LYNX
User Manual

A MATLAB Toolbox for Fast Prototyping of Machine Learning Simulations

SIMONE SCARDAPANE

CONTENTS

Introduction **i**
What is Lynx ? i
Structure of this Document ii

1 First Use **1**
1.1 Installing the Toolbox 1
1.2 Folder Structure 2
1.3 General Workflow 3
1.4 Running the First Test 4
1.4.1 Initialization Phase 5
1.4.2 Training Phase 5
1.4.3 Output Phase 5
1.5 Understanding Tasks and Analyzing the Output 6
1.5.1 Basic Tasks 6
1.5.2 Complex Tasks 7
1.5.3 Training Performance 8

2 Writing a Configuration File **9**
2.1 General Configuration 9
2.2 Adding an Algorithm 10
2.3 Adding a Wrapper 11
2.3.1 Performing a Parameter Sweep 12
2.3.2 Saving and Loading a Configuration 13
2.4 Adding a Dataset 13
2.4.1 Adding a Prediction Dataset 14
2.4.2 Adding a Multi-label Dataset 14
2.5 Adding a Preprocessor to a Dataset 15
2.5.1 Preprocessors and Multi-label Tasks 15
2.6 Choosing a Statistical Test 15

3 Additional Features **17**
3.1 Parallelizing the Experiments 17
3.1.1 Configuring a Cluster 18
3.2 Enabling the GPU 18
3.3 Saving the Results 18
3.4 Adding Additional Output Scripts 19
3.5 Changing the Performance Measure 20
3.6 Using Semi-Supervised Algorithms 20
3.7 Constructing a Hierarchical Learning Algorithm 21

4 Designing New Functionalities **22**
4.1 Storing a Dataset 22
4.2 Implementing an Algorithm 23
4.2.1 Setting the Training Parameters 23
4.2.2 Training 24
4.2.3 Testing 25
4.2.4 Class Information 25
4.2.5 Implementing a Semi-Supervised Algorithm 25
4.3 Implementing a Wrapper 26
4.3.1 Evaluating an Algorithm 26
4.4 Implementing a Preprocessor 27
4.5 Implementing a new Performance Measure 27
4.6 Designing a Non-Basic Task 27
4.7 Designing a Partitioning Strategy 28
4.8 Designing a Statistical Test 28

INTRODUCTION

What is Lynx ?

Lynx is a research-oriented Matlab toolbox for simulating in a fast way large machine learning experiments in the context of supervised and semi-supervised learning. To understand its goal, consider the following scenario: you, the researcher, have just developed a novel learning algorithm, commonly a variant of an already existing one. For simplicity, let us call them A and B. To test the efficacy of A involves comparing it against B (and possibly some additional baseline algorithms) on a large set of datasets. We call such a comparison a *simulation*. Results of a given simulation are then summarized and analyzed (maybe with the use of a statistical test) to understand relative strength and weaknesses of A with respect to B. Additionally, a simulation should be easily repeatable and modifiable, such that any other researcher employing similar tools can repeat it exactly or add its own additional features (e.g. another variant of B).

The converse situation is also typical: testing a large set of algorithms, possibly with different model selection strategies, on a particular dataset, to understand which one has the highest accuracy. Both settings become more complex if we possess more than a single machine, and we want to distribute the computational load among them, or if we are interested in speeding-up some of our computations using a GPU-based approach. This, together with the need of defining suitable strategies for subdividing data, and taking care that data itself is properly preprocessed, typically drive away the researcher from the high-level problems of the algorithm towards low-level, implementation-driven details.

Lynx is designed to handle both these situations using an high-level approach, taking charge of most part of the previously mentioned requirements in an automatic fashion. Hence, the user is left with the possibility of simply focusing on the elements of the simulation (e.g., which algorithms to include), and possibly on the details of a newly implemented feature. In particular, the user can specify the parameters of a simulation in a configuration file, which

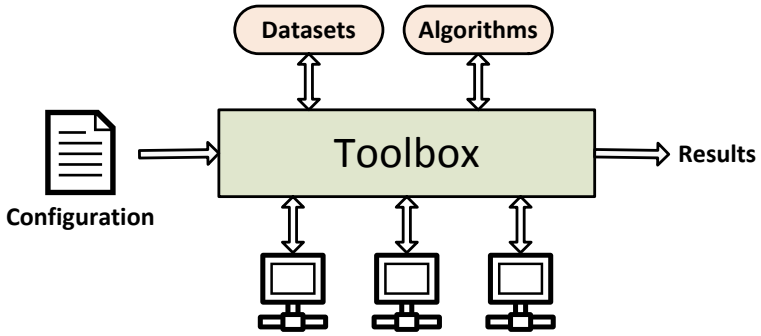

Figure 1: Schematic representation of the toolbox

is then loaded by the software. The toolbox takes charge of importing the requested datasets, partitioning them, running the algorithms, collecting the results and analyzing them. It can distribute such simulations on multiple threads, and multiple computers, using the Parallel Computing Toolbox[1] and the Distributed Computing Server[2] of Matlab. Moreover, many of the pre-implemented algorithms have the possibility of directly running on the GPU. The general schema of the toolbox is summarized in Fig. 1.

At the end of any experiment, the toolbox provides information on the errors of each algorithm, their training times, and performs a statistical testing of the results (if possible). All the information can additionally be saved in a folder (including a full transcript of the simulation) or exported in the form of a pdf file. A wide set of datasets is already included in the library, and new ones can be added by storing them in a proper format[3]. Also, since the main core of the toolbox is fully object-oriented, additional algorithms and functionalities can be designed by simply extending specified classes, and are successively recognized automatically by the software.

Structure of this Document

The rest of this guide is structured as follows. Chapter 1 starts by showing how to install the toolbox, along with any optional library. To start familiarizing with the toolbox, a sample configuration file is provided with it. The second part of Chapter 1 shows how to run it and analyzes briefly the behavior of the software. Then, Chapter 2 details the way in which a configuration file can be customized. Additional functionalities (e.g., saving the results) are explored successively in Chapter 3. Finally, Chapter 4 explains the core classes of the software and how to add new algorithms and datasets to it.

[1] `http://www.mathworks.it/products/parallel-computing/`

[2] `http://www.mathworks.it/products/distriben/`

[3] Additional datasets can also be downloaded from the author's webpage at `http://ispac.ing.uniroma1.it/scardapane/software/lynx/`.

