PALLADIO

PALLADIO Documentation

Release 2.0.3rc1

Matteo Barbieri **Samuele Fiorini** **Federico Tomasi**
Annalisa Barla

May 29, 2017

CONTENTS

PALLADIO is a machine learning framework whose purpose is to provide robust and reproducible results when dealing with data where the signal to noise ratio is low. It also provides tools to determine whether the dataset being analyzed contains any signal at all. **PALLADIO** works by repeating the same experiment many times, each time resampling the learning and the test set so that the outcome is reliable as it is not determined by a *single* partition of the dataset. Besides, using permutation tests, it is possible to provide, to some extent, a measure of how reliable the results produced by an experiments are. Since all experiments performed are independent, PALLADIO is designed so that it can exploit a cluster where it is available, in order to greatly reduce the amount of time required.

The final output of **PALLADIO** consists of several plots and text reports. The main ones are:

- A plot showing the absolute frequencies of features for both *regular* experiments and permutation tests. Another plot shows in more detail the selection frequency for the most frequently selected features (i.e., those above the *selection threshold* defined in the configuration file).
- A plot showing the distribution of accuracies achieved by *regular* experiments and permutation tests.
- Two text files listing the features together with their absolute selection frequency, one for regular experiments and the other for permutation tests.

See *Quick start tutorial* for instructions on how to setup a cluster using the deployment script included in the **PALLADIO** distribution.

CHAPTER

ONE

USER DOCUMENTATION

1.1 Introduction

The issue of reproducibility of experiments is of paramount importance in scientific studies, as it influences the reliability of published findings. However when dealing with biological data, especially genomic data such as gene expression or SNP microarrays, it is not uncommon to have a very limited number of samples available, and these are usually represented by a huge number of measurements.

A common scenario is the so called *case-control study*: some quantities (e.g., gene expression levels, presence of alterations in key *loci* in the genome) are measured in a number of individuals who may be divided in two groups, or classes, depending whether they are affected by some kind of disease or not; the goal of the study is to find **which ones**, if any, among the possibly many measurements, or *features*, taken from the individuals (*samples*), can be used to define a *function* (sometimes the term *model* is used as well) able to *predict*, to some extent, to which *class* (in this case, a diseased individual or a healthy one) an individual belongs.

Machine Learning (ML) techniques work by *learning* such function using only *part* of the available samples (the *training set*), so that the remaining ones (the *test set*) can be used to determine how well the function is able to predict the class of **new** samples; this is done, roughly speaking, to ensure that the function is able to capture some real characteristics of the data and not simply fitting the training data, which is trivial. This is referred to in ML literature as *binary classification scenario*.

In the aforementioned scenario, having only few samples available means that the learned function may be highly dependent on how the dataset was split; a common solution to this issue is to perform *K-fold cross validation* (KCV) which means splitting the dataset in K chunks and performing the experiment K times, each time leaving out a different chunk to be used as test set; this reduces the risk that the results are dependent on a particular split. The K parameter usually is chosen between 3 and 10, depending on the dataset.

This is the idea behind L1L2Signature , a framework specifically designed with this issue in mind. `L1L2Signature` performs *feature selection* while learning the function, that is it tries to identify which ones among the available features are actually *relevant* for the problem, that is *which are actually used* in the learned function. The output of `L1L2Signature` consists of a *signature*, that is a list of relevant features, as well as a measure of *prediction accuracy*, that is the ratio of correctly classified samples in the test set, averaged over all splits.

There are however cases where it is hard to tell whether this procedure actually yielded a meaningful result: for instance, the fact that the accuracy measure is only *slightly* higher than chance can indicate two very different things:

- The available features can only describe the phenomenon to a limited extent.
- There is actually no relationship between features and output class, and getting a result better than chance was just a matter of luck in the subdivision of the dataset.

In order to tackle this issue, **PALLADIO** repeats the experiment many times (~ 100), each time using a different training and test set by randomly sampling from the whole original dataset (without replacement). The experiment is also repeated the same number of times in a similar setting with a difference: in training sets, the labels are randomly shuffled, therefore destroying any connection between features and output class.

