src.DataGenerator namespace

Submodules

src.DataGenerator.DataGenerator module

class **src.DataGenerator.DataGenerator.DataGenerator(*n_samples, n_features, n_informative, n_classes, relevance_decrease='linear'*)**

Bases: `object`

This class provides a way to create artificial datasets with user-defined characteristics.

This function initializes the DataGenerator object.

Pre-conditions: none.
Post-conditions: a new DataGenerator object is created.
Main output: none.

Parameters:
- **n_samples** (*int.*) – number of samples.
- **n_features** (*int.*) – total number of features.
- **n_informative** (*int.*) – number of informative features.
- **n_classes** (*int.*) – number of classes (or labels) of the classification problem.
- **relevance_decrease** (*str.*) – type of strategy used for decreasing feature relevance.

Returns: none.

Raise: none.

make_data(*random_state, n_clusters=2*)

This function generates random data for a classification problem. It considers n_informative features and a certain number of redundant features. Noise is applied to redundant features in an increasing manner.

Pre-conditions: :py:meth:'DataGenerator.__init__'.
Post-conditions: a new artificial dataset is created.
Main output: the new dataset, split into features and labels.

Parameters:
- **random_state** (*int, RandomState instance or None.*) – determines random number generation for dataset creation. Pass an int for reproducible output across multiple function

calls.
- **n_clusters** (*int (>0), optional.*) – number of clusters per class. Default 2.

Returns: features and labels of the new problem.

Return type: ndarray (n_samples, n_features), ndarray (n_samples,)

Raise:

make_data_blob(*n_equal*, *random_state*)

This function computes centroid positions so that they will be orthogonal with respect to N random axis.

Pre-conditions: :py:meth:'DataGenerator.__init__'..
Post-conditions: a new artificial dataset is created.
Main output: the new dataset, split into features and labels.

Parameters:
- **n_equal** (*int (>0)*) – number of features made up by the same value for all classes.
- **random_state** (*int, RandomState instance or None.*) – determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls.

Returns: features and labels of the new problem.

Return type: ndarray (n_samples, n_features), ndarray (n_samples,)

Raise:

src.DRCW namespace

Submodules

src.DRCW.DRCW module

***class* src.DRCW.DRCW.DRCW(*n_classes*, *subject_id*, *models_path='models/'*, *results_path='results/'*, *samples_balancing='none'*, *n_neighbors=5*)**

Bases: `sklearn.base.BaseEstimator`

This class represents the skeleton for DRCW-OVO architecture, described in https://doi.org/10.1016/j.patcog.2014.07.023.

This function initialises the DRCW object using the given parameters. This will be the skeleton of DRCW-OVO architecture.

Pre-conditions: none.
Post-conditions: a new DRCW-OVO object is created.
Main output: none.

Parameters:

- **n_classes** (*int (>1)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the considered subject.
- **models_path** (*str, optional.*) – path in which models are saved. Default 'models/'.
- **results_path** (*str, optional.*) – path in which results are saved. Default 'results/'.
- **samples_balancing** (*str, optional.*) – balancing strategy adopted in case of non-balanced sub-problem. One between 'none', 'random', 'stratified_random', 'clustering', 'smote', 'nearmiss'. In case of '1vs1', 'stratified_random' and 'random' are the same (use 'random'). Default 'random'.
- **n_neighbors** (*int (>0)*) – number of neighbors used by DRCW-OVO's KNNs.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the DRCW-OVO architecture on the given training set.

Pre-conditions: :py:meth:'DRCW.__init__'.
Post-conditions: the NAWAXOVO/NAWAXOVA object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the architecture is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples.

Pre-conditions: :py:meth:'DRCW.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples*n_features)*) – a matrix containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

processing_history_predict(*x, y, ids*)

This function computes the predictions for the given samples.

Pre-conditions: :py:meth:'DRCW.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix containing the features of each sample for which prediction is needed.
- **y** (*ndarray (n_samples,)*) – an array containing labels for the given samples.
- **ids** (*ndarray (n_samples,)*) – an array containing the ids of the given samples.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

src.DRCW.OVOBaseLayer module

***class* src.DRCW.OVOBaseLayer.OVOBaseLayer(*n_classes, subject_id, models_path, results_path, samples_balancing*)**

Bases: `object`

This class implements the base layer for the DRCW-OVO architecture. Base classifiers are trained in a 1 vs 1 manner.

This function initializes the OVOBaseLayer object.

Pre-conditions: none.
Post-conditions: a new OVOBaseLayer (DRCW version) object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.
- **samples_balancing** (*str.*) – balancing strategy adopted in case of non-balanced sub-problem. One between 'none', 'random', 'clustering', 'smote', 'nearmiss'.

Returns: none.

Raise: none.

fit(*x, y*)

This function fits the 1 vs 1 classifiers of the base layer on the given training set.

Pre-conditions: :py:meth:'OVOBaseLayer.__init__'.
Post-conditions: the OVOBaseLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, n_classes, n_classes) matrix representing the probabilities for each class. Considered a single sample, the (n_classes, n_classes) matrix represents the probabilities for each possible pair of classes. Sum of symmetric elements of this matrix is equal to 1.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.EEGTroika namespace

Submodules

src.EEGTroika.EEGMetaClassifiersLayer module

class **src.EEGTroika.EEGMetaClassifiersLayer.EEGMetaClassifiersLayer(*n_classes*, *subject_id*, *models_path*, *results_path*)**

Bases: `object`

This class implements the meta layer for the EEGTroika architecture. Meta classifiers are trained in a 1 vs ALL manner.

This function initializes the EEGMetaClassifiersLayer object.

Pre-conditions: none.
Post-conditions: a new EEGMetaClassifiersLayer object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the meta classifiers on the given training set.

Pre-conditions: :py:meth:'EEGMetaClassifiersLayer.__init__'.
Post-conditions: the EEGMetaClassifiersLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, n_classes) matrix representing the probabilities for each class.

Pre-conditions: :py:meth:'EEGMetaClassifiersLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: predictions (probabilities) for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.EEGTroika.EEGSpecialistClassifiersLayer module

***class* src.EEGTroika.EEGSpecialistClassifiersLayer.EEGSpecialistClassifiersLayer(*n_classes, subject_id, models_path, results_path*)**

Bases: `object`

This class implements the specialist layer for the EEGTroika architecture. Specialist classifiers are trained in a 1 vs 1 manner.

This function initializes the EEGSpecialistClassifiersLayer object.

Pre-conditions: none.
Post-conditions: a new EEGSpecialistClassifiersLayer object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.

Returns: none.

Raise: none.

fit(*x, y*)

This function fits specialist classifiers on the given training set.

Pre-conditions: :py:meth:'EEGSpecialistClassifiersLayer.__init__'.
Post-conditions: the EEGSpecialistClassifiersLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, n_specialist) matrix representing the probabilities from each specialist (probabilities refer to the class with lowest index).

Pre-conditions: :py:meth:'EEGSpecialistClassifiersLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: predictions (probabilities) for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_specialist)

Raise: none.

src.EEGTroika.EEGSuperClassifierLayer module

***class* src.EEGTroika.EEGSuperClassifierLayer.EEGSuperClassifierLayer(*models_path, results_path, n_classes, subject_id*)**

Bases: `object`

This class implements the super classifier layer for the EEGTroika architecture. Super classifier is trained in a ALL vs ALL manner.

This function initializes the EEGSuperClassifierLayer object.

Pre-conditions: none.
Post-conditions: a new EEGSuperClassifierLayer object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the super classifier on the given training set.

Pre-conditions: :py:meth:'EEGSuperClassifierLayer.__init__'.
Post-conditions: the EEGSuperClassifierLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples,) array representing the final predictions.

Pre-conditions: :py:meth:'EEGSuperClassifierLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

src.EEGTroika.EEGTroika module

***class* src.EEGTroika.EEGTroika.EEGTroika(*n_classes*, *subject_id*, *models_path='models/'*, *results_path='results/'*)**

Bases: `sklearn.base.BaseEstimator`

This class represents the skeleton for EEGTroika architecture.

This function initialises the EEGTroika object using the given parameters. This will be the skeleton of EEGTroika architecture.

Pre-conditions: none.
Post-conditions: a new EEGTroika object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the considered subject.
- **models_path** (*str, optional.*) – path in which models are saved. Default 'models/'.
- **results_path** (*str, optional.*) – path in which results are saved. Default 'results/'.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the EEGTroika architecture on the given training set.