CHAPTER
ONE

FIRST USE

1.1 Installing the Toolbox

Once the toolbox has been download and extracted, it can be installed by running the file "*install.m*" in the root folder. This adds all the toolbox folders to the search path of Matlab. Additionally, the installation script checks for the presence of any required external library. Currently, 3 libraries are needed by the toolbox: LibSVM[1], the DeepLearn Toolbox[2], and the Kernel Methods Toolbox[3]. None of them is essential, but some specific algorithms depend on them. If a library is not found, the installation script asks for the permission of downloading it and storing it in the "lib" folder:

```
Checking for presence of Deep Learn Toolbox...
Deep Learn Toolbox not found. Do you want to install it? (Y/N)
```

If the user denies the permission, the installation process continues, but the functionalities depending on the external library cannot be used. In this example, if the user denies the permission to install the DeepLearn Toolbox, a warning message will tell us that the *StackedAutoEncoder* algorithm depends on it:

```
Deep Learn Toolbox not found. Do you want to install it? (Y/N) N
Warning: You will not be able to use the StackedAutoEncoder algorithm
```

Due to the presence of a large set of folders, the search path is not saved by default. Hence, the installation script must be run every time Matlab is restarted. Alternatively, it is possible to save the path after the installation by running the built-in "*savepath*" command.

[1] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`
[2] `https://github.com/rasmusbergpalm/DeepLearnToolbox`
[3] `http://sourceforge.net/projects/kmbox/`

1.2 Folder Structure

Before continuing, let us look briefly at the folder structure of Lynx after the installation process. In Fig. 1.2 we show a unix-like representation of the directories, together with a brief comment on their contents.

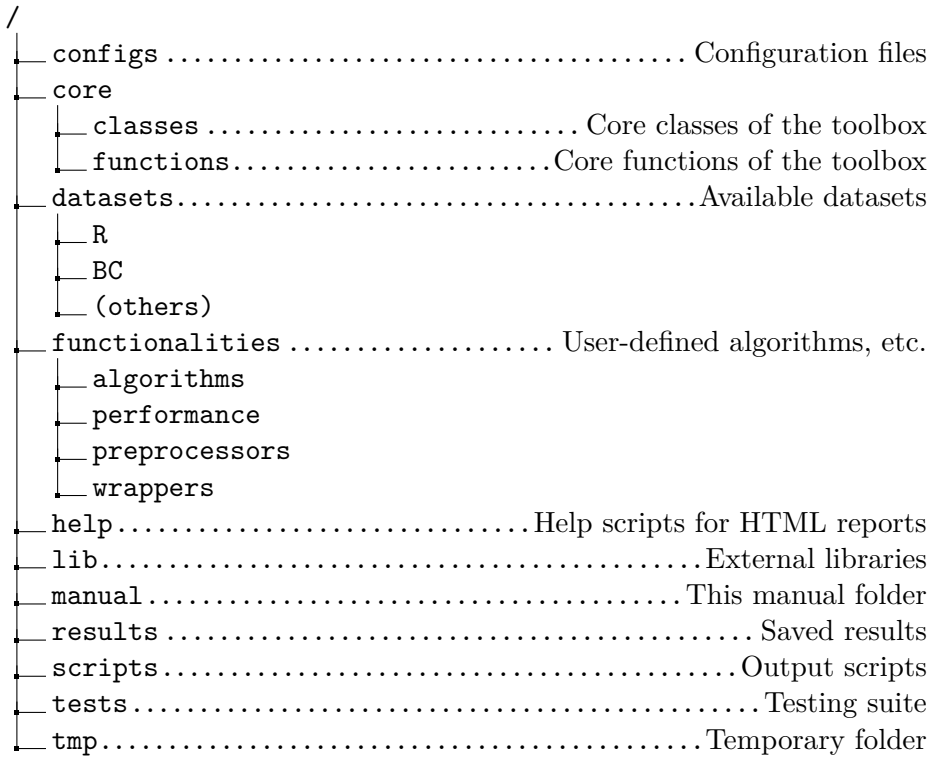

Figure 1.1: Folder Structure of the Toolbox

Most of the directories contents are self-explanatory and will be explored in depth in the rest of this manual, but some general comments are in order. First, datasets are partitioned into multiple subfolders, corresponding to the internal concept of "*task*", introduced in Section 1.5. All the user-defined functionalities, which can be roughly subdivided into 4 families (learning algorithms, performance measures, preprocessors and wrappers) are stored in the corresponding subfolder of "*functionalities*". Note that the toolbox already comes with several ready-to-use implementations and datasets. To get a general overview at the currently available functionalities and datasets, help scripts in the *help* folder can be used to generate HTML reports. Internally, this is done using the Matlab Report Generator[4]. Output scripts are used for showing additional information after the simulation, and are introduced in Section 3.4. The temporary folder is emptied at the end of every simulation,

[4] `http://www.mathworks.it/products/ML_reportgenerator/`

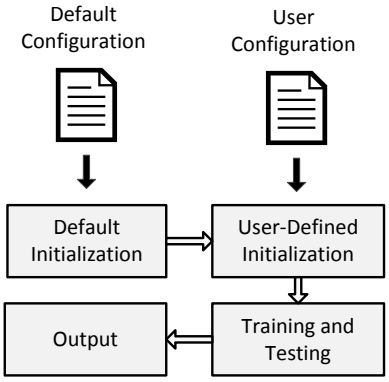

Figure 1.2: High-level schema of the execution of the toolbox

and whenever Matlab is closed. Finally, the toolbox is supplemented by a suite of unitary tests for checking the correctness of the code. This suite should be run every time a change is made to the code of the toolbox.

1.3 General Workflow

The general workflow of the toolbox is depicted in Fig. 1.2. The simulation starts by loading a default configuration, saved in "*configs/default_config.m*". Then, the user-defined configuration is loaded, corresponding to the file "*config.m*" in the root folder of the toolbox. In this file, the user can specify what algorithms must be tested, and on which datasets. Additionally, it is possible to override some general behavior of the toolbox, and to enable some advanced features, such as parallelization of the experiments.

