size*, *stage*, *block*, *strides=(2, 2)*, *l2_regularisation=True*)
Conv block in ResNet with a parameterised skip connection to reduce the width and height controlled by the strides.

Parameters

- **x_in** (Tensor) – Input Tensor.
- **filters** (Tuple[int, int]) – Number of filters for each of the 2 conv layers.
- **kernel_size** (Union[int, list, Tuple[int]]) – Kernel size for the conv layers.
- **stage** (int) – Stage of the block in the ResNet.
- **block** (str) – Block within a stage in the ResNet.
- **strides** (Tuple[int, int]) – Stride size applied to reduce the image size.
- **l2_regularisation** (bool) – Whether to apply L2 regularisation.

Return type Tensor

Returns *Output Tensor of the conv block.*

alibi_detect.models.tensorflow.resnet.**identity_block**(*x_in*, *filters*, *kernel_size*, *stage*, *block*, *l2_regularisation=True*)
Identity block in ResNet.

Parameters

- **x_in** (Tensor) – Input Tensor.
- **filters** (Tuple[int, int]) – Number of filters for each of the 2 conv layers.
- **kernel_size** (Union[int, list, Tuple[int]]) – Kernel size for the conv layers.
- **stage** (int) – Stage of the block in the ResNet.
- **block** (str) – Block within a stage in the ResNet.
- **l2_regularisation** (bool) – Whether to apply L2 regularisation.

Return type Tensor

Returns *Output Tensor of the identity block.*

alibi_detect.models.tensorflow.resnet.**l2_regulariser**(*l2_regularisation=True*)
Apply L2 regularisation to kernel.

Parameters **l2_regularisation** (bool) – Whether to apply L2 regularisation.

Returns *Kernel regularisation.*

alibi_detect.models.tensorflow.resnet.**learning_rate_schedule**(*current_epoch*, *current_batch*, *batches_per_epoch*, *batch_size*)

Linear learning rate scaling and learning rate decay at specified epochs.

Parameters

- **current_epoch** (int) – Current training epoch.
- **current_batch** (int) – Current batch with current epoch, not used.
- **batches_per_epoch** (int) – Number of batches or steps in an epoch, not used.
- **batch_size** (int) – Batch size.

Return type float

Returns *Adjusted learning rate.*

alibi_detect.models.tensorflow.resnet.**preprocess_image**(*x*, *is_training=True*)

Return type ndarray

alibi_detect.models.tensorflow.resnet.**resnet**(*num_blocks*, *classes=10*, *input_shape=(32*, *32*, *3)*)

Define ResNet.

Parameters

- **num_blocks** (int) – Number of ResNet blocks.
- **classes** (int) – Number of classification classes.
- **input_shape** (Tuple[int, int, int]) – Input shape of an image.

Return type Model

Returns *ResNet as a tf.keras.Model.*

alibi_detect.models.tensorflow.resnet.**resnet_block**(*x_in*, *size*, *filters*, *kernel_size*, *stage*, *strides=(2*, *2)*, *l2_regularisation=True*)

Block in ResNet combining a conv block with identity blocks.

Parameters

- **x_in** (Tensor) – Input Tensor.
- **size** (int) – The ResNet block consists of 1 conv block and size-1 identity blocks.
- **filters** (Tuple[int, int]) – Number of filters for each of the conv layers.
- **kernel_size** (Union[int, list, Tuple[int]]) – Kernel size for the conv layers.
- **stage** (int) – Stage of the block in the ResNet.
- **strides** (Tuple[int, int]) – Stride size applied to reduce the image size.
- **l2_regularisation** (bool) – Whether to apply L2 regularisation.

Return type Tensor

Returns *Output Tensor of the conv block.*

alibi_detect.models.tensorflow.resnet.**run**(*num_blocks*, *epochs*, *batch_size*, *model_dir*, *num_classes=10*, *input_shape=(32*, *32*, *3)*, *validation_freq=10*, *verbose=2*, *seed=1*, *serving=False*)

Return type None

alibi_detect.models.tensorflow.resnet.**scale_by_instance**(*x*, *eps=1e-12*)

Return type ndarray

alibi_detect.models.tensorflow.trainer module

alibi_detect.models.tensorflow.trainer.**trainer**(*model*, *loss_fn*, *X_train*, *y_train=None*, *optimizer=tensorflow.keras.optimizers.Adam*, *loss_fn_kwargs=None*, *preprocess_fn=None*, *epochs=20*, *batch_size=64*, *buffer_size=1024*, *verbose=True*, *log_metric=None*, *callbacks=None*)

Train TensorFlow model.

Parameters

- **model** – Model to train.
- **loss_fn** – Loss function used for training.
- **X_train** – Training batch.
- **y_train** – Training labels.
- **optimizer** – Optimizer used for training.
- **loss_fn_kwargs** – Kwargs for loss function.
- **preprocess_fn** – Preprocessing function applied to each training batch.
- **epochs** – Number of training epochs.
- **batch_size** – Batch size used for training.
- **buffer_size** – Maximum number of elements that will be buffered when prefetching.
- **verbose** – Whether to print training progress.
- **log_metric** – Additional metrics whose progress will be displayed if verbose equals True.
- **callbacks** – Callbacks used during training.

alibi_detect.od package

Submodules

alibi_detect.od.ae module

alibi_detect.od.aegmm module

alibi_detect.od.isolationforest module

alibi_detect.od.llr module

alibi_detect.od.mahalanobis module

alibi_detect.od.prophet module

alibi_detect.od.seq2seq module

alibi_detect.od.sr module

alibi_detect.od.vae module

alibi_detect.od.vaegmm module

alibi_detect.utils package

Subpackages

alibi_detect.utils.pytorch package

`alibi_detect.utils.pytorch.`**`mmd2`**(*x*, *y*, *kernel*)

Compute MMD^2 between 2 samples.

Parameters

- **`x`** (`Tensor`) – Batch of instances of shape [Nx, features].
- **`y`** (`Tensor`) – Batch of instances of shape [Ny, features].
- **`kernel`** (`Callable`) – Kernel function.

Return type `float`

Returns *MMD^2 between the samples x and y.*

`alibi_detect.utils.pytorch.`**`mmd2_from_kernel_matrix`**(*kernel_mat*, *m*, *permute=False*, *zero_diag=True*)

Compute maximum mean discrepancy (MMD^2) between 2 samples x and y from the full kernel matrix between the samples.

Parameters

- **`kernel_mat`** (`Tensor`) – Kernel matrix between samples x and y.
- **`m`** (`int`) – Number of instances in y.
- **`permute`** (`bool`) – Whether to permute the row indices. Used for permutation tests.
- **`zero_diag`** (`bool`) – Whether to zero out the diagonal of the kernel matrix.

Return type `Tensor`

Returns *MMD^2 between the samples from the kernel matrix.*

`alibi_detect.utils.pytorch.`**`squared_pairwise_distance`**(*x*, *y*, *a_min=1e-30*)

PyTorch pairwise squared Euclidean distance between samples x and y.

Parameters

- **`x`** (`Tensor`) – Batch of instances of shape [Nx, features].
- **`y`** (`Tensor`) – Batch of instances of shape [Ny, features].

- **a_min** (float) – Lower bound to clip distance values.

Return type Tensor

Returns *Pairwise squared Euclidean distance [Nx, Ny].*

class alibi_detect.utils.pytorch.**GaussianRBF**(*sigma=None*)

Bases: object

__init__(*sigma=None*)

Gaussian RBF kernel: k(x,y) = exp(-(1/(2*sigma^2)||x-y||^2). A forward pass takes a batch of instances x [Nx, features] and y [Ny, features] and returns the kernel matrix [Nx, Ny].

Parameters **sigma** (Optional[Tensor]) – Optional sigma used for the kernel.

Return type None

alibi_detect.utils.pytorch.**predict_batch**(*x, model, device=None, batch_size=10000000000, dtype=numpy.float32*)

Make batch predictions on a model.

Parameters

- **x** (Union[ndarray, Tensor]) – Batch of instances.
- **model** (Union[Module, Sequential]) – PyTorch model.
- **device** (Optional[device]) – Device type used. The default None tries to use the GPU and falls back on CPU if needed. Can be specified by passing either torch.device('cuda') or torch.device('cpu').
- **batch_size** (int) – Batch size used during prediction.
- **dtype** (Union[float32, dtype]) – Model output type, e.g. np.float32 or torch.float32.

Return type Union[ndarray, Tensor]

Returns *Numpy array or torch tensor with model outputs.*

alibi_detect.utils.pytorch.**predict_batch_transformer**(*x, model, tokenizer, max_len, device=None, batch_size=10000000000, dtype=numpy.float32*)

Make batch predictions using a transformers tokenizer and model.