The output of this procedure is not a single value, possibly averaged, for the accuracy, but instead *two distributions of values* (one for each of the two settings described above) which, in case of datasets where the relationship between features and output class is at most faint, allows users to distinguish between the two scenarios mentioned above: in facts, if the available features are somehow connected with the outcome class, even weakly, then the two distributions will be different enough to be distinguished; if on the other hand features and class are not related in any way, the two distributions will be indistinguishable, and it will be safe to draw that conclusion.

1.2 The framework

A dataset consists of two things:

- An input matrix $X \in \mathbb{R}^{n \times p}$ representing n samples each one described by p features; in the case of gene expression microarrays for instance each feature represents
- An output vector $\mathbf{y}$ of length n whose elements are either a continuous value or a discrete label, describing some property of the samples. These may represent for example the levels of a given substance in the blood of an individual (continuous variable) or the *class* to which he or she belongs (for instance, someone affected by a given disease or a healthy control).

For the time being, we will only consider a specific instance of the latter case, where the number of classes is two: this is commonly referred to as *binary classification* scenario.

As previously explained, the core idea behind **PALLADIO** is to return, together with a list of significant features, not just a single value as an estimate for the prediction accuracy which can be achieved, but a distribution, so that it can be compared with the distribution obtained from experiments when the function is learned from data where the labels have been randomly shuffled (see *Introduction*).

1.2.1 Pipeline

Once the main script has been launched, the configuration file is read in order to retrieve all required information to run all the experiments of a **PALLADIO** *session*. These include:

- The location of **data** and **labels** files.
- Experiment design parameters, such as the total number of experiments and the ratio of samples to be used for testing in each experiment.
- Parameters specific to the chosen machine learning algorithm: for instance, for the $\ell_1\ell_2$ regularized algorithm, the values for the τ and λ parameters.

A *session folder* is created within the folder containing the configuration file, in order to keep everything as contained as possible; data and labels file, together with the configuration file itself, are copied inside this folder. Then, experiments are distributed among the machines of the cluster; each machine will be assigned roughly the same number of jobs in order to balance the load.

Experiments

Each experiment is divided in several stages, as shown in Fig. 1.1:

Dataset split and preprocessing

In the very first stage, the dataset is split in **training** and **test** set, in a ratio determined by the corresponding parameter in the experiment configuration file; also, during this stage, any kind of data preprocessing (such as centering or normalization) is performed.

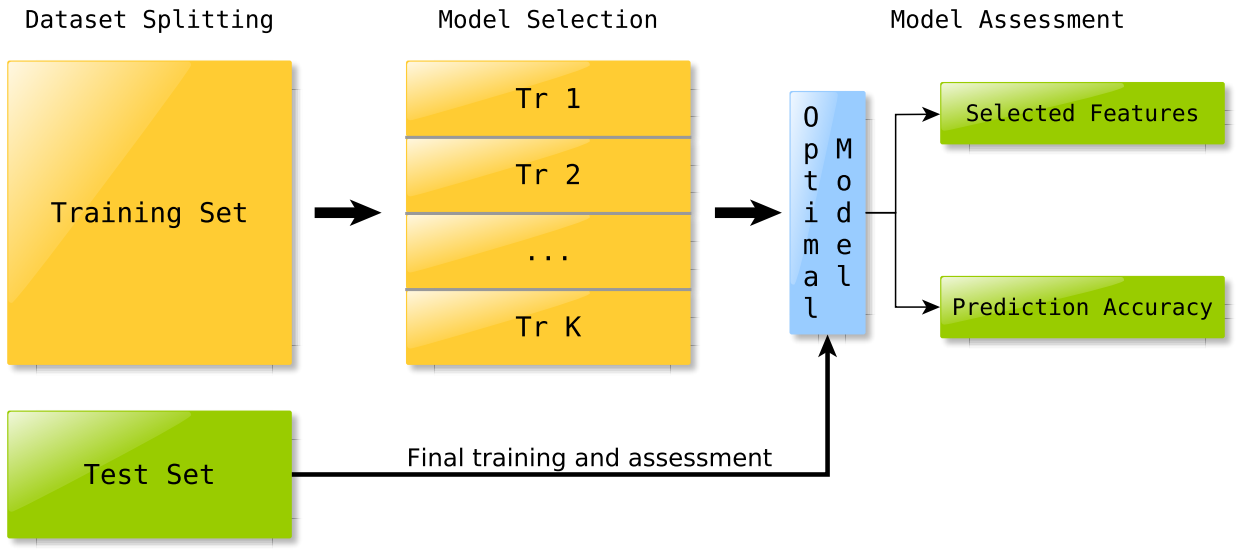

Fig. 1.1: The stages each experiment goes through.

