CONTENTS

CHAPTER

ONE

VIBRODIAGNOSTICS PACKAGE

1.1 extraction module

`extraction.detrending_filter(`*dataframes*, *columns*`)`

Subtract average from each value in columns of multiple data frames

Parameters

- `dataframes` (*List [DataFrame]*) – list of data frames
- `columns` (*List [str]*) – attributes from which mean is removed

Returns

modified data frames with mean removed

Return type

List[*DataFrame*]

`extraction.downsample(`*x*, *k*, *fs_reduced*, *fs*`)`

Downsample the time series by a factor

Parameters

- `x` (*array*) – time series
- `k` (*int*) – factor for downsampling (when it is None, the factor is calculated as a ratio of sampling rates)
- `fs_reduced` (*int*) – desired sampling frequency
- `fs` (*int*) – original sampling frequency of the time series

Returns

downsampled time series

Return type

array

`extraction.energy(`*x*`)`

Calculate energy of the signal

Parameters

`x` (*array*) – input signal

Returns

energy of the signal

Return type

float

`extraction.envelope_signal(`*f*, *pxx*`)`

Approximate envelope of frequency spectrum with quadratic interpolation between peaks

Parameters

- `f` (*array*) – array of frequency bins
- `pxx` (*array*) – array of amplitudes at frequency bins

Returns
envelope of the amplitudes

Return type
array

`extraction.frequency_features_calc(`*df*, *col*, *fs*, *window*`)`

Extract complete set of features in frequency-domain (FD set)

Parameters

- `df` (*DataFrame*) – data frame with input time series in columns
- `col` (*str*) – column in data frame to use for extraction
- `fs` (*int*) – sampling frequency in Hz of the time series
- `window` (*int*) – length of FFT window

Returns
list of frequency-domain features in pairs of feature name with column prefix and its value

Return type
List[*Tuple*[str, *DataFrame*]]

`extraction.fs_list_files(`*root_path*`)`

List csv and tsv files in dataset in the directory

Parameters
`root_path` (*str*) – base path of dataset directory

Returns
list of filenames

Return type
List[str]

`extraction.harmonic_series_detection(`*f*, *pxx*, *fs*, *fft_window*`)`

Find all series of harmonic frequencies in the frequency spectrum according to paper: "Identification of harmonics and sidebands in a finite set of spectral components" (https://hal.science/hal-00845120v2/document)

Parameters

- `f` (*array*) – array of frequency bins
- `pxx` (*array*) – array of amplitudes at frequency bins
- `fs` (*int*) – sampling frequency in Hz
- `fft_window` (*int*) – length of FFT window for calculation of frequency bin resolution

Returns
list of harmonic frequency series with components described by their frequency and amplitude

Return type
List[*List*[*Tuple*[int, float]]]

`extraction.have_intersection(`*interval1*, *interval2*`)`

Check overlap of two numeric intervals

Parameters

- `interval1` (*Tuple[float, float]*) – numbers for bounds of the first interval
- `interval2` (*Tuple[float, float]*) – numbers for bounds of the second interval

Returns
overlap of intervals is present

Return type
bool

`extraction.list_files(`*dataset*`)`

List csv and tsv files in dataset within ZIP archive

Parameters
`dataset` (*ZipFile*) – dataset in zip archive

Returns
list of filenames

Return type
List[str]

`extraction.load_features(`*filename*, *axis*, *label_columns*`)`

Load features from csv file and aggregate values from chosen directions of movement

Parameters

- `filename` (*str*) – File path where extracted features are found
- `axis` (*List[str]*) – Elements to combine one feature from
- `label_columns` (*List[str]*) – Columns that are not features and are not processed just copied

Returns
data frame with columns of unique feature names

Return type
Tuple[*DataFrame*, *DataFrame*]

`extraction.load_files_split(`*dataset*, *func*, *parts=1*, *cores=4*`)`

Load files from the dataset in ZIP archive and process them with callback function

Parameters

- `dataset` (*ZipFile*) – dataset in zip archive
- `func` (*Callable*) – callback for file processing that takes dataset, filename, and number of parts to split file into as parameters
- `parts` (*int*) – number of partitions to split each file into
- `cores` (*int*) – number of cores to use in workload parallelization