Parameters

- **x** (Union[ndarray, Tensor]) – Batch of instances.
- **model** (Union[Module, Sequential]) – PyTorch model.
- **tokenizer** (Callable) – Tokenizer for model.
- **max_len** (int) – Max sequence length for tokens.
- **device** (Optional[device]) – Device type used. The default None tries to use the GPU and falls back on CPU if needed. Can be specified by passing either torch.device('cuda') or torch.device('cpu').
- **batch_size** (int) – Batch size used during prediction.
- **dtype** (Union[float32, dtype]) – Model output type, e.g. np.float32 or torch.float32.

Return type Union[ndarray, Tensor]

Returns *Numpy array or torch tensor with model outputs.*

Submodules

alibi_detect.utils.pytorch.distance module

`alibi_detect.utils.pytorch.distance.`**`mmd2`**(*x*, *y*, *kernel*)

Compute MMD^2 between 2 samples.

Parameters

- **`x`** (`Tensor`) – Batch of instances of shape [Nx, features].
- **`y`** (`Tensor`) – Batch of instances of shape [Ny, features].
- **`kernel`** (`Callable`) – Kernel function.

Return type `float`

Returns *MMD^2 between the samples x and y.*

`alibi_detect.utils.pytorch.distance.`**`mmd2_from_kernel_matrix`**(*kernel_mat*, *m*, *permute=False*, *zero_diag=True*)

Compute maximum mean discrepancy (MMD^2) between 2 samples x and y from the full kernel matrix between the samples.

Parameters

- **`kernel_mat`** (`Tensor`) – Kernel matrix between samples x and y.
- **`m`** (`int`) – Number of instances in y.
- **`permute`** (`bool`) – Whether to permute the row indices. Used for permutation tests.
- **`zero_diag`** (`bool`) – Whether to zero out the diagonal of the kernel matrix.

Return type `Tensor`

Returns *MMD^2 between the samples from the kernel matrix.*

`alibi_detect.utils.pytorch.distance.`**`squared_pairwise_distance`**(*x*, *y*, *a_min=1e-30*)

PyTorch pairwise squared Euclidean distance between samples x and y.

Parameters

- **`x`** (`Tensor`) – Batch of instances of shape [Nx, features].
- **`y`** (`Tensor`) – Batch of instances of shape [Ny, features].
- **`a_min`** (`float`) – Lower bound to clip distance values.

Return type `Tensor`

Returns *Pairwise squared Euclidean distance [Nx, Ny].*

alibi_detect.utils.pytorch.kernels module

class alibi_detect.utils.pytorch.kernels.**GaussianRBF**(*sigma=None*)

Bases: `object`

__init__(*sigma=None*)

Gaussian RBF kernel: k(x,y) = exp(-(1/(2*sigma^2)||x-y||^2). A forward pass takes a batch of instances x [Nx, features] and y [Ny, features] and returns the kernel matrix [Nx, Ny].

Parameters **sigma** (`Optional[Tensor]`) – Optional sigma used for the kernel.

Return type `None`

alibi_detect.utils.pytorch.prediction module

alibi_detect.utils.pytorch.prediction.**predict_batch**(*x, model, device=None, batch_size=10000000000, dtype=numpy.float32*)

Make batch predictions on a model.

Parameters

- **x** (`Union[ndarray, Tensor]`) – Batch of instances.
- **model** (`Union[Module, Sequential]`) – PyTorch model.
- **device** (`Optional[device]`) – Device type used. The default None tries to use the GPU and falls back on CPU if needed. Can be specified by passing either torch.device('cuda') or torch.device('cpu').
- **batch_size** (`int`) – Batch size used during prediction.
- **dtype** (`Union[float32, dtype]`) – Model output type, e.g. np.float32 or torch.float32.

Return type `Union[ndarray, Tensor]`

Returns *Numpy array or torch tensor with model outputs.*

alibi_detect.utils.pytorch.prediction.**predict_batch_transformer**(*x, model, tokenizer, max_len, device=None, batch_size=10000000000, dtype=numpy.float32*)

Make batch predictions using a transformers tokenizer and model.

Parameters

- **x** (`Union[ndarray, Tensor]`) – Batch of instances.
- **model** (`Union[Module, Sequential]`) – PyTorch model.
- **tokenizer** (`Callable`) – Tokenizer for model.
- **max_len** (`int`) – Max sequence length for tokens.
- **device** (`Optional[device]`) – Device type used. The default None tries to use the GPU and falls back on CPU if needed. Can be specified by passing either torch.device('cuda') or torch.device('cpu').
- **batch_size** (`int`) – Batch size used during prediction.

- **dtype** (Union[float32, dtype]) – Model output type, e.g. np.float32 or torch.float32.

Return type Union[ndarray, Tensor]

Returns *Numpy array or torch tensor with model outputs.*

alibi_detect.utils.tensorflow package

alibi_detect.utils.tensorflow.**mmd2**(*x*, *y*, *kernel*)

Compute MMD^2 between 2 samples.

Parameters

- **x** (Tensor) – Batch of instances of shape [Nx, features].
- **y** (Tensor) – Batch of instances of shape [Ny, features].
- **kernel** (Callable) – Kernel function.

Return type float

Returns *MMD^2 between the samples x and y.*

alibi_detect.utils.tensorflow.**mmd2_from_kernel_matrix**(*kernel_mat*, *m*, *permute=False*, *zero_diag=True*)

Compute maximum mean discrepancy (MMD^2) between 2 samples x and y from the full kernel matrix between the samples.

Parameters

- **kernel_mat** (Tensor) – Kernel matrix between samples x and y.
- **m** (int) – Number of instances in y.
- **permute** (bool) – Whether to permute the row indices. Used for permutation tests.
- **zero_diag** (bool) – Whether to zero out the diagonal of the kernel matrix.

Return type Tensor

Returns *MMD^2 between the samples from the kernel matrix.*

alibi_detect.utils.tensorflow.**relative_euclidean_distance**(*x*, *y*, *eps=1e-12*, *axis=-1*)

Relative Euclidean distance.

Parameters

- **x** (Tensor) – Tensor used in distance computation.
- **y** (Tensor) – Tensor used in distance computation.
- **eps** (float) – Epsilon added to denominator for numerical stability.
- **axis** (int) – Axis used to compute distance.

Return type Tensor

Returns *Tensor with relative Euclidean distance across specified axis.*

alibi_detect.utils.tensorflow.**squared_pairwise_distance**(*x*, *y*, *a_min=1e-30*, *a_max=1e+30*)

TensorFlow pairwise squared Euclidean distance between samples x and y.

Parameters

- **x** (Tensor) – Batch of instances of shape [Nx, features].
- **y** (Tensor) – Batch of instances of shape [Ny, features].
- **a_min** (float) – Lower bound to clip distance values.
- **a_max** (float) – Upper bound to clip distance values.

Return type Tensor

Returns *Pairwise squared Euclidean distance [Nx, Ny].*

class alibi_detect.utils.tensorflow.**GaussianRBF**(*sigma=None*)

Bases: object

__init__(*sigma=None*)

Gaussian RBF kernel: k(x,y) = exp(-(1/(2*sigma^2)||x-y||^2). A forward pass takes a batch of instances x [Nx, features] and y [Ny, features] and returns the kernel matrix [Nx, Ny].

Parameters **sigma** (Optional[Tensor]) – Optional sigma used for the kernel.

Return type None

alibi_detect.utils.tensorflow.**predict_batch**(*x, model, batch_size=10000000000, dtype=numpy.float32*)

Make batch predictions on a model.

Parameters

- **x** (Union[ndarray, Tensor]) – Batch of instances.
- **model** (Model) – tf.keras model or one of the other permitted types defined in Data.
- **batch_size** (int) – Batch size used during prediction.
- **dtype** (Union[float32, DType]) – Model output type, e.g. np.float32 or tf.float32.

Return type Union[ndarray, Tensor]

Returns *Numpy array or tensorflow tensor with model outputs.*

alibi_detect.utils.tensorflow.**predict_batch_transformer**(*x, model, tokenizer, max_len, batch_size=10000000000, dtype=numpy.float32*)

Make batch predictions using a transformers tokenizer and model.

Parameters

- **x** (ndarray) – Batch of instances.
- **model** (Model) – Transformer model.
- **tokenizer** – Tokenizer for model.
- **max_len** (int) – Max token length.
- **batch_size** (int) – Batch size.

Return type Union[ndarray, Tensor]

Returns *Numpy array or tensorflow tensor with model outputs.*

Submodules

alibi_detect.utils.tensorflow.distance module

`alibi_detect.utils.tensorflow.distance.`**`mmd2`**(*x*, *y*, *kernel*)

Compute MMD^2 between 2 samples.