Pre-conditions: :py:meth:'EEGTroika.__init__'.
Post-conditions: the EEGTroika object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the architecture is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples.

Pre-conditions: :py:meth:'EEGTroika.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples*n_features)*) – a matrix containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

src.Metrics namespace

Submodules

src.Metrics.Metrics module

class **src.Metrics.Metrics.Metrics**

Bases: `object`

This class implements some static methods useful to compute metrics such as reciprocal_rank, mean_reciprocal_rank, average_precision, mean_average_precision, spearman rho distance, discounted cumulative gain.

> ***classmethod*** **average_precision(*true_order*, *order*, *percentage*)**
>
> This function computes the Average Precision. Average Precision is computed as described in https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval).
>
> Pre-conditions: none.
> Post-conditions: Average Precision is computed.
> Main output: Average Precision.
>
> **Parameters:**
> - **true_order** (*ndarray (n_elements,)*) – an array representing the 'correct' order of the elements.
> - **order** (*ndarray (n_elements,)*) – an array representing the order of the elements.
> - **percentage** (*float (in [0, 1])*) – percentage of elements considered as relevants.
>
> **Returns:** Average Precision.
>
> **Return type:** float.
>
> **Raise:** none.

> ***classmethod*** **discounted_cumulative_gain(*true_order*, *order*, *percentage*)**

This function returns the DCG between true_order and order. See https://en.wikipedia.org/wiki/Discounted_cumulative_gain for more info.

Pre-conditions: none.
Post-conditions: DCG is computed.
Main output: Discounted Cumulative Gain.

Parameters:
- **true_order** (*ndarray (n_elements,)*) – an array representing the 'correct' order of the elements.
- **order** (*ndarray (n_elements,)*) – an array representing the order of the elements.
- **percentage** (*float (in [0, 1])*) – percentage of elements considered as relevants.

Returns: Discounted Cumulative Gain (DCG).

Return type: float.

Raise: none.

classmethod **mean_average_precision(*true_order, order, percentage*)**

This function computes the Average Precision. Average Precision is computed as described in https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval).

Pre-conditions: none.
Post-conditions: Mean Average Precision is computed.
Main output: Mean Average Precision.

Parameters:
- **true_order** (*ndarray (n_query, n_elements)*) – a matrix representing the 'correct' order of the elements. Each row represent a different query.
- **order** (*ndarray (n_query, n_elements)*) – a matrix representing the order of the elements to be analyzed. Each row represent a different query.
- **percentage** (*float (in [0, 1])*) – percentage of elements considered as relevants.

Returns: Mean Reciprocal Rank (MRR).

Return type: float.

Raise: none.

classmethod **mean_reciprocal_rank(*true_order, order, percentage*)**

This function computes the Mean Reciprocal Rank (MRR). MRR is computed as the reciprocal of the index of first relevant element in order.

Pre-conditions: none.
Post-conditions: MRR is computed.
Main output: Mean Reciprocal Rank.

Parameters:
- **true_order** (*ndarray (n_query, n_elements)*) – a matrix representing the 'correct' order of the elements. Each row represent a different query.
- **order** (*ndarray (n_query, n_elements)*) – a matrix representing the order of the elements to be analyzed. Each row represent a different query.
- **percentage** (*float (in [0, 1])*) – percentage of elements considered as relevants.

Returns: Mean Reciprocal Rank (MRR).

Return type: float.

Raise: none.

classmethod **percentage_of_informative_at_n(*informative_features, order, n*)**

This function returns the percentage (in [0, 1]) of informative features contained in the first n elements of order.

Pre-conditions: none.
Post-conditions: Percentage of informative features is computed.
Main output: Percentage of informative features.

Parameters:
- **informative_features** (*ndarray (n_informative,)*) – an array containing the list of informative features.
- **order** (*ndarray (n_elements,)*) – an array representing the order of the elements.
- **n** (*int (>0)*) – number of elements of order to analyze. Only the first n elements of order are considered.

Returns: the percentage of informative features contained in the first n elements of order.

Return type: float (in [0, 1])

Raise: none.

classmethod **reciprocal_rank(*true_order, order, percentage*)**

This function computes the Reciprocal Rank. RR is computed as the reciprocal of the index of first relevant element in order.

Pre-conditions: none.
Post-conditions: RR is computed.
Main output: Reciprocal Rank.

Parameters:
- **true_order** (*ndarray (n_elements,)*) – an array representing the 'correct' order of the elements.
- **order** (*ndarray (n_elements,)*) – an array representing the order of the elements.
- **percentage** (*float (in [0, 1])*) – percentage of elements considered as relevants.

Returns: Reciprocal Rank.

Return type: float.

Raise: none.

***classmethod* spearman_rho_distance(*true_order*, *order*)**

This function computes the spearman rho distance between true_order and order. It is the square root of the sum of the squared differences between the real ranking of an element and the current ranking.

Pre-conditions: none.
Post-conditions: Spearman Rho Distance is computed.
Main output: Spearman Rho Distance.

Parameters:
- **true_order** (*ndarray (n_elements,)*) – an array representing the 'correct' order of the elements.
- **order** (*ndarray (n_elements,)*) – an array representing the order of the elements.

Returns: spearman rho distance between true_order and order.

Return type: float.

Raise: none.

src.NAWAX namespace

Submodules

src.NAWAX.NAWAX module

***class* src.NAWAX.NAWAX.NAWAX(*n_classes*, *subject_id*, *models_path='models/'*, *results_path='results/'*, *n_neighbors=5*, *base_layer_mode='1vs1'*, *samples_balancing='random'*)**

Bases: `sklearn.base.BaseEstimator`

This class represents the skeleton for NAWAXOVO/NAWAXOVA architecture.

This function initialises the NAWAX object using the given parameters. This will be the skeleton of NAWAXOVO/A architecture.

Pre-conditions: none.
Post-conditions: a new NAWAXOVO object is created.
Main output: none.

Parameters:

- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the considered subject.
- **models_path** (*str, optional.*) – path in which models are saved. Default ‘models/’.
- **results_path** (*str, optional.*) – path in which results are saved. Default ‘results/’.
- **n_neighbors** (*int (>0)*) – number of neighbors used by NAWAXOVO/A’s KNNs.
- **base_layer_mode** (*str, optional.*) – one between ‘1vs1’ and ‘1vsALL’. If ‘1vs1’, base classifiers are trained in a 1 vs 1 manner (NAWAXOVO). If ‘1vsALL’, base classifiers are trained in a 1 vs ALL manner (NAWAXOVA). Default ‘1vs1’.
- **samples_balancing** (*str, optional.*) – balancing strategy adopted in case of non-balanced sub-problem. One between ‘none’, ‘random’, ‘stratified_random’, ‘clustering’, ‘smote’, ‘nearmiss’. In case of ‘1vs1’, ‘stratified_random’ and ‘random’ are the same (use ‘random’). Default ‘random’.

Returns: none.

Raise: none.

explain(*x, ids, y=None, mode='all_classes', q=3, k=5, p=0.04*)

This function computes the explanations related to the samples x. For each sample, q nearest neighbors belonging to the majority class (excluded the predicted one)("majority_class"), to all classes (excluded the predicted one) ("all_classes") or to the correct class ("error") are considered, and their explanations are computed. The most similar sample (in the explanations space) is taken as neighbor. Features are ordered by decreasing importance for the classification of the current sample, and the less important are discarded (according to the p parameter). For each feature, k points are taken from the interval [min(sample_feature, neighbor_feature), max(sample_feature, neighbor_feature)]. Cartesian product with the previously extracted points is computed, new artificial points are created, and their predictions are finally obtained. The algorithm stop when the prediction of an artificial point matches the class of the neighbor (or when the most important features have been analyzed).

Pre-conditions: :py:meth:'NAWAX.fit'.
Post-conditions: explanations of the given data are obtained.
Main output: explanations for the given samples are saved in results_path/explanations_(time).