Returns
data frame with extracted features and associated annotations for row

Return type
DataFrame

`extraction.mms_peak_finder(`*x*, *win_len=3*`)`

Robust non-parametric peak identification MMS algorithm according to description in paper: "Non-Parametric Local Maxima and Minima Finder with Filtering Techniques for Bioprocess" (https://doi.org/10.4236/jsip.2016.74018)

Parameters

- `x` (*array*) – time series

- `win_len` (*int*) – window length of points compared in peak finding

Returns
array of indexes where peaks are in the time series

Return type
array

`extraction.negentropy(`*x*`)`

Calculate negentropy of the signal

Parameters
`x` (*array*) – input signal

Returns
negentropy of the signal

Return type
float

`extraction.signal_to_noise(`*x*`)`

Calculate estimated signal to noise ratio (noisiness) of the signal. Formula is taken from: https://www.geeksforgeeks.org/signal-to-noise-ratio-formula/

Parameters
`x` (*array*) – input signal

Returns
SNR of the signal

Return type
float

`extraction.spectral_roll_off_frequency(`*f*, *pxx*, *percentage*`)`

Calculate roll-off frequency. Cumulative sum of energy in spectral bins below roll-off frequency is percentage of total energy

Parameters

- `f` (*array*) – array of frequency bins
- `pxx` (*array*) – array of amplitudes at frequency bins
- `percentage` (*float*) – ratio of total energy below the roll-off frequency

Returns
roll-off frequency in Hz

Return type
float

`extraction.spectral_transform(`*x*, *window*, *fs*`)`

Estimate frequency spectrum using Welch's method. Partition the signal with Hann window and 50% overlap.

Parameters

- `x` (*Series*) – input signal in time domain
- `window` (*int*) – length of Hann FFT window
- `fs` (*int*) – sampling frequency in Hz of the input signal

Returns
envelope of the amplitudes

Return type
Tuple[array, array]

`extraction.split_dataframe(`*dataframe*, *parts=None*`)`

Split to data frames to non overlapping parts

Parameters

- `dataframe` (*DataFrame*) – data frame to be split
- `parts` (*int*) – number of partitions to split data frame into

Returns

list of data frame parts

Return type

List[*DataFrame*]

`extraction.temporal_variation(`*x*, *window*`)`

Calculate temporal variations in successive frequency spectra. It is a inverse correlation of pairs formed from overlapping windows.

Parameters

- `x` (*Series*) – input signal in time domain
- `window` (*int*) – length of Hann FFT window

Returns

array of temporal variations

Return type

List[float]

`extraction.time_features_calc(`*df*, *col*, *fs*, *window*`)`

Extract complete set of features in time-domain (TD set)

Parameters

- `df` (*DataFrame*) – data frame with input time series in columns
- `col` (*str*) – column in data frame to use for extraction
- `fs` (*int*) – sampling frequency in Hz of the time series
- `window` (*int*) – length of FFT window (not used)

Returns

list of time-domain features in pairs of feature name with column prefix and its value

Return type

List[*Tuple*[str, *DataFrame*]]

`extraction.wavelet_features_calc(`*df*, *col*, *fs*, *window*`)`

Extract wavelet coefficients for Meyer wavelet and six levels deep. Each wavelet coefficient produces four features (energy, energy ratio, kurtosis, negentropy)

Parameters

- `df` (*DataFrame*) – data frame with input time series in columns
- `col` (*str*) – column in data frame to use for extraction
- `fs` (*int*) – sampling frequency in Hz of the time series
- `window` (*int*) – length of FFT window (not used)

Returns

list of wavelet-domain features in pairs of feature name with column prefix and its value

Return type

List[*Tuple*[str, *DataFrame*]]

1.2 mafaulda module

`mafaulda.BEARINGS = {'ball_diameter': 0.7145, 'balls': 8, 'bpfi_factor': 5.002, 'bpfo_factor': 2.998, 'bsf_factor': 1.871, 'ftf_factor': 0.375, 'pitch_diameter': 2.8519}`