Parameters

- **`x`** (`Tensor`) – Batch of instances of shape [Nx, features].
- **`y`** (`Tensor`) – Batch of instances of shape [Ny, features].
- **`kernel`** (`Callable`) – Kernel function.

Return type `float`

Returns *MMD^2 between the samples x and y.*

`alibi_detect.utils.tensorflow.distance.`**`mmd2_from_kernel_matrix`**(*kernel_mat*, *m*, *permute=False*, *zero_diag=True*)

Compute maximum mean discrepancy (MMD^2) between 2 samples x and y from the full kernel matrix between the samples.

Parameters

- **`kernel_mat`** (`Tensor`) – Kernel matrix between samples x and y.
- **`m`** (`int`) – Number of instances in y.
- **`permute`** (`bool`) – Whether to permute the row indices. Used for permutation tests.
- **`zero_diag`** (`bool`) – Whether to zero out the diagonal of the kernel matrix.

Return type `Tensor`

Returns *MMD^2 between the samples from the kernel matrix.*

`alibi_detect.utils.tensorflow.distance.`**`relative_euclidean_distance`**(*x*, *y*, *eps=1e-12*, *axis=-1*)

Relative Euclidean distance.

Parameters

- **`x`** (`Tensor`) – Tensor used in distance computation.
- **`y`** (`Tensor`) – Tensor used in distance computation.
- **`eps`** (`float`) – Epsilon added to denominator for numerical stability.
- **`axis`** (`int`) – Axis used to compute distance.

Return type `Tensor`

Returns *Tensor with relative Euclidean distance across specified axis.*

`alibi_detect.utils.tensorflow.distance.`**`squared_pairwise_distance`**(*x*, *y*, *a_min=1e-30*, *a_max=1e+30*)

TensorFlow pairwise squared Euclidean distance between samples x and y.

Parameters

- **x** (Tensor) – Batch of instances of shape [Nx, features].
- **y** (Tensor) – Batch of instances of shape [Ny, features].
- **a_min** (float) – Lower bound to clip distance values.
- **a_max** (float) – Upper bound to clip distance values.

Return type Tensor

Returns *Pairwise squared Euclidean distance [Nx, Ny].*

alibi_detect.utils.tensorflow.kernels module

class alibi_detect.utils.tensorflow.kernels.**GaussianRBF**(*sigma=None*)

Bases: object

__init__(*sigma=None*)

Gaussian RBF kernel: k(x,y) = exp(-(1/(2*sigma^2)||x-y||^2). A forward pass takes a batch of instances x [Nx, features] and y [Ny, features] and returns the kernel matrix [Nx, Ny].

Parameters **sigma** (Optional[Tensor]) – Optional sigma used for the kernel.

Return type None

alibi_detect.utils.tensorflow.prediction module

alibi_detect.utils.tensorflow.prediction.**predict_batch**(*x, model, batch_size=10000000000, dtype=numpy.float32*)

Make batch predictions on a model.

Parameters

- **x** (Union[ndarray, Tensor]) – Batch of instances.
- **model** (Model) – tf.keras model or one of the other permitted types defined in Data.
- **batch_size** (int) – Batch size used during prediction.
- **dtype** (Union[float32, DType]) – Model output type, e.g. np.float32 or tf.float32.

Return type Union[ndarray, Tensor]

Returns *Numpy array or tensorflow tensor with model outputs.*

alibi_detect.utils.tensorflow.prediction.**predict_batch_transformer**(*x, model, tokenizer, max_len, batch_size=10000000000, dtype=numpy.float32*)

Make batch predictions using a transformers tokenizer and model.

Parameters

- **x** (ndarray) – Batch of instances.
- **model** (Model) – Transformer model.
- **tokenizer** – Tokenizer for model.
- **max_len** (int) – Max token length.

- **batch_size** (int) – Batch size.

Return type Union[ndarray, Tensor]

Returns *Numpy array or tensorflow tensor with model outputs.*

Submodules

alibi_detect.utils.data module

class alibi_detect.utils.data.**Bunch**(***kwargs*)

Bases: dict

Container object for internal datasets Dictionary-like object that exposes its keys as attributes.

alibi_detect.utils.data.**create_outlier_batch**(*data, target, n_samples, perc_outlier*)

Create a batch with a defined percentage of outliers.

Return type Union[*Bunch*, Tuple[ndarray, ndarray]]

alibi_detect.utils.data.**sample_df**(*df, n*)

Sample n instances from the dataframe df.

alibi_detect.utils.discretizer module

class alibi_detect.utils.discretizer.**Discretizer**(*data, categorical_features, feature_names, percentiles=[25, 50, 75]*)

Bases: object

__init__(*data, categorical_features, feature_names, percentiles=[25, 50, 75]*)

Initialize the discretizer.

Parameters

- **data** (ndarray) – Data to discretize
- **categorical_features** (List[int]) – List of indices corresponding to the categorical columns. These features will not be discretized. The other features will be considered continuous and therefore discretized.
- **feature_names** (List[str]) – List with feature names
- **percentiles** (List[int]) – 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.*

alibi_detect.utils.distance module

`alibi_detect.utils.distance.`**`abdm`**(*X*, *cat_vars*, *cat_vars_bin={}*)

Calculate the pair-wise distances between categories of a categorical variable using the Association-Based Distance Metric based on Le et al (2005). http://www.jaist.ac.jp/~bao/papers/N26.pdf

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.

Return type `dict`

Returns *Dict with as keys the categorical columns and as values the pairwise distance matrix for the variable.*

`alibi_detect.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_detect.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 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 Tuple[dict, tuple]

Returns *Dict with multidimensional scaled version of pairwise distance matrices.*

alibi_detect.utils.distance.**mvdm**(*X*, *y*, *cat_vars*, *alpha=1*)
Calculate the pair-wise distances between categories of a categorical variable using the Modified Value Difference Measure based on Cost et al (1993). https://link.springer.com/article/10.1023/A:1022664626993

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 Dict[Any, ndarray]

Returns *Dict with as keys the categorical columns and as values the pairwise distance matrix for the variable.*

alibi_detect.utils.distance.**norm**(*x*, *p*)
Compute p-norm across the features of a batch of instances.

Parameters

- **x** (ndarray) – Batch of instances of shape [N, features].
- **p** (int) – Power of the norm.

Return type ndarray

Returns *Array where p-norm is applied to the features.*

alibi_detect.utils.distance.**pairwise_distance**(*x*, *y*, *p=2*)
Compute pairwise distance between 2 samples.

Parameters

- **x** (ndarray) – Batch of instances of shape [Nx, features].
- **y** (ndarray) – Batch of instances of shape [Ny, features].
- **p** (int) – Power of the norm used to compute the distance.

Return type ndarray

Returns *[Nx, Ny] matrix with pairwise distances.*

alibi_detect.utils.fetching module

alibi_detect.utils.frameworks module

alibi_detect.utils.mapping module

`alibi_detect.utils.mapping.`**`num2ord`**`(`*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_detect.utils.mapping.`**`ohe2ord`**`(`*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_detect.utils.mapping.`**`ohe2ord_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_detect.utils.mapping.`**`ord2num`**`(`*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_detect.utils.mapping.**ord2ohe**(*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_detect.utils.metrics module

alibi_detect.utils.metrics.**accuracy**(*y_true*, *y_pred*)

Return type float

alibi_detect.utils.perturbation module

alibi_detect.utils.perturbation.**apply_mask**(*X, mask_size=(4, 4), n_masks=1, coord=None, channels=[0, 1, 2], mask_type='uniform', noise_distr=(0, 1), noise_rng=(0, 1), clip_rng=(0, 1)*)

Mask images. Can zero out image patches or add normal or uniformly distributed noise.

Parameters

- **X** (ndarray) – Batch of instances to be masked.
- **mask_size** (tuple) – Tuple with the size of the mask.
- **n_masks** (int) – Number of masks applied for each instance in the batch X.
- **coord** (Optional[tuple]) – Upper left (x,y)-coordinates for the mask.
- **channels** (list) – Channels of the image to apply the mask to.
- **mask_type** (str) – Type of mask. One of 'uniform', 'random' (both additive noise) or 'zero' (zero values for mask).
- **noise_distr** (tuple) – Mean and standard deviation for noise of 'random' mask type.
- **noise_rng** (tuple) – Min and max value for noise of 'uniform' type.
- **clip_rng** (tuple) – Min and max values for the masked instances.

Return type Tuple[ndarray, ndarray]

Returns *Tuple with masked instances and the masks.*

alibi_detect.utils.perturbation.**brightness**(*x*, *strength*, *xrange=None*)

Change brightness of image.

