LYNX
User Manual

A MATLAB Toolbox for Fast Prototyping of Machine Learning Simulations

SIMONE SCARDAPANE

CONTENTS

INTRODUCTION

What is Lynx ?

Lynx is a research-oriented MATLAB toolbox for supervised and semi-supervised learning. It is aimed at making the comparison of multiple learning algorithms *fast* to implement, *simple* to modify, and easily *repeatable* by others.

Its main idea is summarized in Fig. 1. Instead of writing yourself the code to organize the comparison, you can specify its details in a human-understandable configuration file, which is loaded by Lynx. At this point, the toolbox takes charge of everything else: importing the requested datasets, partitioning them, running the algorithms, collecting the results and analyzing them. Moreover, 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.

In the configuration file, you can specify every detail of the simulation, including:

- the learning algorithms to use in the simulation,
- which performance measures to compute (e.g. mean-squared error, ROC curves, etc.),
- how to partition the data,
- additional preprocessing steps for the datasets, or features for the algorithms (e.g. feature selection procedures),
- enabling GPU support, saving the results, running a statistical test, and many other functionalities.

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

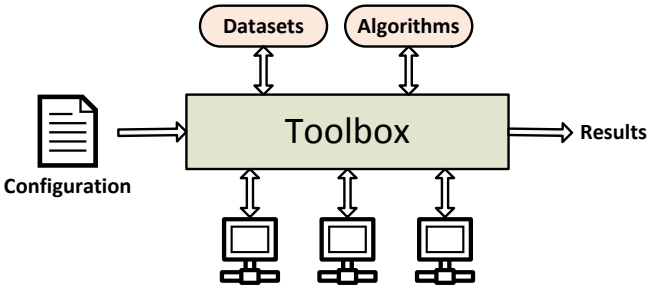

Figure 1: Schematic representation of the toolbox

How do I read this manual?

The strengths of Lynx, we hope, are more easily shown than said. Before starting, however, chapter 1 summarizes briefly the essential concepts of supervised learning. Although most readers will be highly familiar with these, we use the chapter to introduce a small set of definitions around which the toolbox is organized, so we advise even the most experienced reader to a quick glimpse at the chapter.

Following this, chapter 2 details how to install the toolbox, and it contains a step-by-step guide to run two initial simulations. Then, chapter 3 and chapter 4 detail how to write your own configuration files, starting from the basic instructions (e.g. adding a dataset) up to the most advanced functionalities (e.g. parallelizing the experiments). Finally, it is time to develop your own classes: chapter 5 and chapter 6 present guided examples for implementing everything, including new learning algorithms, new performance measures, and even new learning tasks.

Additionally, we have included two appendices. Appendix A details experimental features we are working on, which have no support yet and that are expected to change rapidly over the course of the next months. Then, appendix B concludes by detailing some of other features we plan on implementing. We use this also to mention what we believe are the current "weak" points of the toolbox, that will require further development in the future.

CHAPTER

ONE

A QUICK THEORETICAL BACKGROUND

In this chapter we introduce briefly some essential terminology for supervised learning. Clearly, a complete mathematical treatment of the field is beyond the scope of this manual, thus we go only briefly over most of the introduced ideas. We refer to any reference textbook for an in-depth treatment.[1, 2, 3]

1.1 The supervised learning problem

The starting point in supervised learning is a set S, called a *dataset*, composed of N pairs of the form:

$$S = \{x_i, y_i\}_{i=1}^{N} \tag{1.1}$$

where $x \in \mathcal{X}$ and $y \in \mathcal{Y}$. $\mathcal{X}$ and $\mathcal{Y}$ are called the *input* and *output* spaces, respectively. We call a particular choice of $\mathcal{X}$ and $\mathcal{Y}$ a *task*. Examples of tasks already implemented in Lynx are shown in Table 1.1. Although we have currently focused on real-valued input patterns, nothing prevents the toolbox from being extended to more complex situations, such as graphs, sequences, and so on.

We suppose that points in the two spaces are linked by an unknown probabilistic relationship, of which we only know the N samples in S. Each sample is also called an *example*, while each element $x \in \mathcal{X}$ is sometimes referred to as a *pattern*.

Informally, the goal of supervised learning is finding a suitable function $f : \mathcal{X} \to \mathcal{Y}$ such that, given any new sampling (x, y) of the unknown process, the expected error between $f(x)$ and y is minimized. Formally, a *learning algorithm* $\mathcal{A}$ can be defined as a mapping between the space $\mathcal{S}$ of all possible datasets and a given *hypothesis* of the unknown relation:

$$\mathcal{A}(S) : \mathcal{S} \to \mathcal{H} \tag{1.2}$$

Name	Input	Output
Regression		$y \in \mathbb{R}$
Binary Classification	$\mathbf{x} \in \mathbb{R}^d$	$y \in \{-1, +1\}$
Multiclass Classification		$y \in \{1, \ldots, M\}$

Table 1.1: Summary of some tasks defined in the toolbox

where $\mathcal{H}$ is called the *hypothesis space*, or *model space*. The notion of error is instead formalized by defining a *loss* function of the form:

$$L(S, f) : \mathcal{S} \times \mathcal{H} \to \mathbb{R} \tag{1.3}$$

An example of Eq. (1.3) is the *Mean-Squared Error* (MSE) given by:

$$L(S, f) = \sum_{(x,y) \in S} (f(x) - y)^2$$

In this framework, different learning algorithms differentiate themselves on the choice of the model space, or in the way in which a single hypothesis is extracted from $\mathcal{H}$ given a dataset. It is also possible to compute the performance of an algorithm in term of more complex measures, such as confusion matrices, ROC curves, correlation coefficients, and so on. In this case, we will talk more generally about *performance measures*.