Model selection

Assuming that the chosen classifier requires some parameter to be specified (for instance the ℓ_1 and squared ℓ_2 penalities weights when using the $\ell_1\ell_2$ regularized least square algorithm), the **training** set is split in K chunks (the number K is also specified in the experiment configuration file) and K-fold cross validation is performed in order to choose the best parameters, that is those which lead to the model with the lowest cross validation error.

Model assessment

Finally, the algorithm is trained using the parameters chosen in the previous step on the whole **training set**; the function obtained is then used to predict the labels of samples belonging to the **test set**, which have not been used so far in any way, so that the results of whole procedure are unbiased.

At the end of each experiment, results are stored in a `.pkl` file inside a subfolder whose name will be of the form `regular_p_P_i_I` for regular experiments and `permutation_p_P_i_I` for experiments where the training labels have been randomly shuffled, where `P` and `I` the process number and within that process a counter which is incremented by one after each experiment.

1.2.2 Analysis

The analysis script simply reads the partial results in all experiment folders, consisting of

- A list of features
- The predicted labels for the test set

With these it computes the accuracy achieved and then uses these elaborated results to produce a number of plots:

Fig. 1.2 shows the absolute feature selection frequency in both *regular* experiments and permutation tests; each tick on the horizontal axis represents a different feature, whose position on the vertical axis is the number of times it was selected in an experiment. Features are sorted based on the selection frequency relative to *regular* experiments; green dots are frequencies for *regular* experiments, red ones for permutation tests.

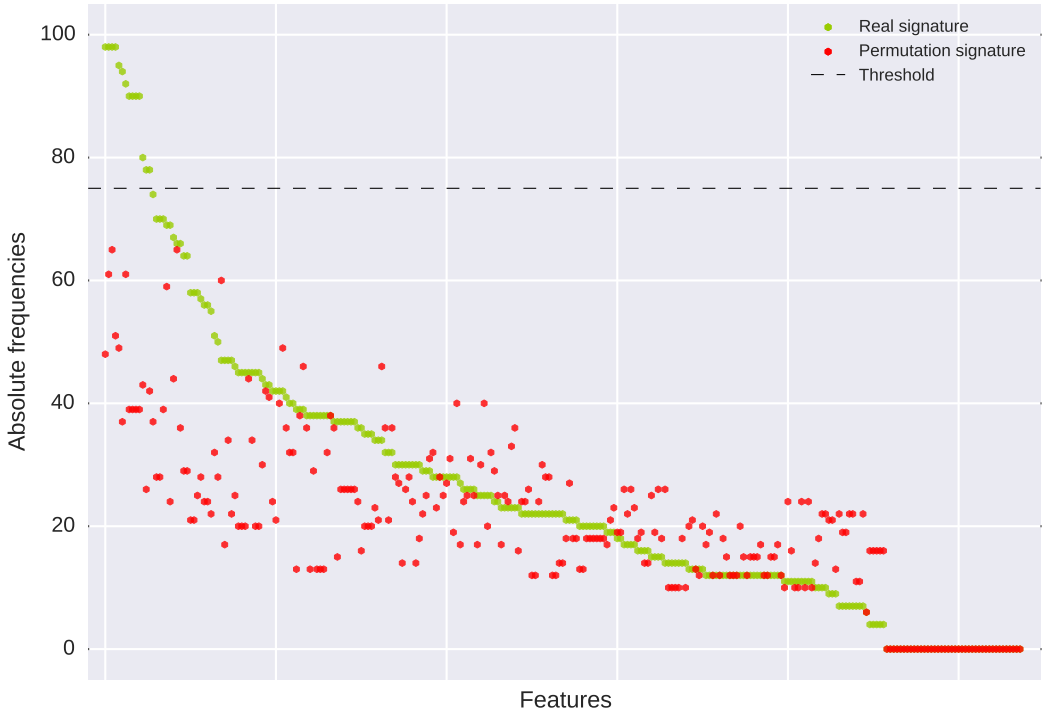

Fig. 1.2: A manhattan plot showing the distribution of frequencies for both *regular* experiments and permutation tests.