Coefficients for bearing characteristic frequencies

`mafaulda.FAULTS = {'A': {'horizontal-misalignment': 'misalignment', 'imbalance': 'imbalance', 'normal': 'normal', 'underhang-ball_fault': 'ball fault', 'underhang-cage_fault': 'cage fault', 'underhang-outer_race': 'outer race fault', 'vertical-misalignment': 'misalignment'}, 'B': {'horizontal-misalignment': 'misalignment', 'imbalance': 'imbalance', 'normal': 'normal', 'overhang-ball_fault': 'ball fault', 'overhang-cage_fault': 'cage fault', 'overhang-outer_race': 'outer race fault', 'vertical-misalignment': 'misalignment'}}`

Annotation of fault types by bearing placement

`mafaulda.LABEL_COLUMNS = ['fault', 'severity', 'rpm']`

Metadata columns extracted from file path within dataset

`mafaulda.SAMPLING_RATE = 50000`

Sampling frequency in Hz of the sensors

`mafaulda.assign_labels(`*df*, *bearing*, *keep=False*`)`

Assign labels to fault types for bearing and optionally clean up data frame to contain only annotated rows with faults

Parameters

- `df` (*DataFrame*) – data frame after feature extraction with column "fault"
- `bearing` (*str*) – bearing to determine labels of fault types ("A" or "B")
- `keep` (*bool*) – do not remove metadata columns

Returns
annotated data frame

Return type
DataFrame

`mafaulda.bearing_frequencies(`*rpm*`)`

Calculate bearing characteristic frequencies for MaFaulDa machine simulator

Parameters
`rpm` (*int*) – Rotational speed of the machine

Returns
Bearing defect frequencies

Return type
Dict[str, float]

`mafaulda.clean_columns(`*df*`)`

Remove excessive columns with metadata and drop rows without label

Parameters
`df` (*DataFrame*) – data frame with excess labels

Returns
data frame that consists of columns with features and "label"

Return type
DataFrame

`mafaulda.csv_import(`*dataset*, *filename*`)`

Open a CSV file from MaFaulda zip archive

Parameters

- `dataset` (*ZipFile*) – ZIP archive of MaFaulDa dataset
- `filename` (*str*) – path to the file within dataset

Returns
data frame of the imported file

Return type
DataFrame

`mafaulda.features_by_domain(`*features_calc*, *dataset*, *filename*, *window=None*, *parts=1*, *multirow=False*`)`

Open a CSV file from MaFaulda zip archive

Parameters

- `features_calc` (*Callable*) – callback feature extraction function that has parameters for data frame, column to process in data frame, sampling frequency, window length of segment
- `dataset` (*ZipFile*) – ZIP archive of MaFaulDa dataset
- `filename` (*str*) – path to the file within dataset
- `window` (*int*) – length of window (usually for FFT)
- `parts` (*int*) – number of parts the input time series is split into
- `multirow` (*bool*) – extracted features are in rows, not in columns

Returns
row(s) of features extracted from the file

Return type
DataFrame

`mafaulda.get_classes(`*df*, *bearing*`)`

Create column "label" in data frame according to chosen bearing

Parameters

- `df` (*DataFrame*) – data frame after feature extraction with column "fault"
- `bearing` (*str*) – bearing to determine labels of fault types ("A" or "B")

Returns
data frame with "label" column

Return type
DataFrame

`mafaulda.label_severity(`*df*, *bearing*, *level*, *debug=False*, *keep=False*`)`

Relabel faults less than set relative severity level as "normal"

Parameters

- `df` (*DataFrame*) – data frame after feature extraction with column "fault"
- `bearing` (*str*) – bearing to determine labels of fault types ("A" or "B")
- `debug` (*bool*) – print relative severity levels
- `keep` (*bool*) – do not remove metadata columns
- `level` (*float*)

Returns
data frame with relabeled observations

Return type
DataFrame

`mafaulda.load_source(`*domain*, *row*, *train_size=0.8*`)`

Load features according to domain and split the observations into training and testing set

Parameters

- `domain` (*str*) – complete feature set ("TD", "FD")
- `row` (*dict*) – parameters for data filtering, e.g.: {"placement": "A", online: False}
- `train_size` (*float*) – ratio of training set to testing set

Returns
X_train, X_test, Y_train, Y_test

Return type
tuple

`mafaulda.lowpass_filter(`*data*, *cutoff=10000*, *fs=50000*, *order=5*`)`

Low-pass filter of n-th order the input signal at the cutoff frequency

Parameters

- `data` (*Series*) – input signal
- `cutoff` (*int*) – cutoff frequency
- `fs` (*int*) – sampling frequency in Hz of the input signal
- `order` (*int*) – steps of the filter