Parameters

- **x** (ndarray) – Instance to be perturbed.
- **strength** (float) – Strength of brightness change.

- **xrange** (Optional[tuple]) – Tuple with min and max data range.

Return type ndarray

Returns *Perturbed instance.*

alibi_detect.utils.perturbation.**clipped_zoom**(*x*, *zoom_factor*)
Helper function for zoom blur.

Parameters

- **x** (ndarray) – Instance to be perturbed.
- **zoom_factor** (float) – Zoom strength.

Return type ndarray

Returns *Cropped and zoomed instance.*

alibi_detect.utils.perturbation.**contrast**(*x*, *strength*, *xrange=None*)
Change contrast of image.

Parameters

- **x** (ndarray) – Instance to be perturbed.
- **strength** (float) – Strength of contrast change. Lower is actually more contrast.
- **xrange** (Optional[tuple]) – Tuple with min and max data range.

Return type ndarray

Returns *Perturbed instance.*

alibi_detect.utils.perturbation.**defocus_blur**(*x*, *radius*, *alias_blur*, *xrange=None*)
Apply defocus blur.

Parameters

- **x** (ndarray) – Instance to be perturbed.
- **radius** (int) – Radius for the Gaussian kernel.
- **alias_blur** (float) – Standard deviation for the Gaussian kernel in both X and Y directions.
- **xrange** (Optional[tuple]) – Tuple with min and max data range.

Return type ndarray

Returns *Perturbed instance.*

alibi_detect.utils.perturbation.**disk**(*radius*, *alias_blur=0.1*, *dtype=numpy.float32*)
Helper function for defocus blur.

Parameters

- **radius** (float) – Radius for the Gaussian kernel.
- **alias_blur** (float) – Standard deviation for the Gaussian kernel in both X and Y directions.
- **dtype** – Data type.

Return type ndarray

Returns *Kernel used for Gaussian blurring.*

`alibi_detect.utils.perturbation.`**`elastic_transform`**(*x*, *mult_dxdy*, *sigma*, *rnd_rng*, *xrange=None*)

Apply elastic transformation to instance.

Parameters

- **`x`** (`ndarray`) – Instance to be perturbed.
- **`mult_dxdy`** (`float`) – Multiplier for the Gaussian noise in x and y directions.
- **`sigma`** (`float`) – Standard deviation determining the strength of the Gaussian perturbation.
- **`rnd_rng`** (`float`) – Range for random uniform noise.
- **`xrange`** (`Optional[tuple]`) – Tuple with min and max data range.

Return type `ndarray`

Returns *Perturbed instance.*

`alibi_detect.utils.perturbation.`**`fog`**(*x*, *fractal_mult*, *wibbledecay*, *xrange=None*)

Apply fog to instance.

Parameters

- **`x`** (`ndarray`) – Instance to be perturbed.
- **`fractal_mult`** (`float`) – Strength applied to *plasma_fractal* output.
- **`wibbledecay`** (`float`) – Decay factor for size of noise that is applied.
- **`xrange`** (`Optional[tuple]`) – Tuple with min and max data range.

Return type `ndarray`

Returns *Perturbed instance.*

`alibi_detect.utils.perturbation.`**`gaussian_blur`**(*x*, *sigma*, *multichannel=True*, *xrange=None*)

Apply Gaussian blur.

Parameters

- **`x`** (`ndarray`) – Instance to be perturbed.
- **`sigma`** (`float`) – Standard deviation determining the strength of the blur.
- **`multichannel`** (`bool`) – Whether the image contains multiple channels (RGB) or not.
- **`xrange`** (`Optional[tuple]`) – Tuple with min and max data range.

Return type `ndarray`

Returns *Perturbed instance.*

`alibi_detect.utils.perturbation.`**`gaussian_noise`**(*x*, *stdev*, *xrange=None*)

Inject Gaussian noise.

Parameters

- **`x`** (`ndarray`) – Instance to be perturbed.
- **`stdev`** (`float`) – Standard deviation of noise.
- **`xrange`** (`Optional[tuple]`) – Tuple with min and max data range.

Return type `ndarray`

Returns *Perturbed instance.*

`alibi_detect.utils.perturbation.`**`glass_blur`**(*x*, *sigma*, *max_delta*, *iterations*, *xrange=None*)

Apply glass blur.

Parameters

- **`x`** (`ndarray`) – Instance to be perturbed.
- **`sigma`** (`float`) – Standard deviation determining the strength of the Gaussian perturbation.
- **`max_delta`** (`int`) – Maximum pixel range for the blurring.
- **`iterations`** (`int`) – Number of blurring iterations.
- **`xrange`** (`Optional[tuple]`) – Tuple with min and max data range.

Return type `ndarray`

Returns *Perturbed instance.*

`alibi_detect.utils.perturbation.`**`impulse_noise`**(*x*, *amount*, *xrange=None*)

Inject salt & pepper noise.

Parameters

- **`x`** (`ndarray`) – Instance to be perturbed.
- **`amount`** (`float`) – Proportion of pixels to replace with noise.
- **`xrange`** (`Optional[tuple]`) – Tuple with min and max data range.

Return type `ndarray`

Returns *Perturbed instance.*

`alibi_detect.utils.perturbation.`**`inject_outlier_categorical`**(*X*, *cols*, *perc_outlier*, *y=None*, *cat_perturb=None*, *X_fit=None*, *disc_perc=[25, 50, 75]*, *smooth=1.0*)

Inject outliers in categorical variables of tabular data.

Parameters

- **`X`** (`ndarray`) – Tabular data with categorical variables to perturb (inject outliers).
- **`cols`** (`List[int]`) – Columns of X that are categorical and can be perturbed.
- **`perc_outlier`** (`int`) – Percentage of observations which are perturbed to outliers. For multiple numerical features, the percentage is evenly split across the features.
- **`y`** (`Optional[ndarray]`) – Outlier labels.
- **`cat_perturb`** (`Optional[dict]`) – Dictionary mapping each category in the categorical variables to their furthest neighbour.
- **`X_fit`** (`Optional[ndarray]`) – Optional data used to infer pairwise distances from.
- **`disc_perc`** (`list`) – List with percentiles used in binning of numerical features used for the 'abdm' pairwise distance measure.
- **`smooth`** (`float`) – Smoothing exponent between 0 and 1 for the distances. Lower values will smooth the difference in distance metric between different features.

Return type *`Bunch`*

Returns

- *Bunch object with the perturbed tabular data, outlier labels and*
- *a dictionary used to map categories to their furthest neighbour.*

alibi_detect.utils.perturbation.**inject_outlier_tabular**(*X*, *cols*, *perc_outlier*, *y=None*, *n_std=2.0*, *min_std=1.0*)

Inject outliers in numerical tabular data.

Parameters

- **X** (ndarray) – Tabular data to perturb (inject outliers).
- **cols** (List[int]) – Columns of X that are numerical and can be perturbed.
- **perc_outlier** (int) – Percentage of observations which are perturbed to outliers. For multiple numerical features, the percentage is evenly split across the features.
- **y** (Optional[ndarray]) – Outlier labels.
- **n_std** (float) – Number of feature-wise standard deviations used to perturb the original data.
- **min_std** (float) – Minimum number of standard deviations away from the current observation. This is included because of the stochastic nature of the perturbation which could lead to minimal perturbations without a floor.

Return type *Bunch*

Returns *Bunch object with the perturbed tabular data and the outlier labels.*

alibi_detect.utils.perturbation.**inject_outlier_ts**(*X*, *perc_outlier*, *perc_window=10*, *n_std=2.0*, *min_std=1.0*)

Inject outliers in both univariate and multivariate time series data.

Parameters

- **X** (ndarray) – Time series data to perturb (inject outliers).
- **perc_outlier** (int) – Percentage of observations which are perturbed to outliers. For multivariate data, the percentage is evenly split across the individual time series.
- **perc_window** (int) – Percentage of the observations used to compute the standard deviation used in the perturbation.
- **n_std** (float) – Number of standard deviations in the window used to perturb the original data.
- **min_std** (float) – Minimum number of standard deviations away from the current observation. This is included because of the stochastic nature of the perturbation which could lead to minimal perturbations without a floor.

Return type *Bunch*

Returns *Bunch object with the perturbed time series and the outlier labels.*

alibi_detect.utils.perturbation.**jpeg_compression**(*x*, *strength*, *xrange=None*)

Simulate changes due to JPEG compression for an image.

Parameters

- **x** (ndarray) – Instance to be perturbed.
- **strength** (float) – Strength of compression (>1). Lower is actually more compressed.

- **xrange** (Optional[tuple]) – Tuple with min and max data range.