Fig. 1.3 shows a detail of the frequeny of the top $2 \times p_{\mathrm{rel}}$ selected features, where p_{rel} is the number of features identified as *relevant* by the framework, i.e. those which have been selected enough times according to the selection threshold defined in the configuration file. Seeing the selection frequency of *relevant* features with respect to the selection frequency of those which have been rejected may help better interpret the obtained results.

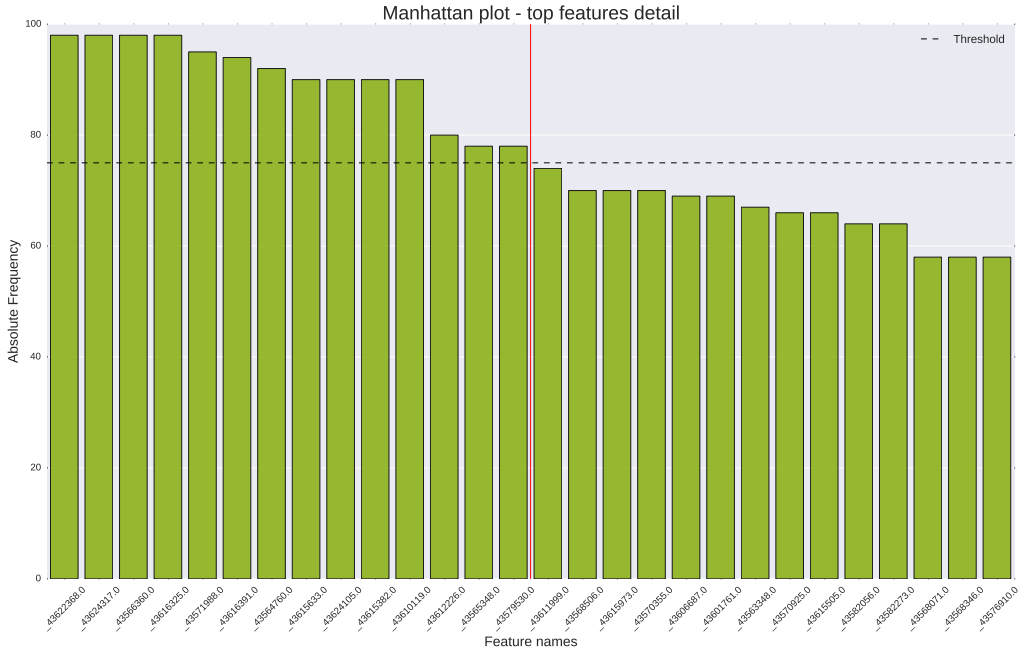

Fig. 1.3: A detail of the manhattan plot.

Finally, Fig. 1.4 shows the distribution of prediction accuracies (corrected for class imbalance) for *regular* experiments and permutation tests; this plot answer the questions:

- Is there any signal in the data being analyzed?
- If yes, how much the model can describe it?

In the example figure, the two distributions are clearly different, and the green one (showing the accuracies of *regular* experiments) has a mean which is significantly higher than chance (50 %). A p-value obtained with the Wilcoxon rank sum test is also present in this plot, indicating whether there is a significant difference between the two distributions.

1.3 Quick start tutorial

PALLADIO may be installed using standard Python tools (with administrative or sudo permissions on GNU-Linux platforms):

```
$ pip install palladio

or

$ easy_install palladio
```

We strongly suggest to use Anaconda and create an environment for your experiments.