An extension of this basic framework that we have implemented inside Lynx is the *semi-supervised* learning (SSL) framework.[4] In SSL, we suppose having available an additional unlabeled dataset of M input patterns $U = \{x_i\}_{i=1}^{M}$. An SSL algorithm uses this additional information trying to improve its own accuracy. Formally, if we denote by $\mathcal{U}$ the space of all possible unlabeled datasets, an SSL algorithm $\mathcal{A}_{\text{SSL}}$ is a mapping of the form:

$$\mathcal{A}_{\text{SSL}}(S, U) : \mathcal{S} \times \mathcal{U} \to \mathcal{H} \tag{1.4}$$

where the difference with Eq. (1.2) is the presence of the additional input U.

1.2 Some additional terminology

We call an *experiment* the test of a particular learning algorithm on a dataset. For a realistic workflow, refer to Fig. 1.1. The dataset, before being used, is processed by one (or more) *preprocessors*. A typical example of preprocessor is the *Principal Component Analysis* (PCA). The dataset is not used all for training, but it is partitioned in a *training set* and a *testing set*. The way in which data is subdivided is called a *partitioning strategy*. A partitioning strategy can also define multiple splits for a single dataset (e.g., k-fold cross-validation procedures): in this case, each split is called a *fold*. Finally, in Fig.

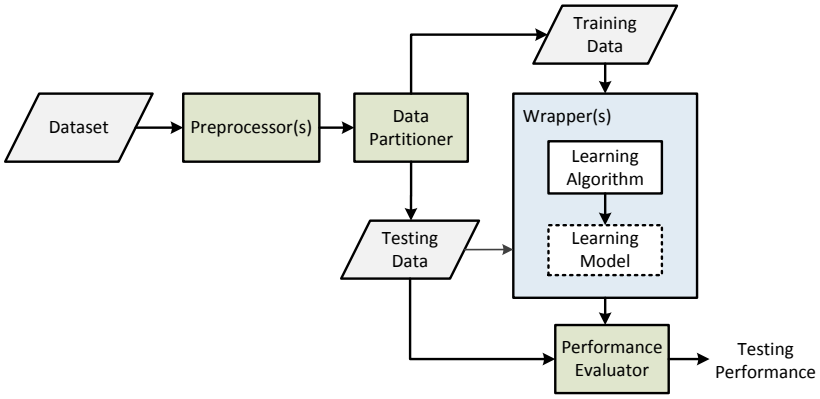

Figure 1.1: Workflow of a single experiment

1.1 the learning algorithm is encapsulated inside a generic "*wrapper*" object, that we define to be any method performing some additional optimization over the training procedure or the resulting model. This rather general definition includes techniques for optimizing the parameters of the algorithm, selecting a suitable subset of features, or training an ensemble of classifiers [1].

CHAPTER
TWO

INSTALLATION AND FIRST SIMULATIONS

In this chapter, we show how to install the toolbox and run some initial simulations.

2.1 Installing Lynx

After uncompressing the toolbox, run the `install.m` script in the root folder. At the end of the process, you will be asked to save the current path. If you select *no*, you will need to run again the installation script after rebooting MATLAB. Moreover, you will need to run again the script every time you move the toolbox to a different folder. Below is a transcript of a successful installation procedure:

```
Creating required folders...
Adding toolbox to path...
Generating configuration file...
--- Do you want to save the path? (Y/N) ---
Y
Saving path...
Installation complete

A few useful commands:

   --> To run a simulation: 'run_simulation'
   --> To generate HTML documentation: 'generate_documentation'
   --> To generate HTML reports: 'info_datasets', 'info_models',
      'info_wrappers', 'info_preprocessors'
   --> To run the test suite: 'run_test_suite'
   --> To clean the installation: 'clean_installation'
```

You can also see some of the main commands of the toolbox. We explore them in the rest of the chapter.

2.2 Folder Structure

Before continuing, let us look briefly at the folder structure of Lynx after the installation process. In Fig. 2.2 we show a unix-like representation of the directories, together with a brief comment on their contents.

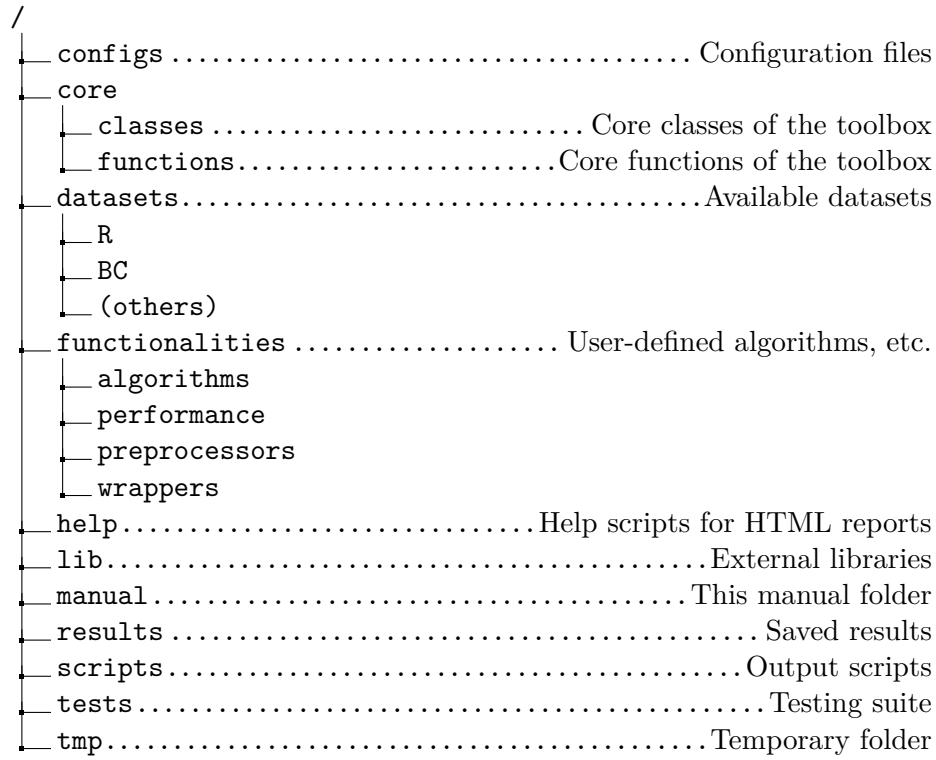

Figure 2.1: Folder Structure of the Toolbox

Most of the directories contents will be explored in depth in the rest of this manual. For the moment, you should remember that configuration files are stored in the "*configs*" folder.