Return type ndarray

Returns *Perturbed instance.*

alibi_detect.utils.perturbation.**mutate_categorical**(*X*, *rate=None*, *seed=0*, *feature_range=(0, 255)*)

Randomly change integer feature values to values within a set range with a specified permutation rate.

Parameters

- **X** (ndarray) – Batch of data to be perturbed.
- **rate** (Optional[float]) – Permutation rate (between 0 and 1).
- **seed** (int) – Random seed.
- **feature_range** (tuple) – Min and max range for perturbed features.

Return type Tensor

Returns *Array with perturbed data.*

alibi_detect.utils.perturbation.**pixelate**(*x*, *strength*, *xrange=None*)

Change coarseness of pixels for an image.

Parameters

- **x** (ndarray) – Instance to be perturbed.
- **strength** (float) – Strength of pixelation (<1). Lower is actually more pixelated.
- **xrange** (Optional[tuple]) – Tuple with min and max data range.

Return type ndarray

Returns *Perturbed instance.*

alibi_detect.utils.perturbation.**plasma_fractal**(*mapsize=256*, *wibbledecay=3.0*)

Helper function to apply fog to instance. Generates a heightmap using diamond-square algorithm. Returns a square 2d array, side length 'mapsize', of floats in range 0-255. 'mapsize' must be a power of two.

Return type ndarray

alibi_detect.utils.perturbation.**saturate**(*x*, *strength*, *xrange=None*)

Change colour saturation of image.

Parameters

- **x** (ndarray) – Instance to be perturbed.
- **strength** (tuple) – Strength of saturation change. Tuple consists of (multiplier, shift) of the perturbation.
- **xrange** (Optional[tuple]) – Tuple with min and max data range.

Return type ndarray

Returns *Perturbed instance.*

alibi_detect.utils.perturbation.**scale_minmax**(*x*, *xrange=None*)

Minmax scaling to [0,1].

Parameters

- **x** (ndarray) – Numpy array to be scaled.
- **xrange** (Optional[tuple]) – Tuple with min and max data range.

Return type `Tuple[ndarray, bool]`

Returns *Scaled array and boolean whether the array is actually scaled.*

`alibi_detect.utils.perturbation.`**`shot_noise`**(*x*, *lam*, *xrange=None*)

Inject Poisson noise.

Parameters

- **`x`** (`ndarray`) – Instance to be perturbed.
- **`lam`** (`float`) – Scalar for the lambda parameter determining the expectation of the interval.
- **`xrange`** (`Optional[tuple]`) – Tuple with min and max data range.

Return type `ndarray`

Returns *Perturbed instance.*

`alibi_detect.utils.perturbation.`**`speckle_noise`**(*x*, *stdev*, *xrange=None*)

Inject speckle noise.

Parameters

- **`x`** (`ndarray`) – Instance to be perturbed.
- **`stdev`** (`float`) – Standard deviation of noise.
- **`xrange`** (`Optional[tuple]`) – Tuple with min and max data range.

Return type `ndarray`

Returns *Perturbed instance.*

`alibi_detect.utils.perturbation.`**`zoom_blur`**(*x*, *max_zoom*, *step_zoom*, *xrange=None*)

Apply zoom blur.

Parameters

- **`x`** (`ndarray`) – Instance to be perturbed.
- **`max_zoom`** (`float`) – Max zoom strength.
- **`step_zoom`** (`float`) – Step size to go from 1 to *max_zoom* strength.
- **`xrange`** (`Optional[tuple]`) – Tuple with min and max data range.

Return type `ndarray`

Returns *Perturbed instance.*

alibi_detect.utils.prediction module

`alibi_detect.utils.prediction.`**`predict_batch`**(*model*, *X*, *batch_size=10000000000*, *proba=False*, *return_class=False*, *n_categories=None*, *shape=None*, *dtype=numpy.float32*)

Make batch predictions on a model.

Parameters

- **`model`** (`Union[Model, Callable, AE, AEGMM, Seq2Seq, VAE, VAEGMM]`) – tf.keras model or one of the other permitted types defined in Data.
- **`X`** (`ndarray`) – Batch of instances.

- **batch_size** (int) – Batch size used during prediction.
- **proba** (bool) – Whether to return model prediction probabilities.
- **return_class** (bool) – Whether to return model class predictions.
- **n_categories** (Optional[int]) – Number of prediction categories. Can also be inferred from the model.
- **shape** (Optional[tuple]) – Optional shape or tuple with shapes of the model predictions.
- **dtype** (type) – Output type.

Return type Union[ndarray, tuple]

Returns *Numpy array with predictions.*

alibi_detect.utils.prediction.**predict_batch_transformer**(*model, tokenizer, X, max_len, batch_size=10000000000*)

Parameters

- **model** (Model) – HuggingFace transformer model.
- **tokenizer** – Tokenizer for model.
- **X** (ndarray) – Batch of instances.
- **max_len** (int) – Max token length.
- **batch_size** (int) – Batch size.

Return type ndarray

Returns *Numpy array with predictions.*

alibi_detect.utils.sampling module

alibi_detect.utils.sampling.**reservoir_sampling**(*X_ref, X, reservoir_size, n*)

Apply reservoir sampling.

Parameters

- **X_ref** (ndarray) – Current instances in reservoir.
- **X** (ndarray) – Data to update reservoir with.
- **reservoir_size** (int) – Size of reservoir.
- **n** (int) – Number of total instances that have passed so far.

Return type ndarray

Returns *Updated reservoir.*

alibi_detect.utils.saving module

alibi_detect.utils.visualize module

`alibi_detect.utils.visualize.`**`plot_feature_outlier_image`**`(`*od_preds, X, X_recon=None, instance_ids=None, max_instances=5, outliers_only=False, n_channels=3, figsize=(20, 20)*`)`

Plot feature (pixel) wise outlier scores for images.

Parameters

- **`od_preds`** (`Dict`) – Output of an outlier detector's prediction.
- **`X`** (`ndarray`) – Batch of instances to apply outlier detection to.
- **`X_recon`** (`Optional[ndarray]`) – Reconstructed instances of X.
- **`instance_ids`** (`Optional[list]`) – List with indices of instances to display.
- **`max_instances`** (`int`) – Maximum number of instances to display.
- **`outliers_only`** (`bool`) – Whether to only show outliers or not.
- **`n_channels`** (`int`) – Number of channels of the images.
- **`figsize`** (`tuple`) – Tuple for the figure size.

Return type `None`

`alibi_detect.utils.visualize.`**`plot_feature_outlier_tabular`**`(`*od_preds, X, X_recon=None, threshold=None, instance_ids=None, max_instances=5, top_n=1000000000000, outliers_only=False, feature_names=None, width=0.2, figsize=(20, 10)*`)`

Plot feature wise outlier scores for tabular data.

Parameters

- **`od_preds`** (`Dict`) – Output of an outlier detector's prediction.
- **`X`** (`ndarray`) – Batch of instances to apply outlier detection to.
- **`X_recon`** (`Optional[ndarray]`) – Reconstructed instances of X.
- **`threshold`** (`Optional[float]`) – Threshold used for outlier score to determine outliers.
- **`instance_ids`** (`Optional[list]`) – List with indices of instances to display.
- **`max_instances`** (`int`) – Maximum number of instances to display.
- **`top_n`** (`int`) – Maixmum number of features to display, ordered by outlier score.
- **`outliers_only`** (`bool`) – Whether to only show outliers or not.

- **feature_names** (Optional[list]) – List with feature names.
- **width** (float) – Column width for bar charts.
- **figsize** (tuple) – Tuple for the figure size.

Return type None

alibi_detect.utils.visualize.**plot_feature_outlier_ts**(*od_preds*, *X*, *threshold*, *window=None*, *t=None*, *X_orig=None*, *width=0.2*, *figsize=(20, 8)*, *ylim=(None, None)*)

Plot feature wise outlier scores for time series data.

Parameters

- **od_preds** (Dict) – Output of an outlier detector's prediction.
- **X** (ndarray) – Time series to apply outlier detection to.
- **threshold** (Union[float, int, list, ndarray]) – Threshold used to classify outliers or adversarial instances.
- **window** (Optional[tuple]) – Start and end timestep to plot.
- **t** (Optional[ndarray]) – Timesteps.
- **X_orig** (Optional[ndarray]) – Optional original time series without outliers.
- **width** (float) – Column width for bar charts.
- **figsize** (tuple) – Tuple for the figure size.
- **ylim** (tuple) – Min and max y-axis values for the outlier scores.

Return type None

alibi_detect.utils.visualize.**plot_instance_score**(*preds*, *target*, *labels*, *threshold*, *ylim=(None, None)*)

Scatter plot of a batch of outlier or adversarial scores compared to the threshold.