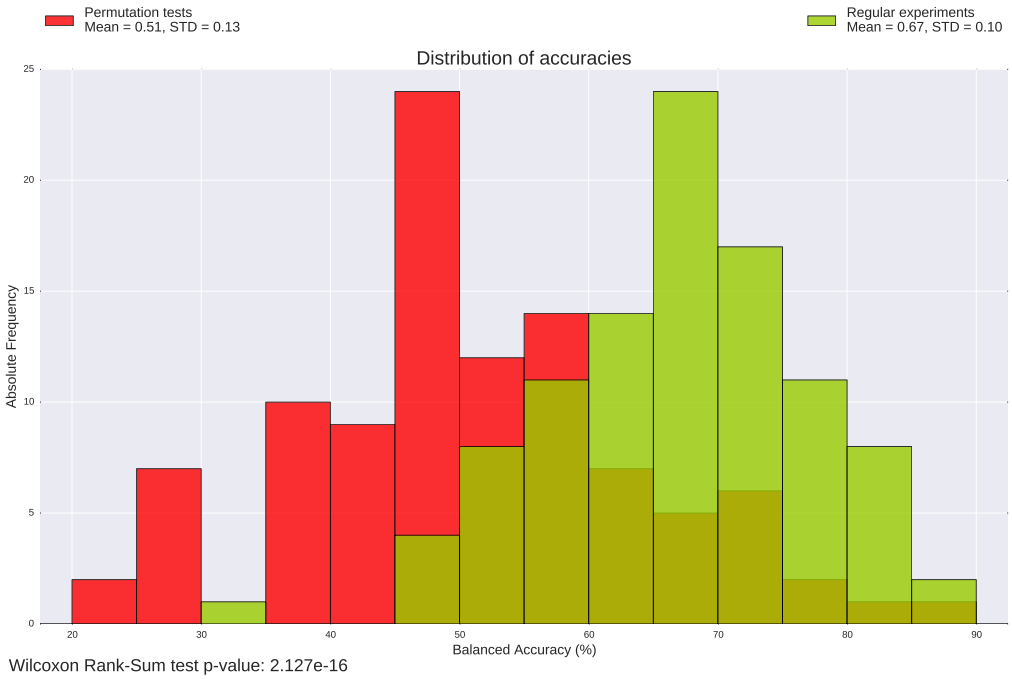

Fig. 1.4: The distributions of accuracies for both *regular* experiments and permutation tests.

1.3.1 Installation from sources

If you like to manually install **PALLADIO**, download the .zip or .tar.gz archive from http://slipguru.github.io/palladio/. Then extract it and move into the root directory:

```
$ unzip slipguru-palladio-|release|.zip
$ cd palladio-|release|/
```

or:

```
$ tar xvf slipguru-palladio-|release|.tar.gz
$ cd palladio-|release|/
```

Otherwise you can clone our GitHub repository:

```
$ git clone https://github.com/slipguru/palladio.git
```

From here, you can follow the standard Python installation step:

```
$ python setup.py install
```

After **PALLADIO** installation, you should have access to two scripts, named with a common `pd_` prefix:

```
$ pd_<TAB>
pd_analysis.py    pd_run.py
```

This tutorial assumes that you downloaded and extracted **PALLADIO** source package which contains a `palladio/config_templates` directory with some data files (`.npy` or `.csv`) which will be used to show **PALLADIO** functionalities.

PALLADIO needs only 3 ingredients:

- `n_samples x n_variables` input matrix
- `n_samples x 1` labels vector
- a `configuration` file

1.3.2 Cluster setup

Since all experiments performed during a run are independent from one another, **PALLADIO** has been designed specifically to work in a cluster environment. It is fairly easy to prepare the cluster for the experiments: assuming a standard configuration for the nodes (a shared home folder and a python installation which includes standard libraries for scientific computation, namely `numpy`, `scipy` and `sklearn`, as well as of course the `mpi4py` library for the MPI infrastructure), it is sufficient to transfer on the cluster a folder containing the dataset (data matrix and labels) and the configuration file, and install **PALLADIO** itself following the instructions above.

1.3.3 Configuration File

PALLADIO configuration file is a standard Python script. It is imported as a module, then all the code is executed. In this file the user defines all the parameters required to run a *session*, that is to perform all the experiments required to produce the final plots and reports.

In folder `palladio/config_templates` you will find an example of a typical configuration file. Every configuration file has several sections which control different aspects of the procedure.

The code below contains all the information required to load the dataset which will be used in the experiments.

```
data_path = 'data/gedm.csv'
target_path = 'data/labels.csv'

# pandas.read_csv options
data_loading_options = {
    'delimiter': ',',
    'header': 0,
    'index_col': 0
}
target_loading_options = data_loading_options

dataset = datasets.load_csv(
    os.path.join(os.path.dirname(__file__),data_path),
    os.path.join(os.path.dirname(__file__),target_path),
    data_loading_options=data_loading_options,
    target_loading_options=target_loading_options,
    samples_on='col')

data, labels = dataset.data, dataset.target
feature_names = dataset.feature_names
```

The last two lines store the input data matrix `data` and the labels vector `labels` in two variables which will be accessible during the session. The names of the features are also saved at this point. Notice how it is possible to load the dataset in any desired way, as long as `data` ends up being a $n \times d$ matrix and `labels` a vector of n elements (both `np.array`-like).