Parameters:

- **x** (*ndarray (n_samples*n_features)*) – a matrix of samples for which explanation is needed.
- **ids** (*ndarray (n_samples,)*) – an array containing the ids of the samples in x.
- **y** (*ndarray (n_samples,), optional.*) – an array containing the true labels for the samples in x. Used only if mode == 'error'. Default None.
- **mode** (*str, optional.*) – one between 'all_classes', 'majority_class' and 'error'. If 'all_classes', neighbors are chosen between elements of all classes except for the predicted one of the current sample. If 'majority_class', neighbors are chosen between the elements of the majority class (except for the predicted one) of the current sample. If 'error', the function is equal to 'majority_class' with the difference that the correct label (specified in y) is considered as secondary class. Default 'all_classes'.
- **q** (*int (>0), optional.*) – number of neighbors considered for each sample. For each sample, q nearest neighbors belonging to the majority class (excluded the predicted one) are considered and their explanations are computed. Default
- **k** (*int (>0), optional.*) – number of points considered in the feature interval. At each iteration, k points are taken from the interval [min(sample_feature, neighbor_feature), max(sample_feature, neighbor_feature)]. Default

- **p** (*float (in [0, 1]), optional.*) – percentage of features considered (a number in [0, 1]). The algorithm stops after examining the p*n_features most important features. Default 0.04.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the NAWAXOVO architecture on the given training set.

Pre-conditions: :py:meth:'NAWAX.__init__'.
Post-conditions: the NAWAXOVO object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the architecture is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples.

Pre-conditions: :py:meth:'NAWAX.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples*n_features)*) – a matrix containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

processing_history_predict(*x*, *y*, *ids*)

This function computes the predictions for the given samples and store information about the alterations experienced by the predictions themselves.

Pre-conditions: :py:meth:'NAWAX.fit'.
Post-conditions: predictions of the given data are obtained and info are stored in a file.
Main output: labels predicted for the given data.

Parameters:
- **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.
- **ids** (*ndarray (n_samples,)*) – an array containing the ids of the given samples.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.NAWAX.OVABaseLayer module

***class* src.NAWAX.OVABaseLayer.OVABaseLayer(*n_classes, subject_id, results_path, models_path, samples_balancing*)**

Bases: `object`

This class implements the base layer for the NAWAXOVA architecture. Base classifiers are trained in a 1 vs ALL manner.

This function initializes the OVABaseLayer (NAWAX version) object.

Pre-conditions: none.
Post-conditions: a new OVABaseLayer (NAWAX version) object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.
- **samples_balancing** (*str.*) – balancing strategy adopted in case of non-balanced sub-problem. One between 'none', 'random', 'clustering', 'smote', 'nearmiss', 'stratified_random'.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the 1 vs ALL classifiers of the base layer on the given training set.

Pre-conditions: :py:meth:'OVABaseLayer.__init__'.
Post-conditions: the OVABaseLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray.*) – a (num_samples*num_features) matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray.*) – a num_samples array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, n_classes) matrix representing the probabilities for each class.

Pre-conditions: :py:meth:'OVABaseLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray.*) – a (n_samples*n_features) matrix containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples ((n_samples, n_classes) array)

Return type: ndarray.

Raise: none.

src.NAWAX.OVOBaseLayer module

***class* src.NAWAX.OVOBaseLayer.OVOBaseLayer(*n_classes*, *subject_id*, *results_path*, *models_path*, *samples_balancing*)**

Bases: `object`

This class implements the base layer for the NAWAXOVO architecture. Base classifiers are trained in a 1 vs 1 manner.

This function initializes the OVOBaseLayer (NAWAXOVO version) object.

Pre-conditions: none.
Post-conditions: a new OVOBaseLayer (NAWAXOVO version) object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.
- **samples_balancing** (*str.*) – balancing strategy adopted in case of non-balanced sub-problem. One between 'none', 'random', 'clustering', 'smote', 'nearmiss'.

Returns: none.

Raise: none.

explain(*x, class1, class2*)

This function computes the explanations of the samples in x from the 1 vs 1 classifier class1[sample] vs class2[sample]. Explanations are obtained as SHAP values of the class with the lowest index between class1[sample] and class2[sample]: if class2[sample] < class1[sample], explanations are changed in sign.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: explanations of the given data are obtained from 1 vs 1 base classifiers.
Main output: explanations for the given data.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix containing the samples for which explanations are required.
- **class1** (*ndarray (n_samples,)*) – a vector containing the primary class for each sample.
- **class2** (*ndarray (n_samples,)*) – a vector containing the secondary class for each sample.

Returns: explanations of samples in x according to classifier class1[sample] vs class2[sample].

Return type: ndarray (n_samples*n_features)

Raise: none.

fit(*x, y*)

This function fits the 1 vs 1 classifiers of the base layer on the given training set.

Pre-conditions: :py:meth:'OVOBaseLayer.__init__'.
Post-conditions: the OVOBaseLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, 2*n_classifiers) matrix representing the two output probabilities of each classifier.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

processing_history_predict(*x, filename*)

This function computes the predictions for the given samples and store information about the alterations experienced by the predictions themselves.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: predictions of the given data are obtained and info are stored in a file.
Main output: labels predicted for the given data.

Parameters:
- **x** (*ndarray (n_samples, n_features)*) – a matrix containing the features of each sample for which prediction is needed.
- **filename** (*str.*) – name of the file in which info are stored.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.NTAWAX namespace

Submodules

src.NTAWAX.NTAWAX module

***class* src.NTAWAX.NTAWAX.NTAWAX(*n_classes, subject_id, models_path='models/', results_path='results/', n_neighbors=5, base_layer_mode='1vs1', samples_balancing='random'*)**

Bases: `sklearn.base.BaseEstimator`

This class represents the skeleton for NTAWAXOVO and NTAWAXOVA architectures.

This function initialises the NTAWAX object using the given parameters. This will be the skeleton of NTAWAXOVO/NTAWAXOVA architecture.

Pre-conditions: none.
Post-conditions: a new NTAWAXOVO/NTAWAXOVA object is created.
Main output: none.

Parameters:

- **n_classes** (*int.*) – number of classes of the given problem.
- **subject_id** (*int.*) – id of the considered subject.
- **models_path** (*str, optional.*) – path in which models are saved. Default ‘models/’.
- **results_path** (*str, optional.*) – path in which results are saved. Default ‘results/’.
- **n_neighbors** (*int (>0)*) – number of neighbors used by NTAWAXOVO/NTAWAXOVA’s KNNs.
- **base_layer_mode** (*str, optional.*) – one between ‘1vs1’ and ‘1vsALL’. If ‘1vs1’, base classifiers are trained in a 1 vs 1 manner (NTAWAXOVO). If ‘1vsALL’, base classifiers are trained in a 1 vs ALL manner (NTAWAXOVA). Default ‘1vs1’.
- **samples_balancing** (*str, optional.*) – balancing strategy adopted in case of non-balanced sub-problem. One between ‘none’, ‘random’, ‘stratified_random’, ‘clustering’, ‘smote’, ‘nearmiss’. In case of ‘1vs1’, ‘stratified_random’ and ‘random’ are the same (use ‘random’). Default ‘random’.

Returns: none.

Raise: none

explain(*x, ids, y=None, mode='all_classes', q=3, k=5, p=0.04*)

This function computes the explanations related to the samples x. For each sample, q nearest neighbors belonging to the majority class (excluded the predicted one)("majority_class"), to all classes (excluded the predicted one) ("all_classes") or to the correct class ("error") are considered, and their explanations are computed. The most similar sample (in the explanations space) is taken as neighbor. Features are ordered by decreasing importance for the classification of the current sample, and the less important are discarded (according to the p parameter). For each feature, k points are taken from the interval [min(sample_feature, neighbor_feature), max(sample_feature, neighbor_feature)]. Cartesian product with the previously extracted points is computed, new artificial points are created, and their predictions are finally obtained. The algorithm stop when the prediction of an artificial point matches the class of the neighbor (or when the most important features have been analyzed).

Pre-conditions: :py:meth:'NTAWAX.fit'.
Post-conditions: explanations of the given data are obtained.
Main output: explanations for the given samples are saved in results_path/explanations_(time).