Returns
output signal after filtering

Return type
Series

`mafaulda.lowpass_filter_extract(`*dataframes*, *columns*`)`

Apply low-pass filter to columns in multiple data frames

Parameters

- `frames` (*data*) – list of input datafrmaes to which filter is applied to
- `columns` (*List [str]*) – columns that filter is applied to
- `dataframes` (*List [DataFrame]*)

Returns
list of data frames after filtering

Return type
List[*DataFrame*]

`mafaulda.mark_severity(`*df*, *bearing*, *debug=False*`)`

Calculate relative severity levels for data frame with original metadata columns

Parameters

- `df` (*DataFrame*) – data frame after feature extraction with column "fault" and "severity"
- `bearing` (*str*) – bearing to determine labels of fault types ("A" or "B")
- `debug` (*bool*) – print relative severity levels

Returns
data frame with columns for absolute and relative fault severity levels

Return type
DataFrame

`mafaulda.parse_filename(`*filename*`)`

Split path of file within dataset structure to label the time series

Parameters
`filename` (*str*) – path to file inside of zip archive

Returns
fault type, severity conditions, and file number

Return type
Tuple[str, str, str]

`mafaulda.rpm_calc(`*tachometer*`)`

Extract rotational speed in rpm units from tachometer pulse signal

Parameters
`tachometer` (*Series*) – tachometer signal

Returns
rotational speed in rpm

Return type
float

1.3 models module

`models.accuracies_to_table(`*domain*, *set*, *distribution*, *accuracy*`)`

Format accuracy to a row with metadata about hyperparamater and compute percentiles in the model accuracy distribution

Parameters

- `domain` (*str*) – source domain from which the features are extracted ("TD" or "FD")
- `set` (*str*) – Title for the feature set or selection method
- `distribution` (*DataFrame*) – accuracy distribution of the model
- `accuracy` (*DataFrame*) – accuracy of the model in training and testing set

Returns
formatted structure for row of accuracies and percentiles

Return type
dict

`models.all_features(`*X*, *Y*, *model_name='knn'*, *power_transform=False*, *k_neighbors=[1, 5, 9, 13, 17, 21, 25, 29, 33, 37]*, *kfolds=5*`)`

Evaluate complete feature sets in k-nearest neighbours classifier
with various k-value parameter

Parameters

- `X` (*DataFrame*) – data frame of predictor features
- `Y` (*Series*) – column of labels

- `model_name` (*str*) – name of the machine learning model to evaluate. Options are: "knn", "lda", "bayes", "svm"
- `power_transform` (*bool*) – apply power transform of features in preprocessing instead of normalization
- `k_neighbors` (*list*) – number of neighbours for k-nearest neighbours model
- `kfolds` (*int*) – number of splits for k-fold cross-validation

Returns
training and testing accuracy for each value of k-neighbours

Return type
Dict[str, float]

`models.enumerate_models(`*X*, *Y*, *domain*, *k_neighbors=(3, 5, 11, 15)*, *num_of_features=(2, 3, 4, 5)*, *kfolds=5*, *power_transform=False*, *model='knn'*`)`

Grid search of parameters k-nearest neighbours classifier with feature subset combinations

Parameters

- `X` (*DataFrame*) – data frame of predictor features
- `Y` (*DataFrame*) – column of labels
- `domain` (*str*) – source domain from which the features are extracted ("TD" or "FD")
- `k_neighbors` (*Tuple[int]*) – neighbours for k-nearest neighbours model to search in
- `num_of_features` (*Tuple[int]*) – number of features in the subset to search in
- `kfolds` (*int*) – number of splits for k-fold cross-validation
- `power_transform` (*bool*) – apply power transform of features in preprocessing instead of normalization
- `model` – name of the machine learning model to evaluate. Options are: "knn", "lda", "bayes", "svm"

Returns
training and testing accuracy for each hyperparameter and feature set combination

Return type
DataFrame

`models.feature_combinations(`*X*, *Y*, *k_neighbors*, *num_of_features*, *kfolds*, *domain*, *model*, *power_transform=False*`)`