Next, we have the section containing settings relative to the session itself:

```
session_folder = 'palladio_test_session'

# The learning task, if None palladio tries to guess it
# [see sklearn.utils.multiclass.type_of_target]
learning_task = None

# The number of repetitions of 'regular' experiments
n_splits_regular = 50

# The number of repetitions of 'permutation' experiments
n_splits_permutation = 50
```

The most important settings are the last two, namely `n_splits_regular` and `n_splits_permutation`, which control how many repetitions of *regular* and *permutations* experiments are performed. Normally you'll want to perform the same number of experiments for the two *batches*, but there are cases in which for instance you may want to perform only one of the two batches: in that case you will want to set one of the two variables to be 0.

Finally, the section of the configuration file where the actual variable selection and learing algorithms (and their parameters) are chosen:

```
model = RFE(LinearSVC(loss='hinge'), step=0.3)

# Set the estimator to be a GridSearchCV
param_grid = {
    'n_features_to_select': [10, 20, 50],
    'estimator__C': np.logspace(-4, 0, 5),
}

estimator = GridSearchCV(
  model,
  param_grid=param_grid,
  cv=3,
  scoring='accuracy',
  n_jobs=1)

# Set options for ModelAssessment
ma_options = {
    'test_size': 0.25,
    'scoring': 'accuracy',
    'n_jobs': -1,
    'n_splits': n_splits_regular
}
```

This is maybe the less intuitive part of the file. Because of the way **PALLADIO** is designed, for all repetitions of the experiment a new learning and test set are generated by resampling without replacement from the whole dataset, then an estimator is used to fit the learning set. This is where that estimator (and its parameter) is defined.

Think about the `estimator` variable as the `sklearn`-compatible object (an estimator) which you would use to fit a training set, with the intent of validating it on a separate test set.

In this example we use a RFE algorithm (see http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html) for variable selection, which internally uses a Linear SVM for classification. Then we use a `GridSearchCV` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) object to wrap the RFE object, because we want to optimize the parameters for the RFE object itself, which are defined just above the declaration of the `estimator` variable.

The dictionary `ma_options` define some more configuration options for the *`ModelAssessment`* object, which is the one responsible for the outer iterations (the ones where the dataset is resampled); the `test_size` key for instance

determins the portion of data left aside for testing.

1.3.4 Running the experiments

Parallel jobs are created by invoking the `mpirun` command; the following syntax assumes that the OpenMPI implementation of MPI has been chosen for the cluster, if this is not the case, please refer to the documentation of the implementation available on your cluster for the command line options corresponding to those specified here:

```
$ mpirun -np N_JOBS --hostfile HOSTFILE pd_run.py path/to/config.py
```

Here `N_JOBS` obviously determines how many parallel jobs will be spawned and distributed among all available nodes, while `HOSTFILE` is a file listing the addresses or names of the available nodes.

Take into account that if optimized linear algebra libraries are present on the nodes (as it is safe to assume for most clusters) you should tune the number of jobs so that cores are optimally exploited: since those libraries already parallelize operations, it is useless to assign too many slots for each node.

Running experiments on a single machine

It is possible to perform experiments using **PALLADIO** also on a single machine, without a cluster infrastructure. The command is similar to the previous one, it is sufficient to omit the first part, relative to the MPI infrastructure:

```
$ pd_run.py path/to/config.py
```

Warning: Due to the great number of experiments which are performed, it might take a very long time for the whole procedure to complete; this option is therefore deprecated unless the dataset is very small (no more than 100 samples and no more than 100 features).

1.3.5 Results analysis

The `pd_analysis.py` script reads the results from all experiments and produces several plots and text files. The syntax is the following:

```
$ pd_analysis.py path/to/results_dir
```

See *Analysis* for further details on the output of the analysis.

1.4 API

1.4.1 Pipeline utilities

Nested Cross-Validation for scikit-learn using MPI.