Parameters:

- **x** (*ndarray (n_samples*n_features)*) – a matrix of samples for which explanation is needed.
- **ids** (*ndarray (n_samples,)*) – an array containing the ids of the samples in x.
- **y** (*ndarray (n_samples,), optional.*) – an array containing the true labels for the samples in x. Used only if mode == 'error'. Default None.
- **mode** (*str, optional.*) – one between 'all_classes', 'majority_class' and 'error'. If 'all_classes', neighbors are chosen between elements of all classes except for the predicted one of the current sample. If 'majority_class', neighbors are chosen between the elements of the majority class (except for the predicted one) of the current sample. If 'error', the function is equal to 'majority_class' with the difference that the correct label (specified in y) is considered as secondary class. Default 'all_classes'.
- **q** (*int (>0), optional.*) – number of neighbors considered for each sample. For each sample, q nearest neighbors belonging to the majority class (excluded the predicted one) are considered and their explanations are computed. Default
- **k** (*int (>0), optional.*) – number of points considered in the feature interval. At each iteration, k points are taken from the interval [min(sample_feature, neighbor_feature), max(sample_feature, neighbor_feature)]. Default

- **p** (*float (in [0, 1]), optional.*) – percentage of features considered (a number in [0, 1]). The algorithm stops after examining the p*n_features most important features. Default

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the NTAWAXOVO/NTAWAXOVA architecture on the given training set.

Pre-conditions: :py:meth:'NTAWAX.__init__'.
Post-conditions: the NTAWAXOVO/NTAWAXOVA object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the architecture is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples.

Pre-conditions: :py:meth:'NTAWAX.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples*n_features)*) – a matrix containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

processing_history_predict(*x*, *y*, *ids*)

This function computes the predictions for the given samples and store information about the alterations experienced by the predictions themselves.

Pre-conditions: :py:meth:'NTAWAX.fit'.
Post-conditions: predictions of the given data are obtained and info are stored in a file.
Main output: labels predicted for the given data.

Parameters:
- **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.
- **ids** (*ndarray (n_samples,)*) – an array containing the ids of the given samples.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.NTAWAX.OVABaseLayer module

***class* src.NTAWAX.OVABaseLayer.OVABaseLayer(*n_classes, subject_id, results_path, models_path, samples_balancing*)**

Bases: `object`

This class implements the base layer for the NTAWAXOVA architecture. Base classifiers are trained in a 1 vs ALL manner.

This function initializes the OVABaseLayer (NTAWAX version) object.

Pre-conditions: none.
Post-conditions: a new OVABaseLayer (NTAWAX version) object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.
- **samples_balancing** (*str.*) – balancing strategy adopted in case of non-balanced sub-problem. One between 'none', 'random', 'clustering', 'smote', 'nearmiss', 'stratified_random'.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the 1 vs ALL classifiers of the base layer on the given training set.

Pre-conditions: :py:meth:'OVABaseLayer.__init__'.
Post-conditions: the OVABaseLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray.*) – a (num_samples*num_features) matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray.*) – a num_samples array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, n_classes) matrix representing the probabilities for each class.

Pre-conditions: :py:meth:'OVABaseLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray.*) – a (n_samples*n_features) matrix containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples ((n_samples, n_classes) array)

Return type: ndarray.

Raise: none.

src.NTAWAX.OVOBaseLayer module

***class* src.NTAWAX.OVOBaseLayer.OVOBaseLayer(*n_classes*, *subject_id*, *results_path*, *models_path*, *samples_balancing*)**

Bases: `object`

This class implements the base layer for the NTAWAXOVO architecture. Base classifiers are trained in a 1 vs 1 manner.

This function initializes the OVOBaseLayer (NTAWAX version) object.

Pre-conditions: none.
Post-conditions: a new OVOBaseLayer (NTAWAX version) object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.
- **samples_balancing** (*str.*) – balancing strategy adopted in case of non-balanced sub-problem. One between 'none', 'random', 'clustering', 'smote', 'nearmiss'.

Returns: none.

Raise: none.

explain(*x, class1, class2*)

This function computes the explanations of the samples in x from the 1 vs 1 classifier class1[sample] vs class2[sample]. Explanations are obtained as SHAP values of the class with the lowest index between class1[sample] and class2[sample]: if class2[sample] < class1[sample], explanations are changed in sign.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: explanations of the given data are obtained from 1 vs 1 base classifiers.
Main output: explanations for the given data.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix containing the samples for which explanations are required.
- **class1** (*ndarray (n_samples,)*) – a vector containing the primary class for each sample.
- **class2** (*ndarray (n_samples,)*) – a vector containing the secondary class for each sample.

Returns: explanations of samples in x according to classifier class1[sample] vs class2[sample].

Return type: ndarray (n_samples*n_features)

Raise: none.

fit(*x, y*)

This function fits the 1 vs 1 classifiers of the base layer on the given training set.

Pre-conditions: :py:meth:'OVOBaseLayer.__init__'.
Post-conditions: the OVOBaseLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, n_classes) matrix representing the sum of the probabilities for each class.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a matrix containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

processing_history_predict(*x, filename*)

This function computes the predictions for the given samples and store information about the alterations experienced by the predictions themselves.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: predictions of the given data are obtained and info are stored in a file.
Main output: labels predicted for the given data.

Parameters:
- **x** (*ndarray (n_samples, n_features)*) – a matrix containing the features of each sample for which prediction is needed.
- **filename** (*str.*) – name of the file in which info are stored.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.NTAWAXBIS namespace

Submodules

src.NTAWAXBIS.NAWAXBIS module

***class* src.NTAWAXBIS.NAWAXBIS.NTAWAXBIS(*n_classes*, *subject_id*, *models_path='models/'*, *results_path='results/'*, *n_neighbors=5*, *base_layer_mode='1vs1'*, *samples_balancing='random'*)**

Bases: `sklearn.base.BaseEstimator`

This class represents the skeleton for NTAWAXOVOBIS architecture. The difference with respect to the original NTAWAXOVO architecture is that now KNN weights are applied before the summation, and they are computed in a different way (weight of each one vs. one classifier is the average between the results of KNN's predict_proba of the classes the classifier is in charge of).

This function initialises the NTAWAXBIS object using the given parameters. This will be the skeleton of NTAWAXOVOBIS architecture.

Pre-conditions: none.
Post-conditions: a new NTAWAXOVOBIS object is created.
Main output: none.

Parameters:

- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the considered subject.
- **models_path** (*str, optional.*) – path in which models are saved. Default 'models/'.
- **results_path** (*str, optional.*) – path in which results are saved. Default 'results/'.
- **n_neighbors** (*int (>0)*) – number of neighbors used by NTAWAXOVOBIS's KNNs.
- **base_layer_mode** (*str, optional.*) – one between '1vs1' and '1vsALL'. If '1vs1', base classifiers are trained in a 1 vs 1 manner (NTAWAXOVO). If '1vsALL', base classifiers are trained in a 1 vs ALL manner (NTAWAXOVA). Default '1vs1'.
- **samples_balancing** (*str, optional.*) – balancing strategy adopted in case of non-balanced sub-problem. One between 'none', 'random', 'stratified_random', 'clustering', 'smote', 'nearmiss'. In case of '1vs1', 'stratified_random' and 'random' are the same (use 'random'). Default 'random'.

Returns: none.

Raise: none.

explain(*x, ids, y=None, mode='all_classes', q=3, k=5, p=0.04*)

This function computes the explanations related to the samples x. For each sample, q nearest neighbors belonging to the majority class (excluded the predicted one)("majority_class"), to all classes (excluded the predicted one) ("all_classes") or to the correct class ("error") are considered, and their explanations are computed. The most similar sample (in the explanations space) is taken as neighbor. Features are ordered by decreasing importance for the classification of the current sample, and the less important are discarded (according to the p parameter). For each feature, k points are taken from the interval [min(sample_feature, neighbor_feature), max(sample_feature, neighbor_feature)]. Cartesian product with the previously extracted points is computed, new artificial points are created, and their predictions are finally obtained. The algorithm stop when the prediction of an artificial point matches the class of the neighbor (or when the most important features have been analyzed).

Pre-conditions: :py:meth:'NTAWAXBIS.fit'.
Post-conditions: explanations of the given data are obtained.
Main output: explanations for the given samples are saved in results_path/explanations_(time).