2.3 Running a first simulation

To execute the first simulation, run the `run_simulation.m` script. You will see a list of the available configuration files, which after installation corresponds to the two demos included with the toolbox. For the moment, select "*demo_basic*" (i.e., select 1):

```
Initializing simulation...

Please select a configuration file:

        * [1] demo_basic - Basic demo of the toolbox
        * [2] demo_extended - Extended demo of the toolbox

Input your selection: 2
```

2.3.1 Initialization phase

You will be prompted with a detail of the simulation. In particular:

- The demo has two algorithms, a simple *baseline* to compare the results, and a more complex *Extreme Learning Machine* (ELM) [5]:

  ```
  Algorithms included in the simulation (2 total):
    * Baseline
      --> BaselineModel, trained with BaselineTrainingAlgorithm
    * Extreme Learning Machine
      --> ExtremeLearningMachine, trained with RegularizedELM
  ```

- The algorithms are tested on two datasets, *Gtzan*[1] (a binary classification dataset for music/speech discrimination) and a regression dataset taken from the UCI repository[2]:

  ```
  Datasets selected for the simulation (2 total):
    * Gtzan (Binary classification)
    * Glass (UCI) (Regression)
  ```

- Lynx will use the misclassification error for the classification tasks and the mean-squared error for the regression tasks.

- The simulation is run a single time, and data is partitioned using a k-fold cross-validation strategy:

  ```
  Will repeat experiments 1 time(s) with partitioning KFoldPartition
  ```

If you wish, you can compare this detail with the commands contained in the configuration file. This, however, will be explored in full depth in the next chapter. When you want to start the simulation, select *yes*.

[1] `http://marsyas.info/download/data_sets/`
[2] `http://archive.ics.uci.edu/ml/`

2.3.2 Training phase

In this phase, the algorithms are compared in turn on each dataset. In this case, both datasets are rather small, so the simulation will take a very short time. Below is an excerpt from this phase:

```
Evaluating Extreme Learning Machine on Glass (UCI) (run 1/1)
   Using k-fold partition 1 of 3... (143 training samples,
      71 testing samples)
   Using k-fold partition 2 of 3... (142 training samples,
      72 testing samples)
   Using k-fold partition 3 of 3... (143 training samples,
      71 testing samples)
```

For slower simulations, the percentage of completed experiments is shown in the bottom left of the MATLAB interface.

2.3.3 Output phase

At the end, the results are shown on screen. You can see, for example, that ELM has a misclassification error on Gtzan of 22.67%, against the baseline error of 48.52%. As expected, training times are all very small:

```
Primary performances:
                        Baseline  Extreme Learning Machine
Gtzan         48.52% (+/- 7.84%)         22.67% (+/- 2.96%)
Glass (UCI)      4.41 (+/- 0.08)            1.28 (+/- 0.31)

Training times:
                                 Baseline   Extreme Learning Machine
Gtzan         0.02 secs (+/- 0.01 secs)  0.01 secs (+/- 0.00 secs)
Glass (UCI)   0.01 secs (+/- 0.00 secs)  0.03 secs (+/- 0.03 secs)
```

2.4 Running a second simulation

Let us run a more complex simulation. To this end, execute once again `run_simulation.m`, but this time select the "*demo_extended*" configuration. You can see the following:

- We have added an additional algorithm to the simulation, a Support Vector Machine (SVM), trained with MATLAB own algorithm. Moreover, you can see that we added a wrapper to the ELM (*ParameterSweep*). This will fine-tune a parameter with a grid search procedure.
- We have also added a third and fourth datasets.

- This time, Lynx will compute also the ROC curve for classification tasks, and the mean-absolute error for regression tasks:

```
The following performance measures have been selected:
         * Binary classification: Misclassification rate
                 (and ROC curve)
         * Regression: Mean squared error
                 (and Mean absolute error)
```

- We have added two additional features: we execute a custom output script (in this case, for printing information on the grid search), and we save the results in a folder:

```
* The following features are active:
  --> SetSeedPRNG (Fix the seed of the PRNG)
  --> ExecuteOutputScripts (Execute 1 custom output scripts)
  --> SaveResults (Save results of the simulation in folder demo_results)
```

Moreover, you can see before the detail of the simulation the following warning:

```
The following tests are not possible:
   SVM on Glass (UCI).
   SVM on Yacht (UCI).
```

MATLAB SVM algorithm does not support regression, but only classification. Hence, Lynx warns the user that it cannot be executed on the two regression datasets.

After running the simulation, you will see (as expected) the ROC curve for the Gtzan and double moons datasets, and the mean-absolute error for the other two. Moreover, you will see some information about the grid search procedure. Finally, if you go in the *results/demo_results* folder, you will find a transcript of the simulation, a .mat file with the results, and the corresponding configuration file.

Let us analyze briefly the results. You can see that two of the three algorithms are shown in the plots with a single point: this is due to the fact that they do not output confidence scores in their classification. On the double moons dataset, the SVM has a perfect result, while it is worse on the Gtzan dataset. Finally, the optimal regularization parameter for ELM is around 1 and 3 on the different datasets:

```
Results of grid search for algorithm ELM:
  Dataset Gtzan:
    Average training time is 0.00 sec
    C = 3.000977
  Dataset Double moons:
    Average training time is 0.00 sec
```

```
    C = 2.667643
  Dataset Glass (UCI):
    Average training time is 0.00 sec
    C = 1.667643
  Dataset Yacht (UCI):
    Average training time is 0.00 sec
    C = 3.334310
```

2.4.1 Making the simulation more complex

We saw that the internal SVM of MATLAB does not support regression. Moreover, it does not return confidence scores in its classifications, so it is represented by a single dot in the ROC curve. To change this, open the configuration file, and uncomment line 11:

```
% Uncomment for LibSVM
set_training_algorithm('SVM', @LibSVM);
```

In this way, we will use the LibSVM library[3] to train the SVM. If you run the simulation again, Lynx will warn you that LibSVM is not currently installed, and (after confirmation) proceeds to the installation:

```
Checking prerequisites...
Checking for presence of LibSVM...
--- LibSVM not found. Do you want to install it? (Y/N) ---
(Required for: LibSVM training algorithm)
Y
Downloading toolbox... (may take some minutes)
Installation of LibSVM succesfull
```

After installing it, the simulation will run as before, but with the different training algorithm the SVM will be tested on all datasets. A warning will tell you that the results are being saved in a non-empty folder, and they are effectively overriding the previously saved results.

