GURLS^{++} User's Guide

1 Design

In supervised learning, the building blocks of a learning experiment are its phases or *processes* (typically the training process and the testing process), that can be run on different data sets (typically the train and test set). What characterizes GURLS is the idea that processes within the same experiment share a common (ordered) sequence of *tasks* which we call the learning *pipeline*. Each GURLS process differs from the others on how each task is performed (e.g. compute or load previously computed results) and on the data used as input.

GURLS basically consists of a set of tasks, each one belonging to a predefined category, and of a method called GURLS *Core*, implemented through the `GURLS` class, that is responsible for processing the task pipeline. An additional "options structure", often referred to as *OPT*, is used to store all configuration parameters needed to customize the tasks behaviour. Tasks receive configuration parameters from the options structure in read-only mode and, after terminating, their results are appended to the structure by the GURLS Core in order to make them available to the subsequent tasks. This allows the user to easily skip the execution of some tasks in a pipeline, by simply inserting the desired results directly into the options structure. All tasks belonging to the same category can be interchanged with each other, so that the user can easily choose how each task shall be carried out. A schema of the design and execution of a GURLS process is shown in Figure 1.

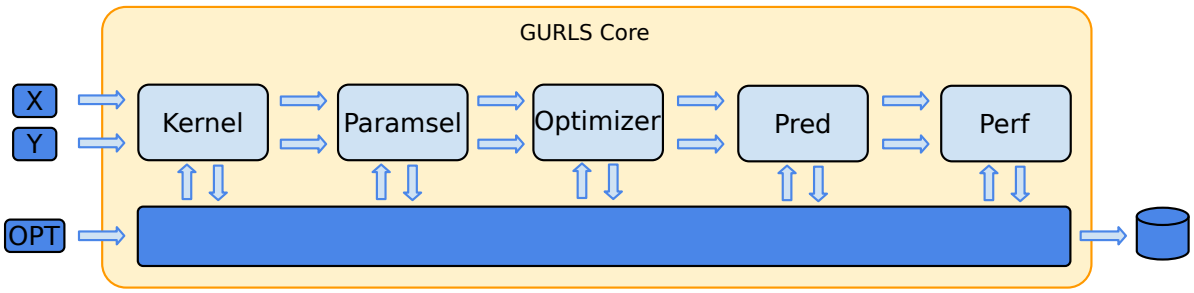

Figure 1: GURLS process.

2 GURLS^{++} main classes

2.1 The `GURLS` class

The `GURLS` class implements the GURLS *Core*. Its only method `run` runs the learning pipeline and is the main method the user would directly call. It accepts exaclty four arguments:

1. the $N \times D$ input data matrix (N is the number of samples, D is the number of variables),

2. the $N \times T$ labels matrix (T is the number of outputs. For (multi-class) classification, labels $+1$ and -1 must be in the One-Vs-All format).

3. An options' structure.

4. A job-id number.

Each time the data need to be changed (e.g. going from training process to testing process) `GURLS.run` needs to be invoked again.

2.2 The `GurlsOptionsList` class

The options' structure is built through the `GurlsOptionsList` class with default fields and values. The three main fields in the options' structure are:

- `name`: identifies the file where results shall be saved.

- `seq`: specifies the (ordered) sequence of tasks, i.e. the pipeline, to be executed. Each task is defined by providing a task category and a choice amongst those available for that category, e.g. with `"optimizer:rlsprimal"` one sets the optimizer to be Regularized Least Squares in the primal space (see Table 1 to know the available categories and choices for each categories).

- `process`: specifies what to do with each task. Possible instructions are:

  - `ignore`
  - `compute`
  - `computeNsave`
  - `load`
  - `delete`

3 Examples in GURLS++

In the 'demo' directory you will find GURLS`loocvprimal.cpp`. The meaning of the demo is the following:

- For the training data: calculate the regularization parameter λ minimizing classification accuracy via Leave-One-Out cross-validation and save the result, solve RLS for a linear classifier in the primal space and save the solution. Ignore the rest.