According to these requirements, the toolbox trains and tests a set of models on the requested data, and at the end shows on the console the average training times and testing accuracies. To understand briefly the training phase, consider the simplest case of a single algorithm, tested on a single dataset, using a random partition of the data. The worflow of execution is detailed in Fig. 1.3. The dataset is loaded from memory and preprocessed using a set of methods specified in the configuration file. Then, the preprocessed dataset is subdivided into a training set and a testing set. The training set is then passed in input to a learning algorithm, to build a suitable model. The learning algorithm can be encapsulated into one (or more) wrappers, allowing for the possibility of fine-tuning its parameters, choosing the right subset of input features, and many additional functionalities. Finally, the testing set is passed as input to the model, which provides a set of predicted outputs. This outputs are compared with the expected ones using a given measure of accuracy, that determines the final testing error of the algorithm. All the blocks of Fig. 1.3 are fully configurable

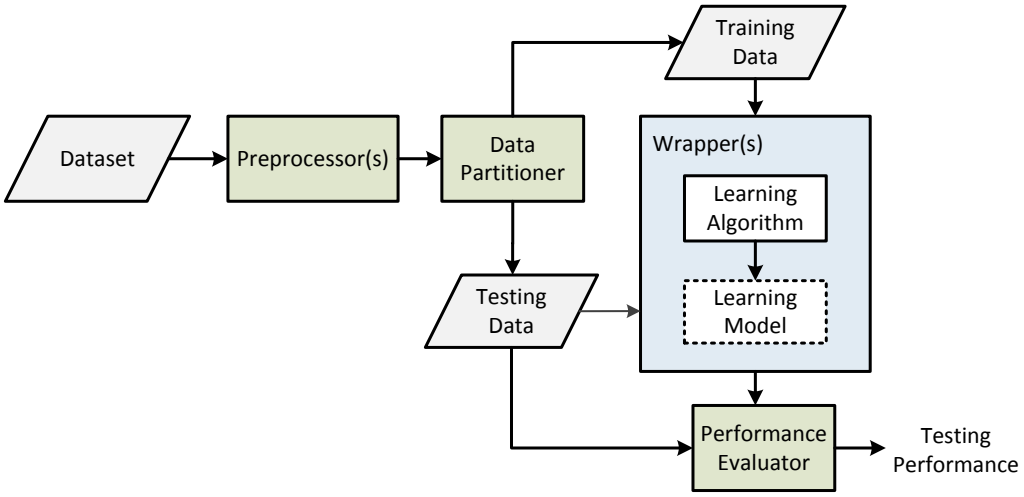

Figure 1.3: Schematic Depiction of the Training Phase

and extensible.

In the more general case, the data can be split according to a k-fold cross validation, hence the training/testing process is repeated k times and results are averaged. This can be repeated for multiple algorithms and datasets, although the preprocessing step and the partitioning steps are common to all the algorithms so as to ensure a fair comparison. Additionally, the overall process can be repeated multiple times (*runs*).

1.4 Running the First Test

To help familiarizing with the software, the configuration file already contains the instructions to run a very simple experiment. Although its syntax is explored in detail in the following chapter, for the sake of this demo it is enough to understand that this example requires the test of two different learning algorithms:

```
add_algorithm('B', 'Baseline', @Baseline);
add_algorithm('ELM', 'Extreme Learning Machine', ↩
    @ExtremeLearningMachine);
```

The "baseline" algorithm is a dummy learning algorithm who, in this case, always outputs the mean value of its training set. The second algorithm is a more sophisticated Extreme Learning Machine trained using an $L2$-regularized ridge regression [1]. The test is executed on a single dataset taken from the UCI repository[5]:

[5] `http://archive.ics.uci.edu/ml/`

```
add_dataset('G', 'Glass', 'uci_glass');
```

In this case, we have no wrappers and no preprocessors. The simulation is started by executing the "*run_simulation.m*" script in the root folder. The result of the simulation is then printed on the console. Below we provide a small description of each phase.

1.4.1 Initialization Phase

In the initialization phase, the configuration file is read and all the data structures are initialized:

```
Initializing simulation...
Current seed for prng: 1340863332
End of initialization, will test 2 algorithm(s) on 1
   dataset(s) for 1 time(s)
```

Other preliminary actions are eventually run in this phase. For example, the toolbox checks the compatibility of all the algorithms with the requested datasets (see next section); it activates a pool of threads if parallelization is requested; and so on.

1.4.2 Training Phase

In the training phase, each algorithm is run on every dataset in turn. In this case, the default configuration specifies to execute a 3-fold cross validation, and a single run:

```
Testing Extreme Learning Machine on Glass (run 1/1)
   Fold 1...
   Fold 2...
   Fold 3...
Testing Baseline on Glass (run 1/1)
   Fold 1...
   Fold 2...
   Fold 3...
```

1.4.3 Output Phase

In the output phase, the results, in the form of mean and standard deviation of the errors and the training times, are printed on screen. As an example, these are typical average errors for the demo:

```
Average error:
       Baseline  Extreme Learning Machine
Glass     1.014                    0.5375
```

ID	Input	Output
R (Regression)	$\mathbf{X} \in \mathbb{R}^{N \times d}$	$\mathbf{y} \in \mathbb{R}^N$
BC (Binary Classification)		$\mathbf{y} \in \{-1, +1\}^N$
MC (Multiclass Classification)		$\mathbf{y} \in \{1, \ldots, M\}^N$

Table 1.1: Summary of the basic tasks defined in the toolbox

Note that the actual meaning of the numbers may vary depending on the dataset. This is explained in more detail in the next section.

After this step, an optional statistical testing is executed. See Section 2.6 for information on how to request it. Currently supported statistical tests include a Wilcoxon signed rank test [2], and a Friedman test [3]. Additional output scripts can also be added to the simulation for gathering information on the run. This functionality is explored in Section 3.4.

1.5 Understanding Tasks and Analyzing the Output

1.5.1 Basic Tasks

To understand the results, a few more details on the structure of the toolbox is required. Speaking broadly, a dataset is composed of an $N \times d$ matrix of real values (currently, categorical variables are not supported), and a $N \times 1$ vector of targets, where N is the number of examples available to the system, and d is the input dimensionality. That is, each row in the input matrix is an observation, whereas each column is a feature.

A *task* identifies the type of learning problem of a given dataset, which changes the type of the requested output. The following basic tasks are currently defined:

Regression (R) : the output is a single real number.

Binary Classification (BC) : the output is an integer number in the set $\{-1, +1\}$.

Multiclass Classification (MC) : the output is an integer number in the set $\{1, \ldots, M\}$, where M is the number of classes.

These are summarized in Table 1.1. An algorithm can be defined to work on one or more of them. For example, a particular algorithm can be defined only for classification tasks, i.e. BC and MC tasks, but not for regression tasks. If in the simulation some inconsistencies are found (i.e., an algorithm is tested on an unsupported task), they are listed in the beginning

ID	Input	Output
PR (Prediction)	$\mathbf{x} \in \mathbb{R}^N$	NA
ML (Multilabel Classification)	$\mathbf{X} \in \mathbb{R}^{N \times d}$	$\{\mathbf{y}_i\}, \mathbf{y}_i = \{-1, +1\}^N$

Table 1.2: Summary of the complex tasks defined in the toolbox

and the statistical testing is avoided. The following is an example of the corresponding warning message:

```
The following tests are not possible:
         KRLS on Yeast.
Statistical testing will not be executed.
```

Successively, the corresponding training phase is skipped:

```
Testing KRLS on Yeast (run 1/1)
        Not Allowed
```

1.5.2 Complex Tasks

Clearly, the tasks of Table 1.1 do not cover all the possibilities arising in supervised learning. For this reason, the toolbox identifies a set of non-basic tasks, i.e., tasks that do not conform to the previous description. Practically, what changes is either the semantic of the problem, or the way in which the data is stored on the disk. A non-basic task is handled by previously transforming it into one or more basic tasks. Currently, two non-basic tasks are defined:

Prediction (PR) : here the input is a vector containing the ordered samples of a time-invariant time series, and the required output is a one-step ahead prediction. This is handled by transforming it into a regression problem, where the input is an embedding of the previous r values of the time-series, with r specified by the user. If the time series has N values, $N - r$ examples are constructed.