Parameters

- **preds** (Dict) – Dictionary returned by predictions of an outlier or adversarial detector.
- **target** (ndarray) – Ground truth.
- **labels** (ndarray) – List with names of classification labels.
- **threshold** (float) – Threshold used to classify outliers or adversarial instances.
- **ylim** (tuple) – Min and max y-axis values.

Return type None

alibi_detect.utils.visualize.**plot_roc**(*roc_data*, *figsize=(10, 5)*)

Plot ROC curve.

Parameters

- **roc_data** (Dict[str, Dict[str, ndarray]]) – Dictionary with as key the label to show in the legend and as value another dictionary with as keys *scores* and *labels* with respectively the outlier scores and outlier labels.
- **figsize** (tuple) – Figure size.

Return type None

47.1.2 Submodules

alibi_detect.base module

class alibi_detect.base.**BaseDetector**

Bases: abc.ABC

Base class for outlier detection algorithms.

property meta

Return type Dict

abstract predict(*X*)

abstract score(*X*)

class alibi_detect.base.**FitMixin**

Bases: abc.ABC

abstract fit(*X*)

Return type None

class alibi_detect.base.**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(*obj*)

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:

```
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)
```

class alibi_detect.base.**ThresholdMixin**

Bases: abc.ABC

abstract infer_threshold(*X*)

Return type None

alibi_detect.base.**adversarial_correction_dict**()

alibi_detect.base.**adversarial_prediction_dict**()

alibi_detect.base.**concept_drift_dict**()

alibi_detect.base.**outlier_prediction_dict**()

alibi_detect.datasets module

alibi_detect.version module

CHAPTER

FORTYEIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

alibi_detect.ad, 345
alibi_detect.ad.adversarialae, 349
alibi_detect.ad.model_distillation, 352
alibi_detect.base, 403
alibi_detect.cd.base, 356
alibi_detect.cd.chisquare, 361
alibi_detect.cd.pytorch, 353
alibi_detect.cd.pytorch.mmd, 354
alibi_detect.cd.pytorch.preprocess, 355
alibi_detect.cd.pytorch.utils, 356
alibi_detect.cd.utils, 362
alibi_detect.models, 363
alibi_detect.models.pytorch.embedding, 363
alibi_detect.models.tensorflow, 364
alibi_detect.models.tensorflow.autoencoder, 369
alibi_detect.models.tensorflow.embedding, 373
alibi_detect.models.tensorflow.gmm, 374
alibi_detect.models.tensorflow.losses, 375
alibi_detect.models.tensorflow.pixelcnn, 377
alibi_detect.models.tensorflow.resnet, 377
alibi_detect.models.tensorflow.trainer, 380
alibi_detect.utils, 381
alibi_detect.utils.data, 389
alibi_detect.utils.discretizer, 389
alibi_detect.utils.distance, 390
alibi_detect.utils.frameworks, 392
alibi_detect.utils.mapping, 392
alibi_detect.utils.metrics, 393
alibi_detect.utils.perturbation, 393
alibi_detect.utils.prediction, 399
alibi_detect.utils.pytorch, 381
alibi_detect.utils.pytorch.distance, 383
alibi_detect.utils.pytorch.kernels, 384
alibi_detect.utils.pytorch.prediction, 384
alibi_detect.utils.sampling, 400
alibi_detect.utils.tensorflow, 385
alibi_detect.utils.tensorflow.distance, 387
alibi_detect.utils.tensorflow.kernels, 388
alibi_detect.utils.tensorflow.prediction, 388
alibi_detect.utils.visualize, 401

INDEX

Symbols

`__init__()` (*alibi_detect.ad.AdversarialAE method*), 345

`__init__()` (*alibi_detect.ad.ModelDistillation method*), 347

`__init__()` (*alibi_detect.ad.adversarialae.AdversarialAE method*), 349

`__init__()` (*alibi_detect.ad.adversarialae.DenseHidden method*), 351

`__init__()` (*alibi_detect.ad.model_distillation.ModelDistillation method*), 352

`__init__()` (*alibi_detect.cd.base.BaseClassifierDrift method*), 356

`__init__()` (*alibi_detect.cd.base.BaseMMDDrift method*), 358

`__init__()` (*alibi_detect.cd.base.BaseUnivariateDrift method*), 359

`__init__()` (*alibi_detect.cd.chisquare.ChiSquareDrift method*), 361

`__init__()` (*alibi_detect.cd.pytorch.mmd.MMDDriftTorch method*), 354

`__init__()` (*alibi_detect.models.tensorflow.AE method*), 364

`__init__()` (*alibi_detect.models.tensorflow.AEGMM method*), 364

`__init__()` (*alibi_detect.models.tensorflow.PixelCNN method*), 366

`__init__()` (*alibi_detect.models.tensorflow.Seq2Seq method*), 364

`__init__()` (*alibi_detect.models.tensorflow.TransformerEmbedding method*), 367

`__init__()` (*alibi_detect.models.tensorflow.VAE method*), 365

`__init__()` (*alibi_detect.models.tensorflow.VAEGMM method*), 365

`__init__()` (*alibi_detect.models.tensorflow.autoencoder.AE method*), 369

`__init__()` (*alibi_detect.models.tensorflow.autoencoder.AEGMM method*), 369

`__init__()` (*alibi_detect.models.tensorflow.autoencoder.Decoder method*), 369

`__init__()` (*alibi_detect.models.tensorflow.autoencoder.DecoderLSTM method*), 370

`__init__()` (*alibi_detect.models.tensorflow.autoencoder.EncoderAE method*), 370

`__init__()` (*alibi_detect.models.tensorflow.autoencoder.EncoderLSTM method*), 370

`__init__()` (*alibi_detect.models.tensorflow.autoencoder.EncoderVAE method*), 371

`__init__()` (*alibi_detect.models.tensorflow.autoencoder.Seq2Seq method*), 371

`__init__()` (*alibi_detect.models.tensorflow.autoencoder.VAE method*), 372

`__init__()` (*alibi_detect.models.tensorflow.autoencoder.VAEGMM method*), 372

`__init__()` (*alibi_detect.models.tensorflow.embedding.TransformerEmb method*), 373

`__init__()` (*alibi_detect.models.tensorflow.pixelcnn.Shift method*), 377

`__init__()` (*alibi_detect.models.tensorflow.resnet.LearningRateBatchSc method*), 377

`__init__()` (*alibi_detect.utils.discretizer.Discretizer method*), 389

`__init__()` (*alibi_detect.utils.pytorch.GaussianRBF method*), 382

`__init__()` (*alibi_detect.utils.pytorch.kernels.GaussianRBF method*), 384

`__init__()` (*alibi_detect.utils.tensorflow.GaussianRBF method*), 386

`__init__()` (*alibi_detect.utils.tensorflow.kernels.GaussianRBF method*), 388

A

`abdm()` (*in module alibi_detect.utils.distance*), 390

`accuracy()` (*in module alibi_detect.utils.metrics*), 393

`activate_train_mode_for_dropout_layers()` (*in module alibi_detect.cd.pytorch.utils*), 356

`adversarial_correction_dict()` (*in module alibi_detect.base*), 403

`adversarial_prediction_dict()` (*in module alibi_detect.base*), 403

`AdversarialAE` (*class in alibi_detect.ad*), 345

`AdversarialAE` (*class in alibi_detect.ad.adversarialae*), 349

AE (*class in alibi_detect.models.tensorflow*), 364
AE (*class in alibi_detect.models.tensorflow.autoencoder*), 369
AEGMM (*class in alibi_detect.models.tensorflow*), 364
AEGMM (*class in alibi_detect.models.tensorflow.autoencoder*), 369
alibi_detect.ad (*module*), 345
alibi_detect.ad.adversarialae (*module*), 349
alibi_detect.ad.model_distillation (*module*), 352
alibi_detect.base (*module*), 403
alibi_detect.cd.base (*module*), 356
alibi_detect.cd.chisquare (*module*), 361
alibi_detect.cd.pytorch (*module*), 353
alibi_detect.cd.pytorch.mmd (*module*), 354
alibi_detect.cd.pytorch.preprocess (*module*), 355
alibi_detect.cd.pytorch.utils (*module*), 356
alibi_detect.cd.utils (*module*), 362
alibi_detect.models (*module*), 363
alibi_detect.models.pytorch.embedding (*module*), 363
alibi_detect.models.tensorflow (*module*), 364
alibi_detect.models.tensorflow.autoencoder (*module*), 369
alibi_detect.models.tensorflow.embedding (*module*), 373
alibi_detect.models.tensorflow.gmm (*module*), 374
alibi_detect.models.tensorflow.losses (*module*), 375
alibi_detect.models.tensorflow.pixelcnn (*module*), 377
alibi_detect.models.tensorflow.resnet (*module*), 377
alibi_detect.models.tensorflow.trainer (*module*), 380
alibi_detect.utils (*module*), 381
alibi_detect.utils.data (*module*), 389
alibi_detect.utils.discretizer (*module*), 389
alibi_detect.utils.distance (*module*), 390
alibi_detect.utils.frameworks (*module*), 392
alibi_detect.utils.mapping (*module*), 392
alibi_detect.utils.metrics (*module*), 393
alibi_detect.utils.perturbation (*module*), 393
alibi_detect.utils.prediction (*module*), 399
alibi_detect.utils.pytorch (*module*), 381
alibi_detect.utils.pytorch.distance (*module*), 383
alibi_detect.utils.pytorch.kernels (*module*), 384
alibi_detect.utils.pytorch.prediction (*module*), 384
alibi_detect.utils.sampling (*module*), 400
alibi_detect.utils.tensorflow (*module*), 385
alibi_detect.utils.tensorflow.distance (*module*), 387
alibi_detect.utils.tensorflow.kernels (*module*), 388
alibi_detect.utils.tensorflow.prediction (*module*), 388
alibi_detect.utils.visualize (*module*), 401
apply_mask() (*in module alibi_detect.utils.perturbation*), 393