If you want, you can also uncomment line 30 to enable a statistical test of the results:

```
% Uncomment for statistical test (requires LibSVM training algorithm↩
    )
add_feature(CheckSignificance(FriedmanTest()));
```

You will see the statistical test just before the output script on the grid search:

[3] http://www.csie.ntu.edu.tw/~cjlin/libsvm/

```
------------------------------------------------
--- STATISTICAL ANALYSIS -----------------------
------------------------------------------------

Table of ranks:
              Baseline  ELM  SVM
Gtzan                3    2    1
Double moons         3    2    1
Glass (UCI)          3    2    1
Yacht (UCI)          3    2    1


Mean ranks:
  Baseline  ELM  SVM
         3    2    1

There is significant difference at alpha = 0.05 using the corrected
   Friedman test (Ff = Inf > 5.143253)

The following pairs have significant differences according to the
  Nemenyi test at alpha = 0.05 (CD = 1.656751):
        Baseline and SVM [rank difference = -2.000000]
```

2.5 Additional scripts

2.5.1 Downloading new datasets

The toolbox comes with a few datasets to start your simulations. To download more of them, run `download_datasets.m`. The process is fully automated, but it requires a working Internet connection.

2.5.2 Generating the documentation

If you want, you can generate a full HTML documentation for the files in the toolbox by running the `generate_documentation.m` script. The first time you run it, it will download the M2HTML toolbox, and it will create the documentation in the "*doc*" folder.

2.5.3 Running the test suite

Lynx comes with a suite of approximately 150 unitary tests. To execute the suite, run the `run_test_suite.m` command. Be careful to run this command again after every change to its internal code.

2.5.4 Cleaning the installation

Finally, you may want to clean the installation, i.e. remove everything from the path, deleting the external libraries etc. To this end, run the `clean_installation.m` script.

CHAPTER
THREE

WRITING A CONFIGURATION FILE

In this chapter, you will learn the basics for writing a configuration file. We focus on the essential elements, i.e. adding models, datasets, wrappers and preprocessors. For additional features, refer to the next chapter.

3.1 Adding a dataset

Datasets are stored in the "*datasets*" folder of the toolbox. They are organized in folders corresponding to the different tasks. To get a summary of available tasks and datasets, run the `info_datasets.m` script.

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

```
add_dataset(id, name, filename);
```

where:

- `id` is a string identifying the dataset in the configuration file,
- `name` is a name for the dataset, to be displayed in the results,
- `filename` is the unique alphanumerical string denoting the dataset in the filesystem.

As an example, the call:

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

adds the dataset "*uci_yacht*" to the simulation with name "Yacht".

3.1.1 Adding a multilabel dataset

You may have seen during the simulation that some tasks have no performance measure assigned to them (e.g., multilabel classification tasks). These may be taught as "*incomplete*" tasks, i.e., tasks for which no training algorithms are available. Currently, they are handled by transforming them (after loading the datasets) into other, more "basic" tasks. In particular:

- Prediction tasks are transformed into regression tasks by embedding the timeseries into a phase space.
- Multilabel classification tasks are transfomed into multiple binary classification tasks (the so-called *binary relevance* method).

As an example, try to add a multilabel dataset to a simulation:

```
add_dataset('C', 'Cal 500', 'cal500_mood);
```

You can see that, during initialization, the 36 labels in the dataset are transformed into 36 different binary classification datasets:

```
Extracted 36 different binary classification datasets from original
  dataset Cal 500
```

The 36 tasks have sequential ids given by C-1, C-2, etc. Similarly, try to add a prediction dataset:

```
add_dataset('M', 'Mackey-glass', 'mackeyglass');
```

During initialization, this is transformed into a regression dataset:

```
Embedded dataset Mackey-glass, 4993 samples extracted
```

How to "complete" these tasks, and how to design new ones, are topics explored in the last chapter of the manual.

3.2 Adding a model

A model can be added to the simulation with the following syntax:

```
add_model(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 of the model.

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_model('SVM', 'Support Vector Machine', @SupportVectorMachine);
```

adds a default-initialized Support Vector Machine to the simulation. Similarly, the call:

```
add_model('SVM', 'Support Vector Machine', @SupportVectorMachine, '↩
    kernel_type', 'lin');
```

adds a Support Vector Machine with a linear kernel.

A concise HTML report with a list of all implemented models and respective parameters can be found by calling the script `info_models.m`. Additional information on each model can be found by visualizing the help for the respective class.

3.2.1 Changing the training algorithm

Each model has a default training algorithm, that can be changed with the `set_training_algorithm` function. For example, to change the training algorithm of the previously defined Support Vector Machine to LibSVM:

```
set_training_algorithm('SVM', @LibSVM);
```

Available training algorithms are listed in the `info_models.m` script.

3.3 Adding a preprocessor

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 should 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', @PrincipalComponentAnalysis, '↩
    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 `info_preprocessors.m`. Additional information on each preprocessor is given by the help of the corresponding class.