Parameters:

- **x** (*ndarray (n_samples*n_features)*) – a matrix of samples for which explanation is needed.
- **ids** (*ndarray (n_samples,)*) – an array containing the ids of the samples in x.
- **y** (*ndarray (n_samples,), optional.*) – an array containing the true labels for the samples in x. Used only if mode == 'error'. Default None.
- **mode** (*str, optional.*) – one between 'all_classes', 'majority_class' and 'error'. If 'all_classes', neighbors are chosen between elements of all classes except for the predicted one of the current sample. If 'majority_class', neighbors are chosen between the elements of the majority class (except for the predicted one) of the current sample. If 'error', the function is equal to 'majority_class' with the difference that the correct label (specified in y) is considered as secondary class. Default 'all_classes'.
- **q** (*int (>0), optional.*) – number of neighbors considered for each sample. For each sample, q nearest neighbors belonging to the majority class (excluded the predicted one) are considered and their explanations are computed. Default

- **k** (*int (>0), optional.*) – number of points considered in the feature interval. At each iteration, k points are taken from the interval [min(sample_feature, neighbor_feature), max(sample_feature, neighbor_feature)]. Default
- **p** (*float (in [0, 1]), optional.*) – percentage of features considered (a number in [0, 1]). The algorithm stops after examining the p*n_features most important features.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the NTAWAXOVOBIS architecture on the given training set.

Pre-conditions: :py:meth:'NTAWAXBIS.__init__'.
Post-conditions: the NTAWAXOVOBIS object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the architecture is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples.

Pre-conditions: :py:meth:'NTAWAXBIS.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples*n_features)*) – a matrix containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

processing_history_predict(*x*, *y*, *ids*)

This function computes the predictions for the given samples and store information about the alterations experienced by the predictions themselves.

Pre-conditions: :py:meth:'NTAWAXBIS.fit'.
Post-conditions: predictions of the given data are obtained and info are stored in a file.
Main output: labels predicted for the given data.

Parameters:
- **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.
- **ids** (*ndarray (n_samples,)*) – an array containing the ids of the given samples.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.NTAWAXBIS.OVOBaseLayer module

***class* src.NTAWAXBIS.OVOBaseLayer.OVOBaseLayer(*n_classes, subject_id, results_path, models_path, samples_balancing*)**

Bases: `object`

This class implements the base layer for the NTAWAXOVOBIS architecture. Base classifiers are trained in a 1 vs 1 manner.

This function initializes the OVOBaseLayer (NTAWAXBIS version) object.

Pre-conditions: none.
Post-conditions: a new OVOBaseLayer (NTAWAXBIS version) object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (>0)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.
- **samples_balancing** (*str.*) – balancing strategy adopted in case of non-balanced sub-problem. One between 'none', 'random', 'clustering', 'smote', 'nearmiss'.

Returns: none.

Raise: none.

explain(*x, class1, class2*)

This function computes the explanations of the samples in x from the 1 vs 1 classifier class1[sample] vs class2[sample]. Explanations are obtained as SHAP values of the class with the lowest index between class1[sample] and class2[sample]: if class2[sample] < class1[sample], explanations are changed in sign.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: explanations of the given data are obtained from 1 vs 1 base classifiers.
Main output: explanations for the given data.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix containing the samples for which explanations are required.
- **class1** (*ndarray (n_samples,)*) – a vector containing the primary class for each sample.
- **class2** (*ndarray (n_samples,)*) – a vector containing the secondary class for each sample.

Returns: explanations of samples in x according to classifier class1[sample] vs class2[sample].

Return type: ndarray (n_samples*n_features)

Raise: none.

fit(*x, y*)

This function fits the 1 vs 1 classifiers of the base layer on the given training set.

Pre-conditions: :py:meth:'OVOBaseLayer.__init__'.
Post-conditions: the OVOBaseLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, 2*n_classifiers) matrix representing the two output probabilities of each classifier.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

processing_history_predict(*x, filename*)

This function computes the predictions for the given samples and store information about the alterations experienced by the predictions themselves.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: predictions of the given data are obtained and info are stored in a file.
Main output: labels predicted for the given data.

Parameters:

- **x** (*ndarray (n_samples, n_features)*) – a matrix containing the features of each sample for which prediction is needed.
- **filename** (*str.*) – name of the file in which info are stored.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.TAWAX namespace

Submodules

src.TAWAX.OVABaseLayer module

***class* src.TAWAX.OVABaseLayer.OVABaseLayer(*n_classes*, *subject_id*, *models_path*, *results_path*, *samples_balancing*)**

Bases: `object`

This class implements the base layer for the NAWAXOVA architecture. Base classifiers are trained in a 1 vs ALL manner.

This function initializes the OVABaseLayer (TAWAX version) object.

Pre-conditions: none.
Post-conditions: a new OVABaseLayer (TAWAX version) object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.
- **samples_balancing** (*str.*) – balancing strategy adopted in case of non-balanced sub-problem. One between 'none', 'random', 'clustering', 'smote', 'nearmiss', 'stratified_random'.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the 1 vs ALL classifiers of the base layer on the given training set.

Pre-conditions: :py:meth:'OVABaseLayer.__init__'.
Post-conditions: the OVABaseLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples, n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, n_classes) matrix representing the probabilities for each class.

Pre-conditions: :py:meth:'OVABaseLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a matrix containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.TAWAX.OVOBaseLayer module

***class* src.TAWAX.OVOBaseLayer.OVOBaseLayer(*n_classes, subject_id, models_path, results_path, samples_balancing*)**

Bases: `object`

This class implements the base layer for the TAWAXOVO architecture. Base classifiers are trained in a 1 vs 1 manner.

This function initializes the OVOBaseLayer (TAWAX version) object.

Pre-conditions: none.
Post-conditions: a new OVOBaseLayer (TAWAX version) object is created.
Main output: none.

Parameters:

- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.
- **samples_balancing** (*str.*) – balancing strategy adopted in case of non-balanced sub-problem. One between 'none', 'random', 'clustering', 'smote', 'nearmiss'.

Returns: none

Returns: none.

Raise: none.

fit(*x, y*)

This function fits the 1 vs 1 classifiers of the base layer on the given training set.

Pre-conditions: :py:meth:'OVOBaseLayer.__init__'.
Post-conditions: the OVOBaseLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, n_classes) matrix representing the sum of the probabilities for each class.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

processing_history_predict(*x, filename*)

This function computes the predictions for the given samples and store information about the alterations experienced by the predictions themselves.

Pre-conditions: :py:meth:'OVOBaseLayer.fit'.
Post-conditions: predictions of the given data are obtained and info are stored in a file.
Main output: labels predicted for the given data.

Parameters:
- **x** (*ndarray (n_samples, n_features)*) – a matrix containing the features of each sample for which prediction is needed.
- **filename** (*str.*) – name of the file in which info are stored.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.TAWAX.TAWAX module

***class* src.TAWAX.TAWAX.TAWAX(*n_classes, subject_id, models_path='models/', results_path='results/', base_layer_mode='1vs1', samples_balancing='none'*)**

Bases: `object`

This class represents the skeleton for TAWAXOVO and TAWAXOVA architectures.

This function initialises the TAWAX object using the given parameters. This will be the skeleton of TAWAXOVO/TAWAXOVA architecture.

Pre-conditions: none.
Post-conditions: a new TAWAXOVO/TAWAXOVA object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the considered subject.
- **models_path** (*str, optional.*) – path in which models are saved. Default 'models/'.
- **results_path** – path in which results are saved. Default 'results/'.
- **base_layer_mode** (*str, optional.*) – one between '1vs1' and '1vsALL'. If '1vs1', base classifiers are trained in a 1 vs 1 manner (TAWAXOVO). If '1vsALL', base classifiers are trained in a 1 vs ALL manner (TAWAXOVA). Default '1vs1'.
- **samples_balancing** (*str, optional.*) – balancing strategy adopted in case of non-balanced sub-problem. One between 'none', 'random', 'stratified_random', 'clustering', 'smote', 'nearmiss'. In case of '1vs1', 'stratified_random' and 'random' are the same (use 'random'). Default 'random'.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the TAWAXOVO/TAWAXOVA architecture on the given training set.

Pre-conditions: :py:meth:'TAWAX.__init__'.
Post-conditions: the TAWAXOVO/TAWAXOVA object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the architecture is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples.

Pre-conditions: :py:meth:'TAWAX.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples*n_features)*) – a matrix containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

processing_history_predict(*x*, *y*, *ids*)

This function computes the predictions for the given samples and store information about the alterations experienced by the predictions themselves.

Pre-conditions: :py:meth:'TAWAX.fit'.
Post-conditions: predictions of the given data are obtained and info are stored in a file.
Main output: labels predicted for the given data.