B

BaseClassifierDrift (*class in alibi_detect.cd.base*), 356
BaseDetector (*class in alibi_detect.base*), 403
BaseMMDDrift (*class in alibi_detect.cd.base*), 358
BaseUnivariateDrift (*class in alibi_detect.cd.base*), 359
bins() (*alibi_detect.utils.discretizer.Discretizer method*), 389
brightness() (*in module alibi_detect.utils.perturbation*), 393
Bunch (*class in alibi_detect.utils.data*), 389

C

call() (*alibi_detect.ad.adversarialae.DenseHidden method*), 351
call() (*alibi_detect.models.tensorflow.AE method*), 364
call() (*alibi_detect.models.tensorflow.AEGMM method*), 364
call() (*alibi_detect.models.tensorflow.autoencoder.AE method*), 369
call() (*alibi_detect.models.tensorflow.autoencoder.AEGMM method*), 369
call() (*alibi_detect.models.tensorflow.autoencoder.Decoder method*), 370
call() (*alibi_detect.models.tensorflow.autoencoder.DecoderLSTM method*), 370
call() (*alibi_detect.models.tensorflow.autoencoder.EncoderAE method*), 370
call() (*alibi_detect.models.tensorflow.autoencoder.EncoderLSTM method*), 371
call() (*alibi_detect.models.tensorflow.autoencoder.EncoderVAE method*), 371

call() (*alibi_detect.models.tensorflow.autoencoder.Sampling method*), 371
call() (*alibi_detect.models.tensorflow.autoencoder.Seq2Seq method*), 372
call() (*alibi_detect.models.tensorflow.autoencoder.VAE method*), 372
call() (*alibi_detect.models.tensorflow.autoencoder.VAEGMM method*), 373
call() (*alibi_detect.models.tensorflow.embedding.TransformerEmbedding method*), 374
call() (*alibi_detect.models.tensorflow.Seq2Seq method*), 365
call() (*alibi_detect.models.tensorflow.TransformerEmbedding method*), 368
call() (*alibi_detect.models.tensorflow.VAE method*), 365
call() (*alibi_detect.models.tensorflow.VAEGMM method*), 366
ChiSquareDrift (*class in alibi_detect.cd.chisquare*), 361
cityblock_batch() (*in module alibi_detect.utils.distance*), 390
clipped_zoom() (*in module alibi_detect.utils.perturbation*), 394
concept_drift_dict() (*in module alibi_detect.base*), 403
contrast() (*in module alibi_detect.utils.perturbation*), 394
conv_block() (*in module alibi_detect.models.tensorflow.resnet*), 378
correct() (*alibi_detect.ad.AdversarialAE method*), 346
correct() (*alibi_detect.ad.adversarialae.AdversarialAE method*), 350
create_outlier_batch() (*in module alibi_detect.utils.data*), 389

D

decode_seq() (*alibi_detect.models.tensorflow.autoencoder.Seq2Seq method*), 372
decode_seq() (*alibi_detect.models.tensorflow.Seq2Seq method*), 365
Decoder (*class in alibi_detect.models.tensorflow.autoencoder*), 369
DecoderLSTM (*class in alibi_detect.models.tensorflow.autoencoder*), 370
default() (*alibi_detect.base.NumpyEncoder method*), 403
defocus_blur() (*in module alibi_detect.utils.perturbation*), 394
DenseHidden (*class in alibi_detect.ad.adversarialae*), 351
discretize() (*alibi_detect.utils.discretizer.Discretizer method*), 389
Discretizer (*class in alibi_detect.utils.discretizer*), 389
disk() (*in module alibi_detect.utils.perturbation*), 394

E

elastic_transform() (*in module alibi_detect.utils.perturbation*), 394
elbo() (*in module alibi_detect.models.tensorflow.losses*), 375
EncoderAE (*class in alibi_detect.models.tensorflow.autoencoder*), 370
EncoderLSTM (*class in alibi_detect.models.tensorflow.autoencoder*), 370
EncoderVAE (*class in alibi_detect.models.tensorflow.autoencoder*), 371
encompass_batching() (*in module alibi_detect.cd.utils*), 362
encompass_shuffling_and_batch_filling() (*in module alibi_detect.cd.utils*), 362
eucl_cosim_features() (*in module alibi_detect.models.tensorflow.autoencoder*), 373

F

feature_score() (*alibi_detect.cd.base.BaseUnivariateDrift method*), 360
feature_score() (*alibi_detect.cd.chisquare.ChiSquareDrift method*), 362
fit() (*alibi_detect.ad.AdversarialAE method*), 346
fit() (*alibi_detect.ad.adversarialae.AdversarialAE method*), 350
fit() (*alibi_detect.ad.model_distillation.ModelDistillation method*), 352
fit() (*alibi_detect.ad.ModelDistillation method*), 348
fit() (*alibi_detect.base.FitMixin method*), 403
FitMixin (*class in alibi_detect.base*), 403
fog() (*in module alibi_detect.utils.perturbation*), 395
forward() (*alibi_detect.cd.pytorch.HiddenOutput method*), 353
forward() (*alibi_detect.cd.pytorch.preprocess.HiddenOutput method*), 355
forward() (*alibi_detect.models.pytorch.embedding.TransformerEmbedd method*), 363

G

gaussian_blur() (*in module alibi_detect.utils.perturbation*), 395

gaussian_noise() (*in module alibi_detect.utils.perturbation*), 395
GaussianRBF (*class in alibi_detect.utils.pytorch*), 382
GaussianRBF (*class in alibi_detect.utils.pytorch.kernels*), 384
GaussianRBF (*class in alibi_detect.utils.tensorflow*), 386
GaussianRBF (*class in alibi_detect.utils.tensorflow.kernels*), 388
get_splits() (*alibi_detect.cd.base.BaseClassifierDrift method*), 357
glass_blur() (*in module alibi_detect.utils.perturbation*), 395
gmm_energy() (*in module alibi_detect.models.tensorflow.gmm*), 374
gmm_params() (*in module alibi_detect.models.tensorflow.gmm*), 375

H

hidden_state_embedding() (*in module alibi_detect.models.pytorch.embedding*), 363
hidden_state_embedding() (*in module alibi_detect.models.tensorflow.embedding*), 374
HiddenOutput (*class in alibi_detect.cd.pytorch*), 353
HiddenOutput (*class in alibi_detect.cd.pytorch.preprocess*), 355

I

identity_block() (*in module alibi_detect.models.tensorflow.resnet*), 378
impulse_noise() (*in module alibi_detect.utils.perturbation*), 396
infer_threshold() (*alibi_detect.ad.AdversarialAE method*), 346
infer_threshold() (*alibi_detect.ad.adversarialae.AdversarialAE method*), 350
infer_threshold() (*alibi_detect.ad.model_distillation.ModelDistillation method*), 352
infer_threshold() (*alibi_detect.ad.ModelDistillation method*), 348
infer_threshold() (*alibi_detect.base.ThresholdMixin method*), 403
inject_outlier_categorical() (*in module alibi_detect.utils.perturbation*), 396
inject_outlier_tabular() (*in module alibi_detect.utils.perturbation*), 397
inject_outlier_ts() (*in module alibi_detect.utils.perturbation*), 397

J

jpeg_compression() (*in module alibi_detect.utils.perturbation*), 397

K

kernel_matrix() (*alibi_detect.cd.base.BaseMMDDrift method*), 359
kernel_matrix() (*alibi_detect.cd.pytorch.mmd.MMDDriftTorch method*), 355