- For the test data set, load the used λ (this is important if you want to save this value for further reference), load the classifier. Predict the output on the test-set and save it. Evaluate the average classification accuracy and as well as the precision-recall save them.

In the following we report and comment the salient part of the demo.

First load the data from file. The training data is assumed to be stored in two .csv files, xtr_file and ytr_file, and the test data in two other .csv files, xte_file and yte_file:

```
gMat2D<T> Xtr, Xte, ytr, yte;
Xtr.readCSV(xtr_file);
Xte.readCSV(xte_file);
ytr.readCSV(ytr_file);
yte.readCSV(yte_file);
```

then initialize an object of class `GURLS` and build an options' list and by assigning it a name, in this case `"Gurlslooprimal"`

```
GURLS G;
GurlsOptionsList* opt = new GurlsOptionsList("Gurlslooprimal", true);
```

specify the task sequence

```
OptTaskSequence *seq = new OptTaskSequence();
*seq << "paramsel:loocvprimal" << "optimizer:rlsprimal";
*seq << "pred:primal" << "perf:macroavg"<< "perf:precrec";
opt->addOpt("seq", seq);
```

initialize the process option

```
GurlsOptionsList * process = new GurlsOptionsList("processes", false);
```

and define instructions for the training process

```
OptProcess* process1 = new OptProcess();
*process1 << GURLS::computeNsave << GURLS::computeNsave;
*process1 << GURLS::ignore << GURLS::ignore << GURLS::ignore;
process->addOpt("one", process1);
opt->addOpt("processes", process);
```

and testing process

```
OptProcess* process2 = new OptProcess();
*process2 << GURLS::load << GURLS::load << GURLS::computeNsave;
*process2 << GURLS::computeNsave << GURLS::computeNsave;
 process->addOpt("two", process2);
```

run gurls for training

```
string jobId0("one");
G.run(Xtr, ytr, *opt, jobId0);
```

run gurls for testing

```
G.run(Xte, yte, *opt, jobId1);
string jobId1("two");
```

4 Further examples

The method `run` of class `GURLS` executes an ordered sequence of tasks, the *pipeline*, specified in the field `seq` of the options' structure as

```
{"<CATEGORY1>:<TASK1>";"<CATEGORY2>:<TASK2>";...}
```

These tasks can be combined in order to build different train-test pipelines. A list of the currently implemented GURLS tasks organized by category, is summarized in Table 1. In order to run the other examples you just have to substitute the code fragment for the task pipeline

```
*seq << ...
```

and for the sequence of instructions for the training process

```
*process1 << ...
```

and testing process

```
*process2 << ...
```

with the desired task pipeline and instructions sequence. In the following we report the fragment of code defining he task sequence and the training and testing instructions some popular learning pipelines.

4.1 Linear classifier, primal case, hold-out cv

```
*seq << "split:ho" << "paramsel:hoprimal" << "optimizer:rlsprimal";
*seq << "pred:primal" << "perf:macroavg";

*process1 << GURLS::computeNsave << GURLS::computeNsave << GURLS::computeNsave;
*process1 << GURLS::ignore << GURLS::ignore;

*process2 << GURLS::load << GURLS::load << GURLS::load;
*process2 << GURLS:: computeNsave << GURLS:: computeNsave;
```

4.2 Linear regression, primal case, hold-out cv

For regression the option `hoperf` in the option structure must be manually set to Root Mean Square Error (RMSE), otherwise average classification accuracy is used as performance measure for parameter selection

```
OptString* hofun = new OptString("rmse");
opt.addOpt("hoperf", hofun);
```

Then the pipeline to be used is the following

```
*seq << "split:ho" << "paramsel:hoprimal" << "optimizer:rlsprimal";
*seq << "pred:primal" << "perf:rmse";

*process1 << GURLS::computeNsave << GURLS::computeNsave << GURLS::computeNsave;
*process1 << GURLS::ignore << GURLS::ignore;

*process2 << GURLS::load << GURLS::load << GURLS::load;
*process2 << GURLS:: computeNsave << GURLS:: computeNsave;
```

Note that the same pipeline can be used for multiple output regression.