Parameters:
- **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.
- **ids** (*ndarray (n_samples,)*) – an array containing the ids of the given samples.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.TAWAX.TAWAXSuperClassifierLayer module

***class* src.TAWAX.TAWAXSuperClassifierLayer.TAWAXSuperClassifierLayer(*n_classes, subject_id, models_path, results_path, classifier*)**

Bases: `object`

This class implements the super classifier layer for the TAWAXOVA architecture. Super classifier is trained in a ALL vs ALL manner.

This function initializes the TAWAXSuperClassifierLayer object.

Pre-conditions: none.
Post-conditions: a new TAWAXSuperClassifierLayer object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.

Returns: none.

Raise: none.

fit(*x, y*)

This function fits the super classifier on the given training set.

Pre-conditions: :py:meth:'TAWAXSuperClassifierLayer.__init__'.
Post-conditions: the TAWAXSuperClassifierLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples,) array representing the final predictions.

Pre-conditions: :py:meth:'TAWAXSuperClassifierLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

src.Troika namespace

Submodules

src.Troika.Troika module

***class* src.Troika.Troika.Troika(*n_classes=4, subject_id=1, models_path='models/', results_path='results/', folds=5, n_base_classifiers=5*)**

Bases: `object`

This class implements Troika architecture, described in https://doi.org/10.1016/j.ins.2009.08.025.

This function initialises the Troika object using the given parameters. This will be the skeleton of Troika architecture.

Pre-conditions: none.
Post-conditions: a new Troika object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the considered subject.
- **models_path** (*str, optional.*) – path in which models are saved. Default 'models/'.
- **results_path** (*str, optional.*) – path in which results are saved. Default 'results/'.
- **folds** (*int (>0), optional.*) – number of folds considered in the cross-validation approach for generating the dataset for the following layer. Default 5.
- **n_base_classifiers** (*int (>0), optional.*) – number of base classifiers used in the architecture. Default 5.

Returns: none.

Raise: none.

fit(*x, y*)

This function fits the Troika architecture on the given training set.

Pre-conditions: :py:meth:'Troika.__init__'.
Post-conditions: the Troika object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the architecture is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples.

Pre-conditions: :py:meth:'Troika.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples*n_features)*) – a matrix containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

src.Troika.TroikaBaseClassifiersLayer module

***class* src.Troika.TroikaBaseClassifiersLayer.TroikaBaseClassifiersLayer(*n_classes, subject_id, models_path, results_path, n_base_classifiers*)**

Bases: `object`

This class implements the base classifiers layer for the Troika architecture. Base classifiers are trained in a ALL vs ALL manner.

This function initializes the TroikaBaseClassifiersLayer object.

Pre-conditions: none.
Post-conditions: a new TroikaBaseClassifiersLayer object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.
- **n_base_classifiers** (*int (>0)*) – number of base classifiers used in the architecture.

Returns: none.

Raise: none.

fit(*x, y*)

This function fits the meta classifiers on the given training set.

Pre-conditions: :py:meth:'TroikaBaseClassifiersLayer.__init__'.
Post-conditions: the TroikaBaseClassifiersLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, n_base_classifiers*n_classes) array representing predictions of the base classifiers. Considering a single sample, the first n_classes values represent the predictions of the first base classifier, the following n_classes values represent the predictions of the second base classifier, and so on.

Pre-conditions: :py:meth:'TroikaBaseClassifiersLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

src.Troika.TroikaMetaClassifiersLayer module

***class* src.Troika.TroikaMetaClassifiersLayer.TroikaMetaClassifiersLayer(*n_classes, models_path, subject_id, results_path*)**

Bases: `object`

This class implements the meta layer for the Troika architecture. Meta classifiers are trained in a 1 vs ALL manner.

This function initializes the TroikaMetaClassifiersLayer object.

Pre-conditions: none.
Post-conditions: a new TroikaMetaClassifiersLayer object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits the meta classifiers on the given training set.

Pre-conditions: :py:meth:'TroikaMetaClassifiersLayer.__init__'.
Post-conditions: the TroikaMetaClassifiersLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, n_classes) matrix representing the probabilities for each class.

Pre-conditions: :py:meth:'TroikaMetaClassifiersLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: predictions (probabilities) for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_classes)

Raise: none.

src.Troika.TroikaSpecialistClassifiersLayer module

***class* src.Troika.TroikaSpecialistClassifiersLayer.TroikaSpecialistClassifiersLayer(*n_classes, subject_id, n_base_classifiers, models_path, results_path*)**

Bases: `object`

This class implements the specialist layer for the Troika architecture. Specialist classifiers are trained in a 1 vs 1 manner.

This function initializes the TroikaSpecialistClassifiersLayer object.

Pre-conditions: none.
Post-conditions: a new TroikaSpecialistClassifiersLayer object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **n_base_classifiers** (*int (>0)*) – number of base classifiers used in the architecture.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits specialist classifiers on the given training set.

Pre-conditions: :py:meth:'TroikaSpecialistClassifiersLayer.__init__'.
Post-conditions: the TroikaSpecialistClassifiersLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples, n_specialist) matrix representing the probabilities from each specialist (probabilities refer to the class with lowest index).

Pre-conditions: :py:meth:'TroikaSpecialistClassifiersLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: predictions (probabilities) for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples, n_specialist)

Raise: none.

src.Troika.TroikaSuperClassifierLayer module

***class* src.Troika.TroikaSuperClassifierLayer.TroikaSuperClassifierLayer(*n_classes, subject_id, models_path, results_path*)**

Bases: `object`

This class implements the super classifier layer for the Troika architecture. Super classifier is trained in a ALL vs ALL manner.

This function initializes the TroikaSuperClassifierLayer object.

Pre-conditions: none.
Post-conditions: a new TroikaSuperClassifierLayer object is created.
Main output: none.

Parameters:
- **n_classes** (*int (>0)*) – number of classes of the given problem.
- **subject_id** (*int (between 1 and 9)*) – id of the subject considered.
- **models_path** (*str.*) – path to directory in which models are saved.
- **results_path** (*str.*) – path to directory in which models are saved.

Returns: none.

Raise: none.

fit(*x*, *y*)

This function fits super classifier on the given training set.

Pre-conditions: :py:meth:'TroikaSuperClassifierLayer.__init__'.
Post-conditions: the TroikaSuperClassifierLayer object is trained on the given data.
Main output: none.

Parameters:
- **x** (*ndarray (n_samples*n_features)*) – a matrix representing the features of each sample on which the layer is trained.
- **y** (*ndarray (n_samples,)*) – an array of labels corresponding to the given features.

Returns: self.

Return type: estimator.

Raise: none.

predict(*x*)

This function computes the predictions for the given samples. Predictions are represented by a (n_samples,) array representing the final predictions.

Pre-conditions: :py:meth:'TroikaSuperClassifierLayer.fit'.
Post-conditions: predictions of the given data are obtained.
Main output: labels predicted for the given data.

Parameters: **x** (*ndarray (n_samples, n_features)*) – a containing the features of each sample for which prediction is needed.

Returns: the predictions for the given samples.

Return type: ndarray (n_samples,)

Raise: none.

src.UCIUtils namespace

Submodules

src.UCIUtils.UCIUtils module

class **src.UCIUtils.UCIUtils.UCIDatasetLoader**

Bases: `object`

This class implements some static methods useful to read UCI datasets from archive repository.

> ***classmethod*** **anneal(*test_split*)**
>
> This function reads the dataset *Anneal* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels. In this dataset, class 4 is empty.
>
> Pre-conditions: none.
> Post-conditions: training and test set obtained from dataset *Anneal* are returned.
> Main output: training features, test_features, training labels, test_labels, n_classes from *Anneal* dataset.
>
> **Parameters:** **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.
>
> **Returns:** training features, test_features, training labels, test_labels, n_classes.
>
> **Return type:** ndarray, ndarray, ndarray, ndarray, int.
>
> **Raise:** none.