Evaluate all combinations of feature subsets of given size out of complete sets

Parameters

- `X` (*DataFrame*) – data frame of predictor features
- `Y` (*DataFrame*) – column of labels
- `k_neighbors` (*int*) – number of neighbours for k-nearest neighbours model
- `num_of_features` (*int*) – number of features in the subset
- `kfolds` (*int*) – number of splits for k-fold cross-validation
- `domain` (*str*) – source domain from which the features are extracted
- `model` (*str*) – name of the machine learning model to evaluate. Options are: "knn", "lda", "bayes", "svm"

- `power_transform` (*bool*) – apply power transform of features in preprocessing instead of normalization

Returns
training and testing accuracy for each combination of features

Return type
List[dict]

`models.feature_selection_accuracies(`*X*, *Y*, *domain*, *models_summary*, *k_neighbors*, *number_of_features*, *power_transform=False*`)`

Apply feature selection methods and evaluate accuracies for chosen number of neighbours and number of features

Parameters

- `X` (*DataFrame*) – data frame of predictor features
- `Y` (*DataFrame*) – column of labels
- `domain` (*str*) – source domain from which the features are extracted ("TD" or "FD")
- `k_neighbors` (*int*) – neighbours for k-nearest neighbours model
- `number_of_features` (*int*) – number of features in the subset
- `power_transform` (*bool*) – apply power transform of features in preprocessing instead of normalization
- `models_summary` (*DataFrame*)

Params model_summary
accuracies from all feature subset combinations

Returns
accuracies and percentiles for all tested feature selection methods

Return type
List[*Dict*[str, int]]

`models.find_best_subset(`*X*, *Y*, *metric*, *members*, *kfolds=5*`)`

Find the best subset of features based on supplied feature selection metric name

Parameters

- `X` (*DataFrame*) – data frame of predictor features
- `Y` (*Series*) – column of labels
- `metric` (*str*) – name of the bivariate similarity metric to compute for each feature and label
- `members` (*int*) – number of features in the subset
- `kfolds` (*int*) – number of splits for k-fold cross validation

Returns
list of the best features according to feature selection metric

Return type
List[str]

`models.kfold_accuracy(`*X*, *Y*, *k_neighbors*, *kfolds*, *model_name='knn'*, *power_transform=True*, *knn_metric='euclidean'*`)`

Evaluate classifier accuracy in k-fold validation on a data frame of features after oversampling to the majority class

Parameters

- `X` (*DataFrame*) – data frame of predictor features
- `Y` (*Series*) – column of labels
- `k_neighbors` (*int*) – number of neighbours for k-nearest neighbours model
- `model_name` (*str*) – name of the machine learning model to evaluate. Options are: "knn", "lda", "bayes", "svm"
- `kfolds` (*int*) – number of splits for k-fold cross-validation
- `power_transform` (*bool*) – apply power transform of features in preprocessing instead of normalization
- `knn_metric` – distance metric name for k-nearest neighbours model

Returns
average accuracy of model over k-folds in training and testing sets

Return type
Dict[str, float]

`models.knn_online_learn(`*X*, *Y*, *window_len=1*, *learn_skip=0*, *clusters=False*, *n_neighbors=5*`)`

Progressive valuation of k-nearest neighbours classifier trained with increamental learning

Parameters

- `X` (*DataFrame*) – data frame of predictor features
- `Y` (*DataFrame*) – column of labels
- `window_len` (*int*) – Length of the tumbling window
- `learn_skip` (*int*) – Gap of labeled observations in amount of samples
- `clusters` (*int*) – return data points instead of valuation
- `n_neighbors` (*int*) – number of neighbours for k-nearest neighbours model

Returns
performance of the model in progressive valuation over all generations

Return type
DataFrame

`models.model_boundaries(`*X*, *Y*, *n=5*, *model_name='knn'*, *knn_metric='euclidean'*`)`

Train k-nearest neighbours classifier to be used in determining its decision boundaries

Parameters

- `X` (*DataFrame*) – data frame of predictor features
- `Y` (*DataFrame*) – column of labels
- `n` (*int*) – number of neighbours for k-nearest neighbours model
- `model_name` (*str*) – name of the machine learning model to evaluate. Options are: "knn", "lda", "bayes", "svm"
- `knn_metric` (*str*) – distance metric name for k-nearest neighbours model