This package provides nested cross-validation similar to scikit-learn's GridSearchCV but uses the Message Passing Interface (MPI) for parallel computing.

class palladio.model_assessment.**ModelAssessment**(*estimator*, *cv=None*, *scoring=None*, *fit_params=None*, *multi_output=False*, *shuffle_y=False*, *n_jobs=1*, *n_splits=10*, *test_size=0.1*, *train_size=None*, *random_state=None*, *groups=None*, *experiments_folder=None*, *verbose=False*)

Cross-validation with nested parameter search for each training fold.

The data is first split into cv train and test sets. For each training set a grid search over the specified set of parameters is performed (inner cross-validation). The set of parameters that achieved the highest average score across all inner folds is used to re-fit a model on the entire training set of the outer cross-validation loop. Finally, results on the test set of the outer loop are reported.

Parameters **estimator** : object type that implements the "fit" and "predict" methods

A object of that type is instantiated for each grid point.

cv : integer or cross-validation generator, optional, default: 3

If an integer is passed, it is the number of folds. Specific cross-validation objects can be passed, see sklearn.cross_validation module for the list of possible objects

scoring : string, callable or None, optional, default: None

A string (see model evaluation documentation) or a scorer callable object / function with signature scorer(estimator, X, y). See sklearn.metrics.get_scorer for details.

fit_params : dict, optional, default: None

Parameters to pass to the fit method.

multi_output : boolean, default: False

Allow multi-output y, as for multivariate regression.

shuffle_y : bool, optional, default=False

When True, the object is used to perform permutation test.

n_jobs : int, optional, default: 1

The number of jobs to use for the computation. This works by computing each of the Monte Carlo runs in parallel. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. Ignored when using MPI.

n_splits: int, optional, default: 10 The number of cross-validation splits (folds/iterations).

test_size : float (default 0.1), int, or None

If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is automatically set to the complement of the train size.

train_size : float, int, or None (default is None)

If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state [int or RandomState, optional, default: None] Pseudo-random number generator state used for random sampling.

groups [array-like, with shape (n_samples,), optional, default: None] Group labels for the samples used while splitting the dataset into train/test set.

experiments_folder : string, optional, default: None

The path to the folder used to save the results.

verbose : bool, optional, default: False

Print debug messages.

Attributes

scorer_	function	Scorer function used on the held out data to choose the best parameters for the model.
cv_results_	dictionary	Result of the fit. The dictionary is pandas.DataFrame-able. Each row is the results of an external split. Columns are: 'split_i', 'learn_score', 'test_score', '**cv_results_**', 'ytr_pred', 'yts_pred', 'test_index', 'train_index', 'estimator' Example: >>> pd.DataFrame(**cv_results_**) split_i | learn_score | test_score | **cv_results_** | ... 0 | 0.987 | 0.876 | {<internal splits>} | ... 1 | 0.846 | 0.739 | {<internal splits>} | ... 2 | 0.956 | 0.630 | {<internal splits>} | ... 3 | 0.964 | 0.835 | {<internal splits>} | ...

fit (*X*, *y*)
Fit the model to the training data.

1.4.2 Extra tools

Utilities functions and classes.

palladio.utils.**save_signature** (*filename*, *selected*, *threshold=0.75*)
Save signature summary.

palladio.utils.**retrieve_features** (*best_estimator*)
Retrieve selected features from any estimator.

In case it has the 'get_support' method, use it. Else, if it has a '**coef_**' attribute, assume it's a linear model and the features correspond to the indices of the coefficients != 0

palladio.utils.**get_selected_list** (*grid_search*, *vs_analysis=True*)
Retrieve the list of selected features.

Retrieves the list of selected features automatically identifying the type of object

Returns **index** : nunmpy.array

The indices of the selected features

palladio.utils.**build_cv_results**(*dictionary*, ***results*)
Function to build final **cv_results_** dictionary with partial results.

palladio.utils.**signatures**(*splits_results*, *frequency_threshold=0.0*)
Return (almost) nested signatures for each correlation value.

The function returns 3 lists where each item refers to a signature (for increasing value of linear correlation). Each signature is orderer from the most to the least selected variable across KCV splits results.

Parameters **splits_results** : iterable

List of results from L1L2Py module, one for each external split.

frequency_threshold : float

Only the variables selected more (or equal) than this threshold are included into the signature.

Returns **sign_totals** : list of `numpy.ndarray`.