Multilabel (ML) : this is the case where a given number T of binary classification problems are defined on the same input matrix. This is handled by constructing T separate binary classification tasks during the initialization phase.

The structure of each complex task is summarized in Table 1.2. In Table 1.3, instead, we show how each complex task is handled by the toolbox.

Original Task	Transformed Tasks
PR	A single R task.
ML	Multiple BC tasks.

Table 1.3: Transformations between Complex Tasks and Basic Tasks

1.5.3 Training Performance

The measure of performance shown in the results changes depending on the task associated to each dataset. By default, the misclassification rate is shown for classification tasks, while the Normalized Root Mean-Squared error [6] for regression, defined as:

$$\text{NRMSE}(\mathbf{y}, \mathbf{d}) = \sqrt{\frac{\sum_{i=1}^{N}(d_i - y_i)^2}{N\hat{\sigma}_y}} \tag{1.1}$$

where $\mathbf{y}$ are the expected outputs, $\mathbf{d}$ are the actual outputs, N is the cardinality of both $\mathbf{y}$ and $\mathbf{d}$, and $\hat{\sigma}_y$ is an empirical estimate of the variance of $\mathbf{y}$. The measure of error is averaged over all the folds and all the runs.

As an example, consider the sample run of Section 1.4.3. Since the Glass dataset is a regression task, the numbers that appear in the console are the NRMSE computed over the testing set, averaged over the different folds. The default performance measures can be changed by the user, a functionality explored in Section 3.5. Also, new performance measures can be defined, see Section 4.5.

[6] `http://en.wikipedia.org/wiki/Root-mean-square_deviation`

CHAPTER
TWO

WRITING A CONFIGURATION FILE

2.1 General Configuration

Referring again to Fig. 1.2, remember that the default parameters governing the behavior of the simulation are stored in the file "*configs/default_config.m*". This file is loaded automatically by the toolbox before loading the user-defined one. Hence, each setting defined in the default configuration can be adapted by simply redefining it in the user-defined configuration file, which effectively overrides the default parameter.

Below is a list of the most important parameters defined in the default config file:

- `nRuns` : the number of times the simulation will be run. As an example, if `nRuns` is set equal to 2, each test will be performed twice. This defaults to 1.
- `partition_strategy`: an object describing how the toolbox should subdivide the data for testing. This can be any object of class `PartitionStrategy`. Currently, two possibilities are available:
  - `partition_strategy` can be an object of class `HoldoutPartition`, in which case a specified fraction of data will be kept for testing (this is called *holdout*). For example, by setting:

    ```
    partition_strategy = HoldoutPartition(0.3);
    ```

    each algorithm will be trained on a randomly selected 70% of the data, and tested on the remaining 30%.
  - `partition_strategy` can be an object of class `KFoldPartition`, in which case a k-fold cross validation is performed. Typical values of folds in this case are between 3 and 10.

The default behavior is a 3-fold cross-validation. Note that the partitioning strategy can equivalently be set using the `set_partition_strategy` method.

- `seed_prng`: the seed for initializing the pseudo-random number of Matlab. This can be an integer number or the string "shuffle" (default value), in which case a random seed is used. In the latter case, the random seed is printed on screen for allowing the repetition of the exact experiment.

Hence, the default configuration comprises the following parameters:

```
nRuns = 1;
partition_strategy = KFoldPartition(3);
seed_prng = 'shuffle';
```

Note that configuration files can be easily nested. As an example, suppose we have a range of configurations (e.g. a set of algorithms or datasets) that we include frequently in our simulations. These can be written on a particular file, e.g. "*configs/own_config.m*", and imported when needed in the standard configuration file:

```
own_config.m;
```

2.2 Adding an Algorithm

An algorithm can be added to the simulation with the following syntax:

```
add_algorithm(ID, name, pointer, varargin);
```

where:

- `ID` is a unique string for identifying the algorithm in the simulation,
- `name` is the algorithm's name (to be displayed in the results),
- `pointer` is a pointer to the algorithm's class,
- `varargin` are the additional parameters to be passed to the constructor.

As in the standard convention of Matlab coding, additional parameters are composed as follows:

- One or more required parameters,
- One or more additional parameters,

- One or more name/value pairs. As a general rule, most parameters are in this form.

For example, the following call:

```
add_algorithm('NN', 'Neural Net', @MultilayerPerceptron);
```

adds a default-initialized Multilayer Perceptron to the simulation. Similarly, the call:

```
add_algorithm('NN', 'Neural Net', @MultilayerPerceptron, '↩
    hiddenNodes', 15);
```

adds a Multilayer Perceptron, with 15 nodes in the hidden layer.

A concise HTML report with a list of all implemented algorithms and respective parameters can be found by calling the script "*help/info_algorithms.m*". Additional information on each algorithm can be found by visualizing the help for the respective class.

2.3 Adding a Wrapper

A wrapper provides additional functionalities to an algorithm (called in this context its *base algorithm*) by encapsulating its behavior and intercepting inputs and outputs to its training and testing functions. Wrappers can be used for fine-tuning model parameters, extracting or selecting features, saving the models resulting from the simulation, etc.

The syntax for adding a wrapper is similar to the syntax for adding an algorithm:

```
add_wrapper(ID, wrapper, varargin);
```

Where:

- `ID` is the ID of the algorithm to be encapsulated,
- `wrapper` is a pointer to the wrapper's class,
- `varargin` are the additional parameters for the wrapper.

Consider as an example the following call:

```
add_wrapper('NN', @Featuresearch_GA);
```

This adds a wrapper to the Multilayer Perceptron defined in the previous section, that runs a genetic algorithm for searching the optimal subset of features. For details on the implemented wrappers and required parameters, the script "*help/info_wrappers.m*" generates a concise report. More information

on each wrapper is available by visualizing the help for the corresponding class.