4.3 Gaussian Kernel, dual case, hold out cross validation

```
*seq <<"split:ho"<<"paramsel:siglamho"<<"kernel:rbf"<<"optimizer:rlsdual";
*seq <<"pred:dual"<<"predkernel:traintest"<<"perf:macroavg";

*process1 <<GURLS::computeNsave<<GURLS::computeNsave<<GURLS::computeNsave;
*process1 <<GURLS::computeNsave<<GURLS::ignore<<GURLS::ignore<<GURLS::ignore;

*process2 <<GURLS::load<<GURLS::load<<GURLS::load<<GURLS::load;
*process2 <<GURLS:: computeNsave<<GURLS:: computeNsave<< GURLS:: computeNsave;
```

Here parameter selection for gaussian kernel requires selection of both the regularization parameter λ and the kernel parameter σ, and is performed selecting the task `siglamho` for the category `paramsel`. Once the value for kernel parameter σ has been chosen, the gaussian kernel is built through the `kernel` task with option `rbf`.

5 Customizing the options' structure

The options structure passed as third input to `GURLS.run` has a set of default fields and values. Some of these fields can be manually changed as in the following line of code

```
opt.addOpt("<FIELD>", <VALUE>);
```

where `opt` is an object of class `GurlOptionList`, and `<VALUE>` belongs the correct class. In the example of Subsection 4.2, we have seen how field `hoPerf` can be changed in order to deal with regression problems. Below we list the most important fields that can be customized

- `nlambda (OptNumber 20)`: number of values for the regularization parameter
- `nsigma (OptNumber 25)`: number of values for the kernel parameter.
- `nholdouts (OptNumber 1)`: number of data splits to be used for hold-out CV.
- `hoproportion (OptNumber 0.2)`: proportion between training and validation set in parameter selection
- `hoperf (OptString "macroavg")`: objective function to be used for parameter selection.
- `epochs (OptNumber 4)`: number of passes over the training set for stocastic gradient descent
- `subsize (OptNumber 50)`: training set size used for parameter selection when using stocastic gradient descent.
- `singlelambda (OptFunction "mean")`: function for obtaining one value for the regularization parameter, given the parameter choice for each class in multiclass classification (for each output in multiple output regression).

6 Available methods

In this section we summarize all the available tasks that have been implemented in GURLS^{++}.

Category	**Class**	**subclasses (task)**	
split (splits data into one or more pair of training and validation sets)	Split	Ho	
paramsel (performs selection of the regularization parameter λ and, if using Gaussian kernel, also of the kernel parameter σ)	ParamSelection	LoocvDual HoDual SiglamHo FixLambda HoPrimalr LooGPRegr SigLamLooGPRegr CalibrateSGD	LoocvPrimal HoPrimal Siglam FixSigLam HoDualr HoGPRegr SigLamHoGPRegr
kernel (builds the symmetric kernel matrix to be used for training)	Kernel	ChisquaredKernel RBFKernel	LinearKernel
optimizer	Optimizer	RLSPrimal RLSAuto RLSPrimalr RLSGPRegr	RLSDual RLSPegasos RLSDualr
predkernel	PredKernel	TrainTest	
pred	Prediction	PredPrimal PredGPRegr	PredDual
perf	Performance	MacroAvg Rmse	PrecisionRecall AbsErr
conf	Confidence	ConfMaxScore ConfBoltzmanGap	ConfGap ConfBoltzman

Table 1: List of GURLS $^{++}$ task classes and subclasses.