3.3.1 Preprocessors and Multi-label Tasks

The fact that the syntax for adding a preprocessor accepts a regular expression is particularly suited for multi-label tasks. Continuing the example of Section 3.1.1, suppose we now want to perform a PCA on all 36 binary classification tasks. This can be done with a single call:

```
add_preprocessor('C-.', @PrincipalComponentAnalysis, '↩
    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.

3.4 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);
```

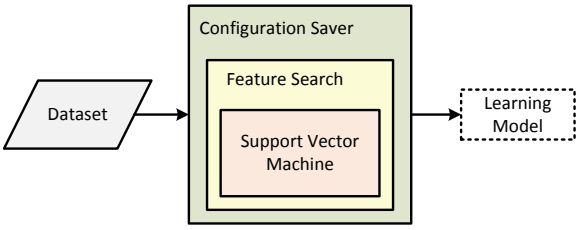

Figure 3.1: Example of piling two different wrappers

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('SVM', @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 `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. As an example, consider the following code:

```
add_wrapper('SVM', @Featuresearch_GA);
add_wrapper('SVM', @SaveConfiguration);
```

In this case, the first wrapper searches for the optimal feature subset using a genetic algorithm, while the second wrapper saves the resulting configuration, as shown in Fig. 3.1.
The wrappers act at different moments during the learning process: the first one searches the subset before training the final SVM, while the second wrapper saves the configuration after the final training. Other wrappers can have different effects also in the testing phase, allowing for the creation of complex behaviors. For example, an unfolded version of Fig. 3.1 is shown in Fig. 3.2.

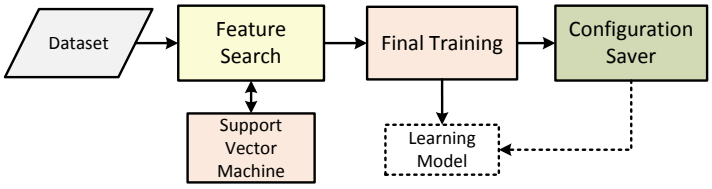

Figure 3.2: Same as Fig. 3.1, but with the actions of the wrappers unfolded in time.

3.4.1 Example 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 (between others) two tunable parameters, a regularization factor (denoted by `regularizationFactor`) and the number of hidden nodes (denoted by `hiddenNodes`). As is standard practice, we want to test the following range of values for the parameters:

$$2^{-5}, 2^{-4}, \ldots, 2^{5} \text{ for the regularization factor} \tag{3.1}$$

$$100, \ldots, 1000 \text{ for the number of hidden nodes} \tag{3.2}$$

Since we have two parameters, this results in $10 \times 9 = 90$ configurations to be tested. The following command generates the corresponding wrapper in our example:

```
add_wrapper('ELM', @ParameterSweep, {'regularizationFactor', '↩
    hiddenNodes'}, {2.^-5:5, 100:100:1000});
```

Although this may seem daunting at first, it is actually rather simple. The first parameter is a cell array with the names of the parameters to be tested. The second parameter is a cell array with the values to be tested. By default, this wrapper uses a 3-fold cross-validation on the training data to test the accuracy. We may change this with any object of class `PartitionStrategy`. For example, to execute an holdout partitioning with 30% of validation data:

```
add_wrapper('ELM', @ParameterSweep, {'regularizationFactor', '↩
    hiddenNodes'}, {2.^-5:5, 100:100:1000}, 'partition_strategy', ↩
    HoldoutPartition(0.3));
```

3.4.2 Example 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 90 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('SVM', @SaveConfiguration, 'ELM_SAVED');
```

After training a model, this will be saved in the "*./models/*" folder. Note that a simulation requires training several models, one for each fold, dataset and run. Hence, this will save several files in the folder. The naming convention is that a file is denoted as ID_DATASET_rRUNIDfFOLDID.mat, where ID is the id defined above (ELM_SAVED in our example), DATASET is the name of the dataset, RUNID is the numerical id of the run, and FOLDID 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, 'ELM_SAVED');
```

CHAPTER
FOUR

ADVANCED CONFIGURATION FEATURES

In this chapter, we explore all the remaining features to customize the simulation.

4.1 Changing partitioning and number of runs

By default, Lynx uses a 3-fold cross-validation to partition the data. You can change this using any `PartitionStrategy` object with the `set_partition_strategy` function. A few examples are:

- Set a 10-fold cross-validation:

```
set_partition_strategy(KFoldPartition(10));
```

- Set an holdout partition with 25% of testing data:

```
set_partition_strategy(HoldoutPartition(0.25));
```

- Set the same partition for training and testing:

```
set_partition_strategy(NoPartition());
```

  You can see all available partition strategies in the "*functionalities/-PartitionStrategies*" folder.

Moreover, you can repeat the experiments more than one time by calling `set_runs`:

```
set_runs(3); % Execute 3 times the experiments
```

4.2 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*. Moreover, 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, it is the user's duty to take care that Lynx is properly installed on all machines. 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.
```

4.2.1 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/`

4.3 Enabling semi-supervised training

The toolbox has 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_model('ELM', 'Extreme Learning Machine', @ExtremeLearningMachine↩
    );
set_training_algorithm('ELM', @LaplacianELM);
```

the ELM 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 calling the `set_semisupervised_partitioning` function:

```
set_semisupervised_partitioning(HoldoutPartition(0.6));
```

If the partition strategy has more than a single fold, the fist one will be used.

4.4 Adding a performance measure

You can add a performance measure to be computed with the following syntax:

```
add_performance(task, perf);
```

where `task` is the id of the task and `perf` the performance measure. For example, you can add the ROC curve for binary classification:

```
add_performance(Tasks.BC, ROCCurve());
```

Every task has a *primary* performance measure, which is used also to compare algorithms. If you want to change the primary performance measure of a task, you can set an additional boolean value:

```
add_performance(Tasks.R, MeanAbsoluteError(), true);
```

Not all performance measures can be set as primary. In particular, they must be at least comparable. For example, the ROC curve cannot be set as primary.

4.5 Other features

Additional features can be added with the `add_feature` function. We explore them in the rest of the chapter.