Counts the number of times each variable in the signature is selected.

sign_freqs : list of `numpy.ndarray`.

Frequencies calculated from `sign_totals`.

sign_idxs : list of `numpy.ndarray`.

Indexes of the signatures variables .

Examples

```
>>> from palladio.utils import signatures
>>> splits_results = [{'selected_list':[[True, False], [True, True]]},
...                   {'selected_list':[[True, False], [False, True]]}]
>>> sign_totals, sign_freqs, sign_idxs = signatures(splits_results)
>>> print sign_totals
[array([ 2.,  0.]), array([ 2.,  1.])]
>>> print sign_freqs
[array([ 1.,  0.]), array([ 1. ,  0.5])]
>>> print sign_idxs
[array([0, 1]), array([1, 0])]
```

palladio.utils.**selection_summary**(*splits_results*)
Count how many times each variables was selected.

Parameters **splits_results** : iterable

List of results from L1L2Py module, one for each external split.

Returns **summary** : `numpy.ndarray`

Selection summary. `# mu_values X # variables` matrix.

palladio.utils.**confusion_matrix**(*labels*, *predictions*)
Calculate a confusion matrix.

From given real and predicted labels, the function calculated a confusion matrix as a double nested dictionary. The external one contains two keys, `'T'` and `'F'`. Both internal dictionaries contain a key for each class label. Then the `['T']['C1']` entry counts the number of correctly predicted `'C1'` labels, while `['F']['C2']` the incorrectly predicted `'C2'` labels.

Note that each external dictionary correspond to a confusion matrix diagonal and the function works only on two-class labels.

Parameters **labels** : iterable

Real labels.

predictions : iterable

Predicted labels.

Returns **cm** : dict

Dictionary containing the confusion matrix values.

`palladio.utils.`**`classification_measures`**(*confusion_matrix*, *positive_label=None*)

Calculate some classification measures.

Measures are calculated from a given confusion matrix (see *`confusion_matrix()`* for a detailed description of the required structure).

The `positive_label` arguments allows to specify what label has to be considered the positive class. This is needed to calculate some measures like F-measure and set some aliases (e.g. precision and recall are respectively the 'predictive value' and the 'true rate' for the positive class).

If `positive_label` is None, the resulting dictionary will not contain all the measures. Assuming to have to classes 'C1' and 'C2', and to indicate 'C1' as the positive (P) class, the function returns a dictionary with the following structure:

```
{
    'C1': {'predictive_value': --,  # TP / (TP + FP)
           'true_rate':        --}, # TP / (TP + FN)
    'C2': {'predictive_value': --,  # TN / (TN + FN)
           'true_rate':        --}, # TN / (TN + FP)
    'accuracy':          --,        # (TP + TN) / (TP + FP + FN + TN)
    'balanced_accuracy': --,        # 0.5 * ( (TP / (TP + FN)) +
                                    #          (TN / (TN + FP)) )
    'MCC':               --,        # ( (TP * TN) - (FP * FN) ) /
                                    # sqrt( (TP + FP) * (TP + FN) *
                                    #       (TN + FP) * (TN + FN) )

    # Following, only with positive_labels != None
    'sensitivity':       --,        # P true rate: TP / (TP + FN)
    'specificity':       --,        # N true rate: TN / (TN + FP)
    'precision':         --,        # P predictive value: TP / (TP + FP)
    'recall':            --,        # P true rate: TP / (TP + FN)
    'F_measure':         --         # 2. * ( (Precision * Recall ) /
                                    #        (Precision + Recall) )
}
```

Parameters **confusion_matrix** : dict

Confusion matrix (as the one returned by *`confusion_matrix()`*).

positive_label : str

Positive class label.

Returns **summary** : dict

Dictionary containing calculated measures.

palladio.utils.**set_module_defaults**(*module*, *dictionary*)
Set default variables of a module, given a dictionary.

Used after the loading of the configuration file to set some defaults.

palladio.utils.**sec_to_timestring**(*seconds*)
Transform seconds into a formatted time string.

Parameters **seconds** : int

Seconds to be transformed.

Returns :

——— :

time : string

A well formatted time string.

palladio.utils.**safe_run**(*function*)
Decorator that tries to run a function and prints an error when fails.

PYTHON MODULE INDEX

p

INDEX