> ***classmethod*** **autos(*test_split*)**

This function reads the dataset *Autos* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Autos* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Autos* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **car(*test_split*)**

This function reads the dataset *Car* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Car* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Car* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **cleveland(*test_split*)**

This function reads the dataset *Cleveland* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Cleveland* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Cleveland* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

***classmethod* ecoli(*test_split*)**

This function reads the dataset *Ecoli* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Ecoli* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Ecoli* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

***classmethod* flag(*test_split*)**

This function reads the dataset *flag* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *flag* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *flag* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **glass(*test_split*)**

This function reads the dataset *Glass* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Glass* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Glass* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **hepatitis_domain(*test_split*)**

This function reads the dataset *Hepatitis* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels. Contains missing values, denoted by '?'.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Hepatitis* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Hepatitis* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **iris(*test_split*)**

This function reads the dataset *Iris* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Iris* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Iris* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **krkopt(*test_split*)**

This function reads the dataset *Krkopt* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Krkopt* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Krkopt* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **letter(*test_split*)**

This function reads the dataset *Letter* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Letter* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Letter* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **pageblocks(*test_split*)**

This function reads the dataset *Pageblocks* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Pageblocks* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Pageblocks* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **penbased(*test_split*)**

This function reads the dataset *Penbased* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Penbased* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Penbased* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **print_class_distribution(*labels*)**

This function returns the distribution of the classes of the dataset whose labels are passed in *labels*.

Pre-conditions: none.
Post-conditions: distribution of the dataset is obtained.
Main output: dictionary containing class labels and their number of occurrences.

Parameters: **labels** (*ndarray (n_samples,)*) – an array representing the labels of all the samples of the dataset.

Returns: none,

Raise: none.

classmethod **satimage(*test_split*)**

This function reads the dataset *Satimage* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Satimage* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Satimage* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **segment(*test_split*)**

This function reads the dataset *Segment* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Segment* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Segment* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **shuttle(*test_split*)**

This function reads the dataset *Shuttle* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Shuttle* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Shuttle* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **splice(*test_split*)**

This function reads the dataset *Splice* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Splice* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Splice* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **uci_dataset_handler(*dataset_name*, *test_split*)**

This function reads the dataset *dataset_name* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Vehicle* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *dataset_name* dataset.

Parameters:
- **dataset_name** (str. One between *anneal*, *autos*, *car*, *cleveland*, *ecoli*, *flag*, *glass*, *hepatitis*, *iris*, *krkopt*, *letter*, *pageblocks*, *penbased*, *satimage*, *shuttle*, *segment*, *splice*, *vehicle*, *vowel*, *waveform*, *yeast*, *zoo*.) – name of the dataset.
- **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

classmethod **vehicle(*test_split*)**

This function reads the dataset *Vehicle* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Vehicle* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Vehicle* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

***classmethod* vowel(*test_split*)**

This function reads the dataset *Vowel* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Vowel* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Vowel* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

***classmethod* waveform(*test_split*)**

This function reads the dataset *Waveform* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Waveform* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Waveform* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

***classmethod* yeast(*test_split*)**

This function reads the dataset *Yeast* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Yeast* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Yeast* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

***classmethod* zoo(*test_split*)**

This function reads the dataset *Zoo* from the repository and returns a training set and test set accordingly to the proportion specified in *test_split*. Training and test set are split in a stratified random way. Categorical features (if present) are converted in numerical features. Class labels are converted in numerical labels.

Pre-conditions: none.
Post-conditions: training and test set obtained from dataset *Zoo* are returned.
Main output: training features, test_features, training labels, test_labels, n_classes from *Zoo* dataset.

Parameters: **test_split** (*float (in [0, 1])*) – Percentage of dataset considered as test set.

Returns: training features, test_features, training labels, test_labels, n_classes.

Return type: ndarray, ndarray, ndarray, ndarray, int.

Raise: none.

src.Utils namespace

Submodules

src.Utils.ClassifiersManager module

class **src.Utils.ClassifiersManager.ClassifiersManager**

Bases: `object`

This class stores optimal hyperparameters configurations and returns estimator objects having that configurations.

> ***classmethod*** **puk_kernel(*x1, x2, sigma=1.0, omega=1.0*)**
>
> This function computes the kernel matrix between two arrays using the Pearson VII function-based universal kernel. It is taken from https://github.com/rlphilli/sklearn-PUK-kernel/blob/master/PUK_kernel.py. For more info, see https://doi.org/10.1016/j.chemolab.2005.09.003. Used as kernel function in SVC classifiers.
>
> Pre-conditions: none.
> Post-conditions: puk kernel is computed using the given data and parameters.
> Main output: kernel matrix.
>
> **Parameters:**
> - **x1** (*ndarray.*) – first data matrix
> - **x2** (*ndarray.*) – second data matrix.
> - **sigma** (*float.*) – sigma parameter.
> - **omega** (*float.*) – omega parameter.
>
> **Returns:** kernel matrix.
>
> **Return type:** ndarray (n_samples, n_samples)
>
> **Raise:** none.

> ***classmethod*** **return_classifier(*configuration_name*)**

This function returns a Sklearn classifier whose name is passed as *configuration_name*. Results of hyperparameters optimization problems are saved into a dictionary and can be retrieved using their names.

Pre-conditions: none.
Post-conditions: a new classifier is returned.
Main output: classifier.

Parameters: **configuration_name** (*str.*) – name of the required configuration.

Returns: classifier whose name correspond to *configuration_name*.

Return type: estimator.

Raise: none.

src.Utils.DBScan module

class **src.Utils.DBScan.DBScan(*min_pts, dataset*)**

Bases: `object`

This class implements the DBScan algorithm. In particular, neighborhood function is designed to work with 22 electrodes placed as described in https://doi.org/10.3389/fnins.2012.00055.

This function initialises the DBScan object using the given parameters.

Pre-conditions: none.
Post-conditions: a new DBScan object is created.
Main output: none.

Parameters:
- **min_pts** (*int (>0)*) – minimum number of neighbors that a point must have in order not to be considered as outliers.
- **dataset** (*ndarray (n_elements,)*) – features dataset. Each element is the name of a features, such as *theta14* or *gamma11*.

Returns: none.

Raise: none.

dbscan()

This function implements the DBScan algorithm.

Pre-conditions: :py:meth:'DBScan.__init__'..
Post-conditions: clusters of close features are obtained, according to the DBScan algorithm.
Main output: none. Clusters are saved in *self.clusters*.

Returns: self.

Return type: DBScan.

Raise: none.

get_clusters()

This function returns a string containing cluster indexes and names formatted in a format suitable for printing.

Pre-conditions: :py:meth:'DBScan.__init__'..
Post-conditions: cluster info are obtained.
Main output: a string containing cluster information.

Returns: none.

Raise: none.

get_n_clusters()

This function return the number of clusters found by the algorithm.

Pre-conditions: :py:meth:'DBScan.__init__'.
Post-conditions: clusters number is obtained. If called before :py:meth:'DBScan.DBScan', it will return 0.
Main output: an int representing the number of clusters created.

Returns: n_clusters.

Return type: int.

Raise: none.

src.Utils.DatasetUtils module

class src.Utils.DatasetUtils.DatasetUtils

Bases: `object`

This class implements some static functions useful for dealing with EEG datasets.

static add_column_to_csv(*filepath*, *columns_names*, *columns_values*)

This function add columns *column_names* of values *columns_values* to the file specified in *filepath*.

Pre-conditions: none.
Post-conditions: new columns are added to the file *filepath*.
Main output: none.

Parameters:
- **filepath** (*str.*) – path of file to read. New dataframe is saved into *filepath* too.
- **columns_names** (*ndarray (n_newcolumns,)*) – an array containing names of the new columns.
- **columns_values** – an array or matrix containing values for the new columns.

Type: columns_values: ndarray (n_samples, n_newcolumns)

Returns: none.

Raise: none.

***static* get_furthest_elements(*subject_id, n_points, dataset_path, n_classes*)**

This function considers the portion of the dataset *dataset_path* related to subject *subject_id* (*n_classes* are retrieved) and returns the *n_points* furthest elements from the centroid of each class. Furthest elements are identified using cosine similarity.

Pre-conditions: none.
Post-conditions: dataset is read and furthest elements from each centroid are returned.
Main output: furthest elements from each class.

Parameters:
- **subject_id** (*int (between 1 and 9)*) – the id of the subject.
- **n_points** (*int (>0)*) – number of points returned for each class.
- **dataset_path** (*str.*) – the path to the dataset.
- **n_classes** (*int (in {4, 5})*) – number of classes considered.

Returns: a matrix containing the furthest *n_points* from the centroid of each class.