Returns
model fitted with training data of 80% from the original dataset

`models.transform_to_pca(`*X*, *n*`)`

Transform features to their principal components after normalization

Parameters

- `dataset` – data frame with columns only for predictors
- `n` (*int*) – number of principal components

- `X` (*DataFrame*)

Returns
data frames with rows of features replaced for principal components

Return type
DataFrame

1.4 pumps module

`pumps.LABEL_COLUMNS = ['date', 'device', 'position']`
Metadata columns extracted from file path within dataset

`pumps.SAMPLING_RATE = 26866`
Sampling frequency in Hz of the sensors

`pumps.assign_labels(`*df*`)`
Assign labels to fault types into "label" column based on measurement placement

Parameters
`df` (*DataFrame*) – data frame after feature extraction with column "fault"

Returns
annotated data frame

Return type
DataFrame

`pumps.beaglebone_measurement(`*filename*, *fs*`)`
Import csv file recorded on BeagleBone Black with accelerometer ADXL335

Parameters

- `filename` (*str*) – file name of the recording
- `fs` (*int*) – sampling frequency in Hz

Returns
data frame of the recording

Return type
Tuple[str, *DataFrame*]

`pumps.csv_import(`*dataset*, *filename*`)`
Open a CSV file from Pump zip archive

Parameters

- `dataset` (*ZipFile*) – ZIP archive of Pump industrial dataset
- `filename` (*str*) – path to the file within dataset

Returns
data frame of the imported file

Return type
DataFrame

`pumps.features_by_domain(`*features_calc*, *dataset*, *filename*, *window=None*, *parts=None*`)`
Open a CSV file from Pump zip archive

Parameters

- `features_calc` (*Callable*) – callback feature extraction function that has parameters for data frame, column to process in data frame, sampling frequency, window length of segment

- `dataset` (*ZipFile*) – ZIP archive of Pump dataset
- `filename` (*str*) – path to the file within dataset
- `window` (*int*) – length of window (usually for FFT)
- `parts` (*int*) – number of parts the input time series is split into

Returns
row(s) of features extracted from the file

Return type
DataFrame

`pumps.features_by_domain_no_metadata(`*features_calc*, *filename*, *window=None*, *parts=None*`)`

Parameters

- `features_calc` (*Callable*)
- `filename` (*str*)
- `window` (*int*)
- `parts` (*int*)

Return type
DataFrame

`pumps.get_classes(`*df*, *labels*, *keep=False*`)`

Assign labels to fault types into "label" column and optionally clean up data frame to contain only annotated rows with faults

Parameters

- `df` (*DataFrame*) – data frame after feature extraction with column "fault"
- `labels` (*Dict[str, dict]*) – Llbels to assign machine and measurement placement
- `keep` (*bool*) – do not remove metadata columns

Returns
annotated data frame

Return type
DataFrame

1.5 ranking module

`class ranking.ExperimentOutput(`*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

Bases: `Enum`

Types of experimental scenarios for feature selection techniques

`BEST_CORR = 7`

`BEST_F_STAT = 8`

`BEST_MI = 9`

`BEST_SET = 2`

`COUNTS = 1`

PCA = 5

RANKS = 3

SCORES_RANGE = 4

SILHOUETTE = 6

`ranking.batch_feature_ranking(`*X*, *Y*, *mode='rank'*`)`

Order features based on their importance

Parameters

- `X` (*DataFrame*) – data frame that contains features
- `Y` (*Series*) – labels for observations
- `mode` (*str*) – feature selection method, options: "corr", "f_stat", "mi", "rank"

Returns

Sorted features with their scores

Return type

DataFrame

`ranking.best_columns(`*ranks*, *corr*, *n*`)`

Retain the best features that does not belong to correlated set

Parameters

- `ranks` (*DataFrame*) – features with their scores
- `corr` (*Set[Tuple[str, str]]*) – set of pairs with high correlations
- `n` (*int*) – number of best features to keep

Returns

list of best features

Return type

List[str]