4.5.1 Enabling GPU support

An additional form of parallelization is given by the use of the GPU. This is enabled with the following call:

```
add_feature(GPUSupport());
```

This functionality requires the Parallel Computing Toolbox of Matlab and a supported NVIDIA GPU device. Additionally, the latest version of the CUDA drivers[1] has to be installed. The GPU compatibility is tested in the initialization phase:

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

If an algorithm has GPU support, the dataset is transferred to the GPU before training. 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.

4.5.2 Fixing the PRNG seed

To fix the seed of the PRNG (for repeatability) call:

```
add_feature(SetSeedPRNG(seed));
```

4.5.3 Adding 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:

```
add_feature(ExecuteOutputScripts(scrip1, script2, ...));
```

As an example, consider the parameter sweep detailed in Section 3.4.1. This is an example of running the simulation with the additional wrapper:

```
Testing ExtremeLearningMachine on Glass (run 1/1)
Fold 1...
 Validated parameters: [ C = 1.000000, hiddenNodes = 100.000000 ],
```

[1] https://developer.nvidia.com/cuda-downloads

```
  with error: 0.558518
 Final training time is: 0.003000
Fold 2...
 Validated parameters: [ C = 0.500000, hiddenNodes = 100.000000 100.000000 ],
  with error: 0.605529
 Final training time is: 0.004000
Fold 3...
 Validated parameters: [ C = 2.000000, hiddenNodes = 100.000000 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:

```
add_feature(ExecuteOutputScripts('info_gridsearch));
```

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
```

4.5.4 Running a statistical test

A statistical testing is requested by calling the following method:

```
add_feature(CheckSignificance(test));
```

where `test` 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 text requires exactly two algorithms and at least two datasets. So, if we add it to a simulation like:

```
add_feature(CheckSignificance(WilcoxonTest()));;
```

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

4.5.5 Saving the results

The results of the simulation can be saved with the following syntax:

```
add_feature(SaveResults(folder));
```

They are saved inside a the folder "*results/folder*'. Three 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).
- The configuration file that has been used.

CHAPTER
FIVE

PROGRAMMING IN LYNX

In this chapter, we show how to design new datasets, models, training algorithms, wrappers and preprocessors.

5.1 Storing a new 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. For the tasks currently implemented in Lynx, 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. For regression, each value is real. For binary classification, each element can take the values -1 or $+1$. Finally, for multiclass classification, each element can take a value in $\{1, \ldots, M\}$, where M is the number of classes.

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:

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.

5.2 Parameterized classes

To make uniform the way in which classes accept parameters, most functionalities in the toolbox (training algorithms, wrappers, preprocessors, etc.) derive from a common abstract class called `Parameterized`. If you inspect its code, you will see that its constructor does the following:

1. It creates an `InputParser` object[1].
2. It calls the abstract method `initParameters` to fill the object with the required parameters.
3. It parses the parameters in input, and it saves them in the `parameters` struct.

Any class implementing `Parameterized` must define the `initParameters` method to specify its internal parameters. It can access them using the `getParameter` method, and change them using the `setParameter` one. Moreover, the class specifies a few static method describing the parameters: `getDescription`, `getParametersNames`, `getParametersDescription`, `getParametersRange`. In this way, all the class in the toolbox have an uniform way of describing their own parameters. Moreover, the `getParameter` and `setParameter` can be overridden to implement a more complex behavior. For example, any wrapper can use the methods to access any property of their internal algorithm.

5.3 Implementing a new model

In this section we show how to implement a new model with a step-by-step example. In particular, we implement a simple K-Nearest Neighbors (KNN) algorithm [1], with the possibility of setting K. We need to define two classes: `KNearestNeighbor`, deriving from `Model`, describing the model, and `SimpleKNN`, deriving from `LearningAlgorithm`, for training it.

Let us start from the first one. We define a skeleton code with two properties for storing the input and output patterns seen during training:

```
classdef KNearestNeighbor < Model

    properties
        trainingInputs;
        trainingOutputs;
    end

    methods
        function obj = KNearestNeighbor(id, name, varargin)
            obj = obj@Model(id, name, varargin{:});
```

[1] http://www.mathworks.it/it/help/matlab/ref/inputparser-class.html

```
        end

        ...
    end
end
```

Note that the constructor of the class simply calls the `Model` constructor.

5.3.1 Setting the Training Parameters

We need to define the abstract methods of the `Parameterized` superclass described in Section 5.2. In our case we only have a single training parameter, so the `initParameters` method is very simple:

```
function p = initParameters(~, p)
   p.addParamValue('K', 3, @(x) assert(isnatural(x), 'Lynx:Runtime:↩
       Validation', 'K must be an integer > 0'));
end
```

We add a single parameter to the `InputParser` object `p`, and we require it to be a non-negative integer. The `isnatural` is a validation function provided in the toolbox. Example of usage of this class are:

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

We also need to define a few static methods. `getDescription` returns a string describing the model:

```
function s = getDescription()
   s = 'K-Nearest Neighbor algorithm';
end
```

`getParametersNames` returns a cell array with the names of the training parameters:

```
function s = getParametersNames()
   s = {'K'};
end
```

`getParametersDescription` returns a cell array with a description of the training parameters:

```
function s = getParametersDescription()
   s = {'Number of neighbors to consider'};
end
```

Finally, `getParametersRange` returns a cell array with the possible values for the training parameters:

```
function s = getParametersRange()
   s = {'Non-negative natural number, defaults to 3'};
```

```
end
```

5.3.2 Testing

The main function of a `Model` class is a `test` method for getting the predictions of the model on a particular set of input patterns. We are guaranteed that this is always called after training the algorithm with a suitable training method. Here is an example implementation in our case:

```
function [labels, scores] = test(obj, Xts)
   neighbors = knnsearch(obj.trainingData, Xts);
   N_testing = size(Xts, 1);
   labels = zeros(N_testing, 1);
   scores = zeros(N_testing, 1);
   for ii = 1:size(Xts, 1)
      if(obj.getCurrentTask() == Tasks.R)
         labels(ii) = mean(obj.trainingOutputs(neighbors(ii, :)));
      else
         labels(ii) = mode(obj.trainingOutputs(neighbours(ii, :)));
      end
      scores(ii) = labels(ii);
   end
end
```

Here, `Xts` is a matrix of input test patterns. `labels` is 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. Note that, if the current task is regression, we use the mean values of the K closest neighbors, otherwise we use the most frequent value (for classification).