It is important to note that wrappers can pile on top of each other:

```
add_wrapper(ID, wrapper1, ...);
add_wrapper(ID, wrapper2, ...);
```

In this case, the *wrapper2* is executed by using as a base algorithm the *wrapper1*, which in turn uses as a base class the algorithm identified by ID. Consider the following:

```
add_wrapper('NN', @Featuresearch_GA);
add_wrapper('NN', @OneVersusAll);
```

In this case, for a multiclass classification problems with M classes M different neural networks are trained. Each network will have a different subset of input parameters, found by the genetic search. The order in which the wrappers are stacked is very important. The semantic of the following set of calls is different from the previous one:

```
add_wrapper('NN', @OneVersusAll);
add_wrapper('NN', @Featuresearch_GA);
```

Here, a single genetic search is executed, and internally the algorithm is trained using a one-versus-all strategy.

Many wrappers require a parameter designating how to subdivide the data for performing a validation step. In this case, this parameter follows the same semantic as the `partition_strategy` defined for the general configuration.

2.3.1 Performing a Parameter Sweep

A very common need in supervised learning is testing a given set of values of one or more parameters of an algorithm, then choosing the combination resulting in the highest level of accuracy. In the toolbox, this functionality is provided natively by the *ParameterSweep* wrapper. As an example of its usage, consider the Extreme Learning Machine (ELM) adopted in the sample configuration, whose simulation has been analyzed in the previous chapter. It has two parameters, a regularization factor (denoted by `regularizationFactor`) and a kernel parameter (denoted by `kernel_para`). As is standard practice, we want to test the following range of values for each parameter:

$$2^{-5}, 2^{-4}, \ldots, 2^{5} \tag{2.1}$$

Since we have two parameters, this results in $10 \times 10 = 100$ configurations to be tested. The validation accuracy of each configuration should be computed by performing an inner 3-fold cross validation on the training data. The following command generates the corresponding wrapper in our example:

```
add_wrapper('ELM', @ParameterSweep, KFoldPartition(3), {'↩
    regularizationFactor', 'kernel_para'}, {'exp', 'exp'}, [-5 5; -5↩
     5], [1 1]);
```

Although this may seem daunting at first, it is actually rather simple. The first parameter has a semantic similar to the `partition_strategy` discussed in Section 2.1, and it instructs the wrapper to perform a 3-fold cross validation. This is followed by a cell array with the names of the parameters to be tested. The next cell array tells the wrapper that the values must be computed in an exponential fashion, i.e., by powers of 2. Then, we provide the lower and upper bounds for each parameter, and the step values for computing the ranges.

2.3.2 Saving and Loading a Configuration

The second common requirement that we analyze here briefly is that of saving the configuration of an algorithm, for retrieving it in a following simulation. As an example, the grid search of the previous subsection requires training and testing 100 models, each time performing a 3-fold cross validation. For large datasets, this requires a very large time, hence we would like to save the results for loading it successively. The *SaveConfiguration* wrapper can be used for saving a configuration:

```
add_wrapper('ELM', @SaveConfiguration, 'ELM_SAVED', './sweeps');
```

After training a model, this will be saved in the "*./sweeps*" folder. Note that a simulation requires training several models, one for each fold and dataset. Hence, this will save several files in the folder. The naming convention is that a file is denoted as ID_DATASET_FOLD.mat, where ID is the id defined above ("*ELM_SAVED*" in this example), DATASET is the name of the dataset, and FOLD is the numerical id of the fold. The saved models can be retrieved on a successive simulation using the *LoadConfiguration* wrapper:

```
add_wrapper('ELM', @LoadConfiguration, './sweeps', 'ELM_SAVED', {'↩
    regularizationFactor', 'kernel_para'});
```

Note that this wrapper requires explicitly a list of training parameters to be loaded.

2.4 Adding a Dataset

A new dataset can be added to the simulation with the following syntax:

```
add_dataset(ID, name, dataset, [subsample], varargin);
```

Where:

- `ID` is a string identifying the dataset in the simulation,
- `name` is a name for the dataset, to be displayed in the results,
- `dataset` is the unique alphanumerical string denoting the dataset in the filesystem,
- `subsample` is a percentage of the dataset to load (optional),
- `varargin` are additional parameters to be given in specific cases.

As an example, the call:

```
add_dataset('Y', 'Yacht', 'uci_yacht');
```

adds the dataset "*uci_yacht*" to the simulation with name "*Yacht*", while:

```
add_dataset('Y', 'Yacht', 'uci_yacht', 0.5);
```

adds the same dataset, but loads only a randomly chosen 50% of it. A list of available datasets (and respective ids) can be obtained by calling "*help/info_datasets.m*". The datasets are divided depending on the task. Below we provide more information on the way in which each complex task can be loaded.

2.4.1 Adding a Prediction Dataset

In the case of a prediction task, the toolbox requires the embedding dimension for the input vector:

```
add_dataset('MG', 'Mackey-Glass', 'mackeyglass', 1, 'embeddingFactor↩
    ', 7);
```

Note that in this case it is necessary to specify a subsampling percentage.

2.4.2 Adding a Multi-label Dataset

Remember from Section 1.5 that a multi-label task with M labels is loaded as M different binary classification tasks. The name of each of these sub-tasks is created from the name provided by the user and an additional string contained in the dataset. The id, instead, is created sequentially starting from the user-defined id. As an example, suppose *ml_example* is a multi-label dataset consisting of 3 labels. Consider the following call:

```
add_dataset('EX', 'Multi-Label Dataset', 'ml_example');
```

This call creates 3 distinct binary classification tasks, with ids given by EX-1, EX-2, and EX-3.

2.5 Adding a Preprocessor to a Dataset

A preprocessor applies some specific transformation to a dataset in the initialization phase. A preprocessor can be added with the following syntax:

```
add_preprocessor(ID, preprocessor, varargin);
```

Where:

- `ID` is a regular expression that should match with all the datasets ids to which the preprocessor will be applied,
- `preprocessor` is a pointer to the preprocessor's class,
- `varargin` are the additional parameters of the preprocessor.

As an example, the call:

```
add_preprocessor('Y', @ApplyPca, 'varianceToPreserve', 0.95);
```

applies a PCA transformation to the dataset previously defined, retaining only the principal components encompassing at least 95% of the variance of the original input.

A list of the available preprocessors is obtained by calling the script "*help/info_preprocessors.m*". Additional information on each preprocessor is given by the help of the corresponding class.

2.5.1 Preprocessors and Multi-label Tasks

The fact that the syntax for adding a pre-processor accepts a regular expressions is particularly suited for multi-label tasks. Continuing the example of Section 2.4.2, suppose we now want to perform a PCA on all three binary classification tasks. This can be done with a single call:

```
add_preprocessor('EX-.', @ApplyPca, 'varianceToPreserve', 0.95);
```

The regular expression follows the standard Matlab convention: in particular, the dot refers to "any character", allowing to match all three datasets together. More information on the syntax of regular expressions can be found by reading the help of the built-in "*regexp*" function.