`ranking.best_subset(`*ranks*, *corr*, *n*`)`

Retain the best features

Parameters

- `ranks` (*DataFrame*) – features with their scores
- `corr` (*Set[Tuple[str, str]]*) – set of pairs with high correlations
- `n` (*int*) – number of best features to keep

Returns

best features

Return type

DataFrame

`ranking.compute_correlations(`*X*, *corr_above*`)`

Find pairs of features correlated more than the threshold

Parameters

- `X` (*DataFrame*) – data frame that contains features
- `corr_above` (*float*) – correlation threshold level

Returns

pairs or correlated features

Return type
Set[*Tuple*[str, str]]

`ranking.online_feature_ranking(`*X*, *Y*, *mode='rank'*`)`

Sort features by gradual process

Parameters

- `X` (*DataFrame*) – data frame with sequence of events
- `Y` (*Series*) – labels for observations
- `mode` (*str*) – feature selection method, options: "corr", "f_stat", "mi", "rank"

Returns
leaderboard of the features

Return type
DataFrame

`ranking.pca_explained_variances(`*X_train*, *pc*`)`

Explained variances of the principal components

Parameters

- `X_train` (*DataFrame*) – data points of the training set
- `pc` (*int*) – number of principal components

Returns
dictionary of principal components and explained variances

Return type
Dict[str, float]

`ranking.silhouette_scores(`*X_train*, *X_test*, *Y_train*, *Y_test*, *best_features*, *pc*`)`

Calculate silhouette score of data points after normalization of training and testing set with and without the principal components analysis

Parameters

- `X_train` (*DataFrame*) – data points of the training set
- `X_test` (*DataFrame*) – data points of the testing set
- `Y_train` (*DataFrame*) – labels for the training set
- `Y_test` (*DataFrame*) – labels for the testing set
- `best_features` (*List [str]*) – list of chosen feature names
- `pc` (*int*) – number of principal components

Returns
silhouette scores for data points

Return type
Dict[str, float]

1.6 selection module

`class selection.Correlation`

Bases: `Bivariate`

Online correlation to classes as dichotomous variables

`get()`

Return the current value of the statistic.

`update(x, y)`

Update and return the called instance.

`class selection.FisherScore`

Bases: `Bivariate`

Online F statistic

`get()`

Return the current value of the statistic.

`update(x, y)`

Update and return the called instance.

`class selection.MutualInformation`

Bases: `Bivariate`

Online Mutual information to binned labels

`get()`

Return the current value of the statistic.

`update(x, y)`

Update and return the called instance.

`selection.corr_classif(X, y)`

Calculate point-biserial correlations to features

Parameters

- `X` (*array*) – matrix of features
- `y` (*array*) – labels of observations

Returns

list of absolute value of correlations between classes and features

Return type

array

1.7 visualize module

`visualize.DOMAIN_TITLES = {'FD': 'Frequency domain', 'TD': 'Time domain', 'TD+FD': 'Time and Frequency domain'}`

Titles for signal source domain abbreviation

`visualize.boxplot_enumerate_models_accuracy(results, metric, plots_col, inplot_col)`

Boxplot of model accuracy distributions for various

number of features or neighbours

Parameters

- `results` (*DataFrame*) – model accuracy distributions
- `metric` (*str*) – column of values to show in values for accuracy
- `plots_col` (*str*) – constant parameter for subplot ("f" or "k")
- `inplot_col` (*str*) – comparison of different values for the parameter within subplot ("f" or "k")

`visualize.cross_cuts_3d_cluster(`*X_train*, *cluster*, *title*`)`

Scatter plot of clusters in 3D feature space shown in planar cross-sections through coordinate axes

Parameters

- `X_train` (*DataFrame*) – data frame of features
- `cluster` (*str*) – clusters that observations belong to
- `title` (*str*) – figure title

`visualize.evolution_of_severity_levels(`*df*`)`

Line chart of the amount of observations at relative severity levels

Parameters

`df` (*DataFrame*) – data frame with sorted "severity" level column

`visualize.loading_plot(`*loadings*, *feature_names*, *bottom*, *top*`)`

Loading plot of features

Parameters

- `loadings` (*list*) – Relation of features to coordinates that are created by two principal components
- `feature_names` (*List[str]*) – list of feature names corresponding to their loadings
- `bottom` (*float*) – lower limit of graph coordinates in x and y axes
- `top` (*float*) – upper limit of graph coordinates in x and y axes