Return type: ndarray (n_classes, n_points)

Raise: none.

***static* get_neighbors_elements(*subject_id, dataset_path, n_classes*)**

This function considers the portion of the dataset *dataset_path* related to subject *subject_id* (*n_classes* are retrieved) and for each possible combination of the classes returns a pair of elements (one of class_1 and one of class_2) such that no other pair has an higher cosine similarity than it.

Pre-conditions: none.
Post-conditions: dataset is read and closest elements of different classes are returned.
Main output: closest elements of different classes.

Parameters:
- **subject_id** (*int (between 1 and 9)*) – the id of the subject.
- **dataset_path** (*str.*) – the path to the dataset.
- **n_classes** (*int (in {4, 5})*) – number of classes considered.

Returns: a matrix containing the furthest *n_points* from the centroid of each class.

Return type: ndarray (n_classes, n_points)

Raise: none.

static **read_dataset(*subject_id, n_classes, test_split, dataset_path*)**

This function read the dataset whose path is passed as parameter, considering only the specified subject and number of classes.

Pre-conditions: none.
Post-conditions: dataset is read and split into training and test set.
Main output: training set and test set, according to the given parameters.

Parameters:
- **subject_id** (*int (between 1 and 9)*) – the id of the subject.
- **n_classes** (*int (in {4, 5})*) – number of classes considered.
- **test_split** (*float (in [0, 1])*) – the proportion of data that will form the test set.
- **dataset_path** (*str.*) – the path to the dataset.

Returns: training_features, test_features, training_labels, test_labels, training_ids, test_ids.

Return type: ndarray, ndarray, ndarray, ndarray, ndarray, ndarray.

:raise:none.

static **read_dataset_by_ids(*subject_id, test_ids, dataset_path*)**

This function read the dataset whose path is passed as parameter, considering only the specified subject and number of classes. Elements whose ids are *test_ids* will represent the test set, while all the other samples will represent the training set (5 classes are considered)

Pre-conditions: none.
Post-conditions: dataset is read and split into training and test set.
Main output: training set and test set, according to the given parameters.

Parameters:
- **subject_id** (*int (between 1 and 9)*) – the id of the subject.
- **test_ids** (*ndarray (n_samples,)*) – ids of the elements that will form the test set.
- **dataset_path** (*str.*) – the path to the dataset.

Returns: training_features, test_features, training_labels, test_labels, training_ids, test_ids.

Return type: ndarray, ndarray, ndarray, ndarray, ndarray, ndarray.

:raise:none.

***static* return_band(*feature_number*)**

This function returns band of the feature whose index is passed as parameter.

Pre-conditions: none.
Post-conditions: band of the feature is obtained.
Main output: band.

Parameters: **feature_number** (*int (between 0 and 131)*) – the index of the feature for which name is requested.

Returns: band name.

Return type: str.

Raise: none.

***static* return_electrode_number(*feature_number*)**

This function returns electrode number of the feature whose index is passed as parameter.

Pre-conditions: none.
Post-conditions: electrode number of the feature is obtained.
Main output: electrode number.

Parameters: **feature_number** (*int (between 0 and 131)*) – the index of the feature for which name is requested.

Returns: electrode number.

Return type: str.

Raise: none.

***static* return_feature_name(*feature_number*)**

This function returns the name of the feature whose index is passed as parameter.

Pre-conditions: none.
Post-conditions: name of the feature is obtained.
Main output: the name of the feature.

Parameters: **feature_number** (*int (between 0 and 131)*) – the index of the feature for which name is requested.

Returns: the name of the feature.

Return type: str.

Raise: none.

src.Utils.ExplanationsUtils module

***class* src.Utils.ExplanationsUtils.ExplanationsUtils**

Bases: `object`

This class implements some useful functions for obtaining explanations for given samples.

***static* random_forest_explanations(*x*, *y*, *n_classes*)**

This function creates a new RandomForestClassifier and generates explanations from that model for the given data. Explanations for each class are computed as the sum of explanations of all samples that are part of that class.

Pre-conditions: none.
Post-conditions: RandomForestExplanation for the given data are computed.
Main output: explanations for each class.

Parameters:

- **x** (*ndarray (n_samples, n_features)*) – an array that contains features of the samples for which explanation is required.
- **y** (*ndarray (n_samples,)*) – an array containing labels for the samples in *x*.
- **n_classes** (*unsigned int.*) – number of classes of the considered problem.

Returns: explanation for the given classes.

Return type: ndarray (n_classes, n_features)

Raise: none.

src.Utils.Plots module

class src.Utils.Plots.Plots

Bases: `object`

This class implements some functions useful for plotting results.

static abs_shap(*df_shap*, *df*)

This function is taken from https://towardsdatascience.com/explain-your-model-with-the-shap-values-bc36aac4de3d. It generates a particular bar plot from the given data.

Pre-conditions: none.
Post-conditions: a new bar plot is generated from the given data.
Main output: none.

Parameters:
- **df_shap** (*DataFrame.*) – a dataframe containing shap values for each feature.
- **df** (*DataFrame.*) – a dataframe containing a set of samples.

Returns: none.

Raise: none.

static plot_bar_chart(*values*, *categories*, *x_label*, *xlim*, *title*, *display_error*)

This function plots a bar chart using the given parameters.

Pre-conditions: none.
Post-conditions: a new bar chart is generated from the given data.
Main output: none.

Parameters:
- **values** (*ndarray (n_categories,)*) – values on which plot is created. Each value is in the form 'mean±std'.
- **categories** (*ndarray (n_categories,)*) – categories that will be used into the bar chart. A category will represent a label on the axis.
- **x_label** (*str.*) – name associated to the x axis.
- **xlim** (*ndarray (2,)*) – lower and upper bound of the x axis.
- **title** (*str.*) – title of the plot.
- **display_error** (*boolean.*) – True if error lines are needed.

Returns: none.

Raise: none.

static **plot_confusion_matrix(*cm, cms, classes, filepath, title, cmap=<matplotlib.colors.LinearSegmentedColormap object>*)**

This function prints and plots the confusion matrix, given matrices of means and stds. Taken from https://stackoverflow.com/questions/59319533/plot-a-confusion-matrix-in-python-using-a-dataframe-of-strings.

Pre-conditions: none.
Post-conditions: a new confusion matrix is generated from the given data.
Main output: none.

Parameters:
- **cm** (*ndarray (n_classes,n_classes)*) – matrix of means.
- **cms** (*ndarray (n_classes,n_classes)*) – matrix of std.
- **classes** (*ndarray (n_classes,)*) – labels of classes.
- **filepath** (*str.*) – filepath in which confusion matrix image is stored.
- **title** (*str.*) – title of the plot.
- **cmap** (*str or Colormap*) – color map.

Returns: none.

Raise: none.

static **plot_feature_importance(*labels, values, subject_id, sample_id*)**

This function creates a bar plot from the data passed in labels and values.

Pre-conditions: none.
Post-conditions: bar plot created from labels and values is shown.
Main output: none.

Parameters:
- **labels** (*ndarray.*) – an array that specifies the labels of the bar plot.
- **values** (*ndarray.*) – an array that specifies the values for the elements in *labels*.
- **sample_id** (*int (>0)*) – id of the sample to which data refer.
- **subject_id** (*int (between 1 and 9)*) – id of the subject to which data refer.

Returns: none.

Raise: none.

static **plot_grouped_bar_chart(*labels, values, categories, x_label, y_label, xlim, title, display_error*)**

This function plots a grouped bar chart using the given parameters.

Pre-conditions: none.
Post-conditions: a new grouped bar chart is generated from the given data.
Main output: none.

Parameters:

- **labels** (*ndarray (n_labels,)*) – labels that will be used into the bar chart. Each set of bars is associated with a label.
- **values** (*ndarray (n_categories, n_labels)*) – values on which plot is created. Each value is in the form 'mean±std'.
- **categories** (*ndarray (n_categories,)*) – categories that will be used into the bar chart. Each set of bars is made up by n_categories bars.
- **x_label** (*str.*) – name associated to the x axis.
- **y_label** (*str.*) – name associated to the y axis.
- **xlim** (*ndarray (2,)*) – lower and upper bound of the x axis.
- **title** (*str.*) – title of the plot.
- **display_error** (*boolean.*) – True if error lines are needed.

Returns: none.

Raise: none.