2.6 Choosing a Statistical Test

A statistical testing is requested by calling the following method:

```
set_statistical_test(class);
```

where `class` is a pointer to a class deriving from `StatisticalTest`. Each statistical test has certain conditions that must be satisfied so that it can be called. As an example, the Wilcoxon signed-rank test requires exactly two algorithms and at least two datasets. So, if we add it to a simulation like:

```
set_statistical_test(@WilcoxonTest);
```

but we do not respect the conditions, a critical error is issued during the initialization phase.

CHAPTER
THREE

ADDITIONAL FEATURES

3.1 Parallelizing the Experiments

We refer to the test of an algorithm on a dataset as an *experiment*. The simulation can be easily parallelized due to the fact that each experiment is independent of all the others. Hence, in the general case where you are testing N algorithms on M datasets for R times, you have $N \times M \times R$ independent experiments to be performed. These can be parallelized over multiple threads and multiple machines by enabling the *parallelized* flag in the configuration file:

```
set_flag('parallelized');
```

This functionality requires the Parallel Computing Toolbox of Matlab. It uses the pool of workers defined as *default* in the configuration of Matlab itself. When this is enabled, the order in which the experiments are performed cannot be defined *a-priori*. For this reason, learning algorithms are not allowed to print additional information on the screen. The pool of workers is started in the initialization phase:

```
Starting matlabpool using the 'local' profile ...
        connected to 2 workers.
```

If the workers are subdivided on multiple machines, the user has the responsibility of making the toolbox code available on every worker (to save long transfer times during initialization). During the training phase, printing is disabled but we are shown the worker in which each experiment is performed between square brackets:

```
Testing Baseline on Glass (run 1/1) [Worker 1]
Testing ExtremeLearningMachine on Glass (run 1/1) [Worker 2]
```

In the output phase, the pool is closed:

```
Sending a stop signal to all the workers ... stopped.
```

3.1.1 Configuring a Cluster

Describing how to setup a cluster goes beyond the scope of this manual. To this end, we refer to the Mathworks documentation:
`http://www.mathworks.it/support/product/DM/installation/ver_current/`

3.2 Enabling the GPU

An additional form of parallelization is given by the use of the GPU. This is enabled with the *gpu_enabled* flag in the configuration file:

```
set_flag('gpu_enabled');
```

This functionality requires the Parallel Computing Toolbox of Matlab and a supported NVIDIA GPU device. Additionally, the latest version of the CUDA drivers has to be installed. The GPU compatibility is tested in the initialization phase:

```
Initializing GPU device...
```

The rest of the simulation does not change. Note that only a subset of the implemented algorithms actually use GPU acceleration. To see if a particular algorithm can benefit from this functionality, refer to the respective help of the class.

3.3 Saving the Results

The results of the simulation can be saved by enabling the *save_results* flag:

```
set_flag('save_results');
```

They are saved inside a subfolder of the *results* folder. The default name for the subfolder is "*test*", but this can be changed by redefining the `simulationName` parameter. Two files are saved:

- A .mat file with the workspace at the end of the simulation.
- A .txt file with the full transcript of the simulation (the same which is shown in the console).

Additionally, a pdf file with the results can be saved by enabling the *generate_pdf* flag in the configuration file:

```
set_flag('generate_pdf');
```

This requires the *pdflatex* compiler available on the system. Both the source file and the resulting pdf are saved in the same folder as the previous files. Fig. 3.1 shows an example of results as shown in the pdf file[1].

Dataset	Baseline	ExtremeLearningMachine
Glass	1.02 ±0.000000	0.5476 ±0.000000

Table 3.1: Experimental results (Error).

3.4 Adding Additional Output Scripts

Output scripts are used to analyze the simulation and print additional information on the console. They must be placed inside the *scripts* folder of the toolbox. One or more of them are run after a simulation with the following syntax:

```
output_scripts = {script1, script2, ...}
```

As an example, consider the parameter sweep detailed in Section 2.3.1. This is an example of running the simulation with the additional wrapper:

```
Testing ExtremeLearningMachine on Glass (run 1/1)
        Fold 1...
                 Validated parameters: [ 1.000000 100.000000 ],
                         with error: 0.558518
                 Final training time is: 0.003000
        Fold 2...
                 Validated parameters: [ 0.500000 100.000000 ],
                         with error: 0.605529
                 Final training time is: 0.004000
        Fold 3...
                 Validated parameters: [ 2.000000 50.000000 ],
                         with error: 0.490066
                 Final training time is: 0.003000
```

It can be seen that the resulting parameters are printed for each fold, together with the final training time of the ELM model. This is not easily understandable. To this end, we can use the *info_gridsearch* script:

```
output_scripts = {'info_gridsearch'};
```

[1]For increased readability, results are rotated 90° counter-clockwise in the resulting file.

In this way the average values of the training parameters, along with the average final training time, are printed at the end of the simulation on the console:

```
-------------------------------------------------
--- USER-DEFINED OUTPUT -------------------------
-------------------------------------------------
Results of grid search for algorithm ExtremeLearningMachine:
   Dataset Glass:
      Average training time is 0.003333 sec
      C = 1.166667
      hiddenNodes = 83.333333
```

3.5 Changing the Performance Measure

In some situations, the default performance measure is not adequate to the needs of the user. As an example, we may be interested in seeing the raw Mean-Squared Error (MSE) for our demo, instead of the more complex NRMSE. This can be changed with the following syntax:

```
set_performance(task, new_performance);
```

where `new_performance` is a pointer to the new performance measure class, and `task` is the task at which we want to associate it. In our case we can add:

```
set_performance(Tasks.R, @PerfMse);
```

The result of the simulation changes to reflect the new performance measure:

```
Average error:
       Baseline  ExtremeLearningMachine
Glass     4.409                   1.291
```

A full list of the available performance measures is found by calling the *info_perfmeasures* script in the "help" folder.

3.6 Using Semi-Supervised Algorithms

The toolbox has an experimental support for semi-supervised learning (SSL) algorithms [4]. SSL algorithms differ from classical learning algorithms in that they can use an additional set of unlabeled input patterns to increase their performance. By default, this functionality is disabled. Hence, if we add an SSL algorithm to the simulation:

```
add_algorithm('LAP-RLS', 'Laplacian RLS' @LaplacianRLS);
```

the LAP-RLS algorithms will be called with an empty additional training set. To enable it, we can set the corresponding flag:

```
set_flag('semisupervised');
```

If this flag is enabled, the following is performed:

- A given fraction of the original dataset (default is 25 %) is separated from the dataset. Labels for this part are discarded, and the corresponding input patterns are used as the additional training set.
- The rest of the dataset is split according to the testing requirements of the user.