`visualize.plot_all_knn(`*td_results*, *fd_results*`)`

Line chart of relationship of k-value to k-NN classifier accuracy

Parameters

- `td_results` (*Dict[str, float]*) – lists of k-values and accuracies for time-domain features in training and testing set
- `fd_results` (*Dict[str, float]*) – lists of k-values and accuracies for frquency-domain features in training and testing set

`visualize.plot_cumulative_explained_variance(`*td_variance*, *fd_variance*`)`

Line chart of relationship of number of principal components to total explained variance

Parameters

- `td_variance` (*array*) – Explained variances for time-domain features
- `fd_variance` (*array*) – Explained variances for frequency-domain features

`visualize.plot_label_occurences(`*y*`)`

Line chart of counters for classes in incremental learning

Parameters

`y` (*Series*) – sorted labels of observations `visualize.plot_models_performance_bar(`*results*`)`

Bar chart of feature selection accuracy comparison

Parameters

`results` (*DataFrame*) – training and testing accuracies of seven feature selection methods in both source domains

`visualize.project_classes(`*X*, *Y*, *size=(10, 8)*, *boundary=False*, *model_name='knn'*, *pc=None*`)`

Scatter plot of two principal components of data points in feature space

Parameters

- `X` (*DataFrame*) – data frame of features
- `Y` (*DataFrame*) – labels of observations
- `size` (*tuple*) – figure size
- `boundary` (*bool*) – show decision boundary for k-NN model with 5 neighbours
- `model_name` (*str*) – name of the machine learning model to evaluate. Options are: "knn", "lda", "bayes", "svm"
- `pc` (*int*) – number of principal components

`visualize.project_classes_3d(`*X*, *Y*, *size=(15, 6)*`)`

Scatter plot of three principal components of data points in feature space shown in planar cross-sections through coordinate axes

Parameters

- `X` (*DataFrame*) – data frame of features
- `Y` (*DataFrame*) – labels of observations
- `size` (*tuple*) – figure size

`visualize.project_classifier_map_plot(`*X*, *y_true*, *y_predict*`)`

Scatter plots of two principal components from data points that shows mistakes in prediction versus true labels

Parameters

- `X` (*DataFrame*) – data frame of features
- `y_true` (*Series*) – true labels of observations
- `y_predict` (*Series*) – predicted labels of observations

`visualize.scatter_classif(`*X*, *y_label*, *categories*, *colors*, *ax*`)`

Scatter plot of data points with color based on their labels

Parameters

- `X` (*DataFrame*) – data frame of features
- `y_label` (*Series*) – labels of observations
- `categories` (*List [str]*) – list of unique clasess
- `colors` (*List [str]*) – list of colors for clasess
- `ax` – subplot axis

`visualize.scatter_features_3d(`*X*, *Y*, *features*, *size=(15, 5)*, *boundary=False*, *model_name='knn'*, *power_transform=False*`)`

Scatter plot of data points in 3D feature space shown in planar cross-sections through coordinate axes

Parameters

- X (*DataFrame*) – data frame of features
- Y (*DataFrame*) – labels of observations
- features (*list*) – names of three features to display
- size (*tuple*) – figure size
- boundary (*bool*) – show decision boundary for k-NN model with 5 neighbours
- model_name (*str*) – name of the machine learning model to evaluate. Options are: "knn", "lda", "bayes", "svm"
- power_transform (*bool*) – apply power transform of features in preprocessing instead of normalization

visualize.scatter_features_3d_plot(*X*, *Y*, *features*, *size=(8, 8)*, *boundary=False*, *model_name='knn'*, *power_transform=False*)

Three dimensional scatter plot of data points in feature space

Parameters

- X (*DataFrame*) – data frame of features
- Y (*DataFrame*) – labels of observations
- features (*list*) – names of three features to display
- size (*tuple*) – figure size
- boundary (*bool*) – show decision boundary for k-NN model with 5 neighbours
- model_name (*str*) – name of the machine learning model to evaluate. Options are: "knn", "lda", "bayes", "svm"
- power_transform (*bool*) – apply power transform of features in preprocessing instead of normalization

PYTHON MODULE INDEX

INDEX