5.3.3 Other methods

We need to define a few more methods. First, a `getDefaultTrainingAlgorithm` that returns the default training algorithm (that we will implement in the next section):

```
function a = getDefaultTrainingAlgorithm(obj)
   a = SimpleKNN(obj);
end
```

Then, a function returning all the possible tasks that are allowed by the model:

```
function res = isTaskAllowed(~, t)
   res = (t == Tasks.R || t == Tasks.BC || t == Tasks.MC);
end
```

5.4 Implementing a training algorithm

Let us now implement the training algorithm for our KNN. Training algorithms derives from `LearningAlgorithm`, and they also derive from `Parameterized`. In our case, we have no training parameters:

```
classdef SimpleKNN < LearningAlgorithm
   function obj = SimpleKNN(model, varargin)
      obj = obj@LearningAlgorithm(model, varargin{:});
   end

   function p = initParameters(~, p)
   end

   ...
end
```

Because we have no training parameters, we can define simply the `getDescription` method:

```
methods(Static)
   function s = getDescription()
      s = 'Simple KNN training procedure';
   end
end
```

Now we need to implement a training method. In this case, we simply save the input matrix `Xtr` and output vector `Ytr` in the corresponding properties of the model:

```
function obj = train(obj, Xtr, Ytr)
   obj.model.trainingInputs = Xtr;
   obj.model.trainingOutputs = Ytr;
end
```

We also need to define a `checkForCompatibility` method to check that the algorithm is initialized with a `KNearestNeighbor` model:

```
function b = checkForCompatibility(obj, model)
   b = isa(model, 'KNearestNeighbor');
end
```

5.4.1 Other methods of a training algorithm

A `TrainingAlgorithm` object can have additional methods that we have not used in our example:

- `checkForPrerequisites` is used to check if the required libraries and toolboxes are installed. The `LibraryHandler` class provides two utility methods to this end.

- `hasCustomTesting` can be used to tell Lynx that the training algorithm needs a custom testing procedure different from the one defined in the corresponding model (e.g. in the case of a wrapper to an external library). In this case, the custom testing procedure can be implemented as the `test_custom` method.

- `hasGPUSupport` is used to tell Lynx that the training algorithm supports GPU acceleration. If GPU support is enabled in a simulation, the training values will be passed to the algorithm as `gpuArray`[2] objects.

5.4.2 Implementing a Semi-Supervised Algorithm

Implementing a semi-supervised algorithm follows the same guidelines as before. The only differences are that the algorithm should derive from `SemiSupervisedLearningAlgorithm` instead of `LearningAlgorithm`, and that the training method has a different signature:

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

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

5.5 Implementing a Wrapper

In this section we show how to implement a new wrapper by implementing a `SubsampleTrainingData` wrapper to subsample the training data of a user-specified factor d.

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.

Let us begin to write our code:

```
classdef SubsampleTrainingData < Wrapper
    methods
        function obj = SubsampleTrainingData(wrappedAlgo, varargin)
           obj = obj@Wrapper(wrappedAlgo, varargin{:});
        end

        ...
    end
end
```

We define the training parameters:

```
function p = initParameters(~, p)
```

[2] `http://www.mathworks.it/it/help/distcomp/gpuarray.html`

```
    p.addParamValue('d', 0.5);
end
...
methods(Static)
  function info = getDescription()
    info = 'Subsample the training data';
  end

  function pNames = getParametersNames()
    pNames = {'d'};
  end

  function pInfo = getParametersDescription()
    pInfo = {'Subsampling percentage'};
  end

  function pRange = getParametersRange()
     pRange = {'Real number in [0, 1], default 0.5'};
  end
end
```

Now we define the training method:

```
function obj = train(obj, Xtr, Ytr)
   p = cvpartition(Ytr, 'holdout', obj.parameters.d);
   obj.wrappedAlgo = obj.wrappedAlgo.setCurrentTask(obj.↩
       geCurrentTask());
   obj.wrappedAlgo = obj.wrappedAlgo.train(Xtr(training(p), :), Ytr(↩
       training(p));
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 to 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` overrides `getTrainingParam` and `setTrainingParam` to this end. As an example, suppose we want to access property "`C`" of our base algorithm. This is done as:

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

Then, we can increment it by one with:

```
obj.wrappedAlgo = obj.setParameter('C', C + 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 a 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.generateSinglePartition(p);
```

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

```
perfs = PerformanceEvaluator.computePerformance(obj.wrappedAlgo, d);
```

`perfs` is a cell array containing all the performance values that were selected in the simulation for the given task. To access the primary measure, we can select the first element of the cell array. If we have two performance measures `p1` and `p2`, we can compare them using the `isBetterThan` method of `PerformanceMeasure`:

```
b = p1.isBetterThan(p2);
if(b)
  // p1 is better
else
  // p2 is better
end
```

5.6 Implementing a Preprocessor

The `SubsampleTrainingData` can also be implemented as a preprocessor to sample the original dataset.The class must derive from `Preprocessor` instead of `Wrapper`, but is has the same parameter `d`. Instead of the training method, however, we define a `process` method, which takes a `Dataset` object as single parameter, and returns the transformed dataset:

```
function d = process(obj, d)
   p = cvpartition(d.Y, 'holdout', obj.parameters.d);
   d.X = d.X(training(p), :);
   d.Y = d.Y(training(p));
end
```

We also need to define a `processAsBefore` method, which should process a dataset using the same settings resulting from a previous call to `process`. This is needed for using the preprocessor as a wrapper using the `ApplyPrepreprocessor` wrapper (see its help). In this case, we can simply call the `process` method:

```
function d = processAsBefore(obj, d)
   d = obj.process(d);
end
```

CHAPTER
SIX

ADVANCED PROGRAMMING IN LYNX

In this chapter we describe how to design advanced functionalities in Lynx.

6.1 Performance measures

6.1.1 Understanding containers

Before implementing a performance measure, you need to understand the structure of the `ValueContainer` object. A `ValueContainer` is an object used to store several elements of a particular type (e.g. numbers or percentages), and to return an "average" value on request. Every performance measure should derive from the abstract class `PerformanceMeasure`, and from a suitable `ValueContainer` object. For example, the mean-squared error derives from `NumericalContainer`; the misclassification error derives from `PercentageContainer`; the ROC curve derives from `XYPlotContainer`.

6.1.2 Implementing performance measures

The main method of a performance measure is the `compute` method:

```
perf = compute(obj, true_labels, predictions, scores);
```

Here, `true_labels` are the targets values, `predictions` are the predictions of a model, and `scores` are the confidence scores. `perf` is the output value of the performance measure, which should be compatible with the chosen `ValueContainer` object. For example, a class deriving from `NumericalContainer` should output numerical values; a class deriving from `XYPlotContainer` should output `XYPlot` objects; and so on. We also need to implement two additional methods:

- `isCompatible(t)` checks that the performance measure is compatible with the task `t`.

- `isBetterThan(obj, p))` checks if the current performance measure is better than performance measure `p`.

Moreover, performance measures that are not comparable should set the internal property `isComparable` to false in the constructor.

6.2 Partition strategies

Partitioning strategies must be extend the `PartitionStrategy` class, and they can be placed inside the "*functionalities/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.

6.3 Statistical tests

A statistical test must extend the class `StatisticalTest`. It requires the implementation of three 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.
- `s = getDescription` returns a string describing the statistical test.

6.4 Tasks and factories

Any task in Lynx derives from `BasicTask`. Loading a dataset in Lynx proceeds as follows:

- Every task has an associated set of folders that contains datasets of the corresponding type. Starting from the filename, the dataset is searched from all the basic tasks, and associated to the first task containing a .mat file with the corresponding name.
- The .mat file is checked for consistency using the `checkForConsistency` method of the corresponding class.
- One or more datasets are created using a `DatasetFactory` object.

For example, `RegressionTask` defined a regression task, which is loaded using the `DummyFactory` object. The `PredictionTask`, instead, is loaded using the `EmbedTimeseriesFactory`, that transform it into a regression task by embedding the timeseries. You can have more than one factory associated to a particular class, and you can change them in the configuration file using the `set_factory` method.

In the constructor of a task, you must define three properties:

- `folders` is a cell array of folders containing datasets of this type.
- `performance_measure` is the default performance measure for the task (empty if not defined).
- `dataset_factory` is the default `DatasetFactory` object associated to the task.

Together with the `checkForConsistency` method, you must define two additional methods:

- `s = getDescription(obj)` returns a string describing the task.
- `id = getTaskId(obj)` returns the id of the task, which should be contained in the `Tasks` enumeration.

Implementing a `DatasetFactory` object, instead, consists in a single method:

```
datasets = create(obj, task, dID, data_name, o);
```

where `task` is the task id, `dID` and `data_name` are the id and name given to the dataset by the user, and `o` is a structure array with the content of the .mat file. `datasets` is a cell array of `Dataset` objects.

6.5 Additional features

Additional features can derive from the abstract `AdditionalFeature` class. They can implement one or more of its methods, that act at predefined moments during the simulation. For additional information, see the help of `AdditionalFeature`.

APPENDIX
A

EXPERIMENTAL FEATURES

Experimental features are currently under active development. They are expected to change rapidly, and they have no full support.

A.1 Hierarchical learning algorithms

A hierarchical learning algorithm (HLA) can be any tree structure of models, where an input is propagated through the tree until a final decision is taken on a leaf node. At every node, a learning algorithm is employed to choose which branch to follow. To understand its utility, try to plot the output values of the *uci_yacht* dataset: you will see that there is a strong gap between positive and negative values. In this case, it may be better to envision a two-stage architecture:

- The first level discriminates between positive and negative output values.
- The second level has two learning algorithms trained only on positive and negative outputs respectively.

Currently, you can implement this with the following code:

```
add_model('NODE', 'Internal node', @ExtremeLearningMachine);
add_model('LEAF1', 'First leaf', @ExtremeLearningMachine);
add_model('LEAF2', 'Second leaf', @ExtremeLearningMachine);

construct_hierarchical_algorithm([2 0 0], {RangeAggregator(0)}, {'↩
    NODE', 'LEAF1', 'LEAF2'});

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

The parameters to the `construct_hierarchical_algorithm` method are:

- A cell array describing the number of childs of the tree, in a depth-first fashion.

- A cell array of `Aggregator` objects. Aggregators are used to associate to every original output a particular branch of the node. In our case, `RangeAggregator` discriminates between negative and positive values, sending them to the left and right branches of the node.
- A cell array containing the ids of the models composing the tree.

We can also add an output script for plotting the tree structure:

```
add_feature(ExecuteOutputScripts('info_hla'));
```

APPENDIX
B

FUTURE DEVELOPMENTS

Here we list briefly some of the developments that we are working on. We also discuss a few suboptimal features of the current version of the toolbox.

B.1 Data types

Currently, we distinguish between *tasks*, but not between *types* of data (e.g. real-valued vectors, graphs, sequences, etc.). We plan on including this distinction in the future.

B.2 Load balancer

Experiments are distributed throughout the workers in a fully random way. We plan on implementing a more efficient load balancer, that should take into consideration the average training time of the algorithms, together with the characteristics of the hardware.

B.3 Web service

We would like to implement a web service where to expose additional functionalities developed by the users, that can be downloaded automatically from the toolbox itself.

BIBLIOGRAPHY

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

[2] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, 2nd ed. Springer, 2009.

[3] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning.* MIT Press, 2012.

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

[5] 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.