The default fraction of semi-supervised data can be changed by setting the `semisupervised_holdout` variable:

```
semisupervised_holdout = 0.5;
```

3.7 Constructing a Hierarchical Learning Algorithm

TODO

CHAPTER
FOUR

DESIGNING NEW FUNCTIONALITIES

Adding new functionalities to the toolbox is composed of two distinct steps: storing a new dataset (detailed in the next section), or implementing a new algorithm. This last step requires extending a predefined abstract class. A general overview of these abstract classes is shown in Fig. 4.1. Each class is then explained in the corresponding section in this chapter.

4.1 Storing a Dataset

A new dataset must be stored as a .mat file inside the "*datasets/xxx*" folder, where *xxx* is the id of the corresponding task. Each mat file must contains the following variables:

1. `X`: an $N \times d$ matrix of input patterns, where N is the number of observations and d the dimensionality of the input.
2. `Y`: an N-dimensional vector of corresponding output values. See Section 1.5 for details on how each output is associated to a task.
3. `info`: a string describing the dataset.

Matrices can be stored as dense matrices or as sparse matrices. The name of the file identifies the dataset in the program. For non-basic tasks, the following applies, as detailed in Table 1.2:

1. For prediction tasks, `X` is an N-dimensional column vector containing the samples of the time-series, while `Y` is an empty matrix.
2. For multi-label tasks, `Y` is a $N \times T$ matrix, where T is the number of labels. Additionally, a `labels_info` variable must be present, corresponding to a $T \times 1$ cell array of strings with the label descriptions.

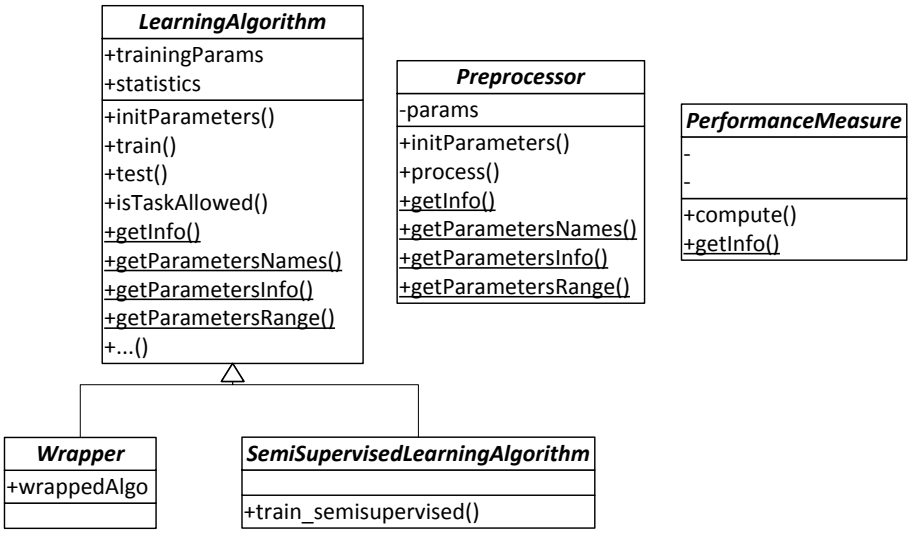

Figure 4.1: Principal Abstract Classes of the Toolbox

4.2 Implementing an Algorithm

Any algorithm should derive from the abstract class `LearningAlgorithm`. This requires implementing 8 abstract methods, 4 of which are static. To understand this process, throughout this section we suppose we want to implement a simple K-Nearest Neighbours algorithm [5], with the possibility of setting a given K.

4.2.1 Setting the Training Parameters

`LearningAlgorithm` defines two class properties: `trainingParams` is a struct containing all the training parameters, while `statistics` is a second struct where all additional information from the training process is saved. Training parameters are those that can be modified by the user when calling `add_algorithm`. They must be defined with the `initParameters` method:

```
initParameters(obj, p);
```

where p is an InputParser object[1]. The duty of the programmer is to fill the InputParser object with all required input parameters. The LearningAlgorithm constructor then parse its inputs, and save them in the *trainingParams* struct. In our case we only have a single training parameter:

```
classdef SimpleKNN < LearningAlgorithm

    properties
```

[1] `http://www.mathworks.it/it/help/matlab/ref/inputparserclass.html`

```
        knnStruct;
    end

    methods

        function obj = SimpleKNN(varargin)
            obj = obj@LearningAlgorithm(varargin);
        end

        function initParameters(~, p)
            p.addParamValue('K', 3, @(x) assert(mod(x,1) == x && x >↩
                0, 'K must be an integer > 0'));
        end

        ...
     end
end
```

Let us analyze this. First, we define a *knnStruct* that will hold the data pertaining to the training process. In the *initParameters* method, we add a single parameter, and we require it to be a non-negative integer. Since we have no additional initialization requirements, the constructor of our class simply calls the base constructor of LearningAlgorithm. Example of usage of this class are:

```
add_algorithm('K', 'K-NN', @SimpleKNN);
add_algorithm('K', 'K-NN', @SimpleKNN, 'K', 10);
```

4.2.2 Training

The next function we need to define is the one used for training a model:

```
function obj = train(obj, Xtr, Ytr)
   obj.knnStruct = ClassificationKNN.fit(Xtr,Ytr, 'NumNeighbors', ↩
       obj.trainingParams.K);
end
```

`Xtr` and `Ytr` are the input and output matrices, as defined in Section 1.5. In our case, we simply call the KNN algorithm in Matlab, setting properly the number of neighbors.

In general, the algorithm's behavior may change depending on the task. The current task can be obtained by calling the `getTask()` method. As an example, to check if the current task is a regression task, we can call:

```
obj.getTask() == Tasks.R.
```

Additionally, we need to define a function describing which tasks the algorithm allows:

```
function res = isTaskAllowed(~, ~)
   res = true;
end
```

Note that in this case we always return true, since we can use the KNN both for regression and for classification.

4.2.3 Testing

The third core method we need to define is the one used for testing. We are guaranteed that this is always called after calling the *train* method. Here is an example implementation in our case:

```
function [labels, scores] = test(obj, Xts)
    labels = predict(obj.knnStruct,Xts);
    scores = labels;
end
```

Here, `Xts` is a matrix of input test patterns in the same format as the matrix for training. `labels` should be an $N \times 1$ vector of predictions, where N is the number of rows in `Xts`, with the format specified by the current task. `scores` is a matrix of "raw" values of the algorithm, such as confidence values or probability estimates for each class. In our case, for simplicity, we set them equal to the labels.

4.2.4 Class Information

The class should define 4 static methods describing its model and the training parameters. These methods are called when constructing the report in the *help* folder. Below is a brief description of each method.