L

l2_regulariser() (*in module alibi_detect.models.tensorflow.resnet*), 378
learning_rate_schedule() (*in module alibi_detect.models.tensorflow.resnet*), 378
LearningRateBatchScheduler (*class in alibi_detect.models.tensorflow.resnet*), 377
loss_adv_ae() (*in module alibi_detect.models.tensorflow.losses*), 375
loss_aegmm() (*in module alibi_detect.models.tensorflow.losses*), 376
loss_distillation() (*in module alibi_detect.models.tensorflow.losses*), 376
loss_vaegmm() (*in module alibi_detect.models.tensorflow.losses*), 376

M

meta() (*alibi_detect.base.BaseDetector property*), 403
mmd2() (*in module alibi_detect.utils.pytorch*), 381
mmd2() (*in module alibi_detect.utils.pytorch.distance*), 383
mmd2() (*in module alibi_detect.utils.tensorflow*), 385
mmd2() (*in module alibi_detect.utils.tensorflow.distance*), 387
mmd2_from_kernel_matrix() (*in module alibi_detect.utils.pytorch*), 381
mmd2_from_kernel_matrix() (*in module alibi_detect.utils.pytorch.distance*), 383
mmd2_from_kernel_matrix() (*in module alibi_detect.utils.tensorflow*), 385
mmd2_from_kernel_matrix() (*in module alibi_detect.utils.tensorflow.distance*), 387
MMDDriftTorch (*class in alibi_detect.cd.pytorch.mmd*), 354
ModelDistillation (*class in alibi_detect.ad*), 347
ModelDistillation (*class in alibi_detect.ad.model_distillation*), 352
multidim_scaling() (*in module alibi_detect.utils.distance*), 390
mutate_categorical() (*in module alibi_detect.utils.perturbation*), 398

mvdm() (*in module alibi_detect.utils.distance*), 391

N

norm() (*in module alibi_detect.utils.distance*), 391
num2ord() (*in module alibi_detect.utils.mapping*), 392
NumpyEncoder (*class in alibi_detect.base*), 403

O

ohe2ord() (*in module alibi_detect.utils.mapping*), 392
ohe2ord_shape() (*in module alibi_detect.utils.mapping*), 392
on_batch_begin() (*alibi_detect.models.tensorflow.resnet.LearningRateBatchScheduler method*), 378
on_epoch_begin() (*alibi_detect.models.tensorflow.resnet.LearningRateBatchScheduler method*), 378
ord2num() (*in module alibi_detect.utils.mapping*), 392
ord2ohe() (*in module alibi_detect.utils.mapping*), 392
outlier_prediction_dict() (*in module alibi_detect.base*), 403

P

pairwise_distance() (*in module alibi_detect.utils.distance*), 391
pixelate() (*in module alibi_detect.utils.perturbation*), 398
PixelCNN (*class in alibi_detect.models.tensorflow*), 366
plasma_fractal() (*in module alibi_detect.utils.perturbation*), 398
plot_feature_outlier_image() (*in module alibi_detect.utils.visualize*), 401
plot_feature_outlier_tabular() (*in module alibi_detect.utils.visualize*), 401
plot_feature_outlier_ts() (*in module alibi_detect.utils.visualize*), 402
plot_instance_score() (*in module alibi_detect.utils.visualize*), 402
plot_roc() (*in module alibi_detect.utils.visualize*), 402
predict() (*alibi_detect.ad.AdversarialAE method*), 347
predict() (*alibi_detect.ad.adversarialae.AdversarialAE method*), 351
predict() (*alibi_detect.ad.model_distillation.ModelDistillation method*), 353
predict() (*alibi_detect.ad.ModelDistillation method*), 348
predict() (*alibi_detect.base.BaseDetector method*), 403
predict() (*alibi_detect.cd.base.BaseClassifierDrift method*), 357
predict() (*alibi_detect.cd.base.BaseMMDDrift method*), 359
predict() (*alibi_detect.cd.base.BaseUnivariateDrift method*), 360
predict_batch() (*in module alibi_detect.utils.prediction*), 399
predict_batch() (*in module alibi_detect.utils.pytorch*), 382
predict_batch() (*in module alibi_detect.utils.pytorch.prediction*), 384
predict_batch() (*in module alibi_detect.utils.tensorflow*), 386
predict_batch() (*in module alibi_detect.utils.tensorflow.prediction*), 388
predict_batch_transformer() (*in module alibi_detect.utils.prediction*), 400
predict_batch_transformer() (*in module alibi_detect.utils.pytorch*), 382
predict_batch_transformer() (*in module alibi_detect.utils.pytorch.prediction*), 384
predict_batch_transformer() (*in module alibi_detect.utils.tensorflow*), 386
predict_batch_transformer() (*in module alibi_detect.utils.tensorflow.prediction*), 388
preprocess() (*alibi_detect.cd.base.BaseClassifierDrift method*), 358
preprocess() (*alibi_detect.cd.base.BaseMMDDrift method*), 359
preprocess() (*alibi_detect.cd.base.BaseUnivariateDrift method*), 361
preprocess_drift() (*in module alibi_detect.cd.pytorch*), 353
preprocess_drift() (*in module alibi_detect.cd.pytorch.preprocess*), 355
preprocess_image() (*in module alibi_detect.models.tensorflow.resnet*), 379

R

relative_euclidean_distance() (*in module alibi_detect.utils.tensorflow*), 385
relative_euclidean_distance() (*in module alibi_detect.utils.tensorflow.distance*), 387
reservoir_sampling() (*in module alibi_detect.utils.sampling*), 400
resnet() (*in module alibi_detect.models.tensorflow*), 366
resnet() (*in module alibi_detect.models.tensorflow.resnet*), 379
resnet_block() (*in module alibi_detect.models.tensorflow.resnet*), 379
run() (*in module alibi_detect.models.tensorflow.resnet*), 379

S

sample_df() (*in module alibi_detect.utils.data*), 389

Sampling (*class in alibi_detect.models.tensorflow.autoencoder*), 371
saturate() (*in module alibi_detect.utils.perturbation*), 398
scale_by_instance() (*in module alibi_detect.models.tensorflow.resnet*), 380
scale_minmax() (*in module alibi_detect.utils.perturbation*), 398
score() (*alibi_detect.ad.AdversarialAE method*), 347
score() (*alibi_detect.ad.adversarialae.AdversarialAE method*), 351
score() (*alibi_detect.ad.model_distillation.ModelDistillation method*), 353
score() (*alibi_detect.ad.ModelDistillation method*), 348
score() (*alibi_detect.base.BaseDetector method*), 403
score() (*alibi_detect.cd.base.BaseClassifierDrift method*), 358
score() (*alibi_detect.cd.base.BaseMMDDrift method*), 359
score() (*alibi_detect.cd.base.BaseUnivariateDrift method*), 361
score() (*alibi_detect.cd.pytorch.mmd.MMDDriftTorch method*), 355
Seq2Seq (*class in alibi_detect.models.tensorflow*), 364
Seq2Seq (*class in alibi_detect.models.tensorflow.autoencoder*), 371
Shift (*class in alibi_detect.models.tensorflow.pixelcnn*), 377
shift() (*alibi_detect.models.tensorflow.pixelcnn.Shift property*), 377
shot_noise() (*in module alibi_detect.utils.perturbation*), 399
speckle_noise() (*in module alibi_detect.utils.perturbation*), 399
squared_pairwise_distance() (*in module alibi_detect.utils.pytorch*), 381
squared_pairwise_distance() (*in module alibi_detect.utils.pytorch.distance*), 383
squared_pairwise_distance() (*in module alibi_detect.utils.tensorflow*), 385
squared_pairwise_distance() (*in module alibi_detect.utils.tensorflow.distance*), 387

T

test_probs() (*alibi_detect.cd.base.BaseClassifierDrift method*), 358
ThresholdMixin (*class in alibi_detect.base*), 403
trainer() (*in module alibi_detect.models.tensorflow*), 368
trainer() (*in module alibi_detect.models.tensorflow.trainer*), 380
TransformerEmbedding (*class in alibi_detect.models.pytorch.embedding*), 363
TransformerEmbedding (*class in alibi_detect.models.tensorflow*), 367
TransformerEmbedding (*class in alibi_detect.models.tensorflow.embedding*), 373

U

update_reference() (*in module alibi_detect.cd.utils*), 362

V

VAE (*class in alibi_detect.models.tensorflow*), 365
VAE (*class in alibi_detect.models.tensorflow.autoencoder*), 372
VAEGMM (*class in alibi_detect.models.tensorflow*), 365
VAEGMM (*class in alibi_detect.models.tensorflow.autoencoder*), 372

Z

zoom_blur() (*in module alibi_detect.utils.perturbation*), 399