- `info = getInfo()` returns a string describing the algorithm.
- `names = getParametersNames()` returns a $P \times 1$ cell array of strings with the names of the training parameters.
- `info = getParametersInfo()` returns a $P \times 1$ cell array of strings with a description of each training parameter.
- `range = getParametersRange()` returns a $P \times 1$ cell array of string with the type of each training parameter (i.e. its range and possible default values).

Any algorithm inserted into the "*functionalities/algorithms*" subfolder is automatically detected by the system.

4.2.5 Implementing a Semi-Supervised Algorithm

Implementing a semi-supervised algorithm follows the same guidelines as before. The only difference is the signature of the training method:

```
function obj = train_semisupervised(obj, Xtr, Ytr, Xu);
```

Note that an implementation must handle the case where the additional matrix `Xu` is empty.

4.3 Implementing a Wrapper

A new wrapper should derive from the abstract class `Wrapper`, which itself derives from `LearningAlgorithm`. The main difference is that a wrapper has an additional property, `wrappedAlgo`, corresponding to an instance of its base learning algorithm.

As an example, consider part of the code for a Wrapper allowing to subsample the training dataset of a user-specified factor d:

```
classdef Subsampler < Wrapper
    methods
        function obj = train(obj, Xtr, Ytr)
           p = cvpartition(Ytr, 'holdout', obj.trainingParams.d);
           obj.wrappedAlgo = obj.wrappedAlgo.setTask(obj.getTask());
           obj.wrappedAlgo = obj.wrappedAlgo.train(Xtr(training(p), ↩
               :), Ytr(training(p));
        end
    end
end
```

The most important part in this code is that we need to set the current task in our base algorithm, before calling its training method. In general, we may also need to access some training property of the base algorithm, or set it to a new value (think of the `ParameterSweep` wrapper). However, wrappers can nest one inside each other, and the required training parameter can in principle be at any level of the hierarchy. `Wrapper` provides 2 methods to this end: `getTrainingParam` and `setTrainingParam`. As an example, suppose we want to access property "C" of our base algorithm. This is done as:

```
C = obj.getTrainingParam('C');
```

Then, we can increment it by one with:

```
obj.wrappedAlgo = obj.setTrainingParam('C', C + 1);
```

4.3.1 Evaluating an Algorithm

A common need when designing a wrapper is that of evaluating the performance of the base algorithm using a given partition of the data. To this end, three steps are needed. First, starting from the input training matrices X and Y we generate an anonymous dataset:

```
d = Dataset.generateAnonymousDataset(obj.getTask(), X, Y);
```

Successively, we generate the partitions to be used for validating. Supposing we have chosen a `PartitionStrategy` "p", we can call:

```
d = g.generateNPartitions(1, p);
```

Finally, we call an utility function to perform the actual test:

```
error = eval_algo(obj.wrappedAlgo, d);
```

4.4 Implementing a Preprocessor

A preprocessor should extend the abstract class `Preprocessor`. It design is very similar to that of a learning algorithm, with the following differences:

- The struct of parameters is called `params` instead of `trainingParams`.
- The preprocessor needs only a single non-static method instead of `train`, `test`, and `getAllowedTasks`. The signature of this method is:

  ```
  dataset = process(dataset);
  ```

  where `dataset` is an object of class `Dataset` representing the data. The output format should be in the same format.

4.5 Implementing a new Performance Measure

A new performance measure should derive from the base class `PerformanceMeasure`, and declare two static methods:

```
err = compute(labels, predictions);
```

where labels is an $N \times 1$ vector of true labels, and predictions an $N \times 1$ vector of predictions. `err` should be the computed measure of error. Note that, typically, a performance measure is defined to work only on a single task (e.g., misclassification error only for classification tasks). It is the duty of the user to associate a performance measure to its required task. The second method's signature is:

```
info = getInfo();
```

which must return a string describing the performance measure.

4.6 Designing a Non-Basic Task

Designing a non basic task requires three steps:

1. Defining a new task value in the class `Tasks`.

2. Creating the corresponding folder inside "datasets".

3. Defining a suitable `DatasetFactory` class. This must be defined inside the folder "*core/classes/DatasetFactories*", it must be called `DatasetFactoryxxx`, where `xxx` is the id of the task, and it must implement a single static method:

```
datasets = create(task, data_id, data_name, fileName, subsample↩
    , varargin);
```

   where `task` is the id of the task; `data_id` and `data_name` are the id and the name given by the user; `fileName` is the name of the mat file where the dataset is stored; `subsample` is the requested percentage of the dataset, in the interval $[0, 1]$; and `varargin` are all the additional parameters requested by the method. The output must be a cell array of `Dataset` objects.

4.7 Designing a Partitioning Strategy

Partitioning strategies must be extend the `PartitionStrategy` class, and they must be placed inside the "*core/classes/PartitionStrategies*" folder. The constructor of a partition strategy can have any number of parameters. The strategy must implement four methods:

- `obj = partition(obj,Y)` is used to request a partition of the vector `Y`. Each partition can define any number of *splits* of `Y`, whose number must be stored in the property `num_folds`.

- `ind = getTrainingIndexes(obj)` and `ind = getTestIndexes(obj)` return the training and testing indexes of the current fold. The current fold can be retrieved from the property `current_fold`. Indexes are $N \times 1$ vectors of logical elements, where N is the dimension of the `Y` vector of the previous method.

- `s = getFoldInformation(obj)` returns a string with information on the current fold. This is used for printing information on the console during the simulation.

4.8 Designing a Statistical Test

A statistical test must extend the class `StatisticalTest`. It requires the implementation of two static methods:

- `[b, res] = check_compatibility(algorithms, datasets)` is used to check the compatibility of the procedure with the chosen datasets and algorithms. `b` is a boolean indicating whether there is or not compatibility. If `b` is false, `res` is a string describing the source of error.

- `perform_test(datasets_names, algorithms_names, errors)` performs the test and prints the information on screen. The first two arguments are cell arrays containing the names of the datasets and of the algorithms, while the third argument is a $A \times D$ matrix of errors, where A is the number of algorithms and D is the number of datasets. The ij-th element of `errors` is the averaged error of the i-th algorithm on the j-th dataset.

BIBLIOGRAPHY

[1] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang, "Extreme learning machine for regression and multiclass classification," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 42, no. 2, pp. 513–29, 2012.

[2] T. G. Dietterich, "Approximate statistical tests for comparing supervised classification learning algorithms," *Neural computation*, vol. 10, no. 7, pp. 1895–1923, 1998.

[3] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *The Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

[4] O. Chapelle, B. Schölkopf, A. Zien *et al.*, *Semi-supervised learning*. MIT press, 2006, vol. 2.

[5] E. Alpaydin, *Introduction to machine learning*. MIT press, 2004.