GURLS User's Guide

May 29, 2014

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` routines, 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 Fig.1.

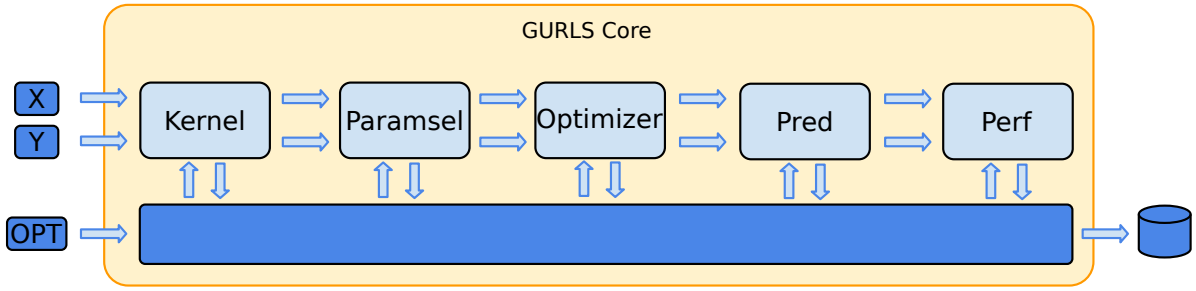

Figure 1: GURLS process.

2 The first example

The `gurls` command runs the learning pipeline and is the main function the user would directly call. It accepts exaclty four arguments:

1. the input data, stored in a $N \times D$ matrix, where N is the number of samples, D is the number of variables.

2. The data encoded labels stored in a $N \times T$ matrix, where T is the number of outputs. For (multi-class) classification, labels ($+1$,-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` needs to be called again.

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

- `opt.name`: defines a name for a given experiment.

- `opt.seq`: specifies the (ordered) sequence of tasks, i.e. the pipeline, to be executed.

- `opt.process`: specifies what to do with each task. It has to be a cell array, where each cell specify the executions code for each job, i.e. `gurls` call. In particular here are the codes:

  - 0 = Ignore
  - 1 = Compute
  - 2 = Compute and save
  - 3 = Load from file
  - 4 = Explicitly delete

Now, let's suppose we want to run the training process on a dataset `(Xtr,ytr)` and then test on a different dataset `(Xte,yte)`. We are interested in the average classification accuracy across classes. In order to train a linear classifier using a leave one out cross-validation approach, we just need the following lines of code:

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.seq = ...
    {'paramsel:loocvprimal','rls:primal','pred:primal','perf:macroavg'};
opt.process{1} = [2,2,0,0];
opt.process{2} = [3,3,2,2];
gurls (Xtr, ytr, opt,1)
gurls (Xte, yte, opt,2)
```

The meaning of the above code fragment 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 save it.

Note that the field `opt.name` is implicitly specified by the `defopt` function which assigns to it its only input argument. Fields `opt.seq` and `opt.process` have to be explicitly assigned.

3 Further examples

The `gurls` command 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. Type

```
help <CATEGORY>_<TASK>
```

(ex. `help paramsel_hoprimal`) for further reference on each task.

3.1 Linear classifier, primal case, hold-out cv

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.seq = {'split:ho','paramsel:hoprimal','rls:primal',...
  'pred:primal','perf:macroavg'};
opt.process{1} = [2,2,2,0,0];
opt.process{2} = [3,3,3,2,2];
gurls(Xtr, ytr, opt,1)
gurls(Xte, yte, opt,2)
```

Here hold-out cross validation requires the training test to be split in one pair of train and validation sets. Splitting is performed in the first task, `split`, with choice `ho`.

3.2 Linear regression, primal case, hold-out cv

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.seq = {'split:ho','paramsel:hoprimal','rls:primal',...
      'pred:primal','perf:rmse'};
opt.process{1} = [2,2,2,0,0];
opt.process{2} = [0,3,3,2,2];
opt.hoperf = @perf_rmse;
gurls(Xtr, ytr, opt,1)
gurls(Xte, yte, opt,2)
```

Here GURLS is used for regression. Note that the objective function is explicitly set to `@perf_rmse`, i.e. root mean square error, whereas in the first example `opt.hoperf` is set to its default `@perf_macroavg` which evaluates the average classification accuracy per class. The same code can be used for multiple output regression.

3.3 Linear classifier, dual case, leave one out cv

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.seq = {'kernel:linear', 'paramsel:loocvdual', 'rls:dual', ...
   'pred:dual', 'perf:macroavg''};
opt.process{1} = [2,2,2,0,0];
opt.process{2} = [3,3,3,2,2];
```

```
gurls(Xtr, ytr, opt,1)
gurls(Xte, yte, opt,2)
```

Here the dual formulation requires the kernel matrix, which is built through the task `linear` belonging to the category `kernel`. Note that the train-test kernel matrix is not build as, with linear kernel, prediction is implicitly performed in the primal formulation.

3.4 Gaussian Kernel, dual case, leave one out cv

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.seq = {'paramsel:siglam', 'kernel:rbf', 'rls:dual', ...
 'predkernel:traintest', 'pred:dual', 'perf:macroavg'};
opt.process{1} = [2,2,2,0,0,0];
opt.process{2} = [3,3,3,2,2,2];
gurls(Xtr, ytr, opt,1)
gurls(Xte, yte, opt,2)
```

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 `siglam` 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`.

3.5 Random features RLS, hold-out cv

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.seq = {'split:ho','paramsel:horandfeats',...
'rls:randfeats','pred:randfeats','perf:macroavg'};
opt.process{1} = [2,2,2,0,0];
opt.process{2} = [3,3,3,2,2];
normX = 1/normest(Xtr);
Xtr = Xtr.*s;
Xte = Xte.*s;
gurls(Xtr, ytr, opt,1)
gurls(Xte, yte, opt,2)
```

Computes a classifier for the primal formulation of RLS using the Random Features approach proposed by [1]. In this approach the primal formulation is used in a new space built through random projections of the input data. Note that the data has been rescaled to unitary norm.

3.6 Stocastic gradient descent

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.seq = {'paramsel:calibratesgd','rls:pegasos',...
      'pred:primal','perf:macroavg'};
opt.process{1} = [2,2,0,0];
opt.process{2} = [3,3,2,2];
gurls(Xtr, ytr, opt,1)
gurls(Xte, yte, opt,2)
```

Here the optimization is carried out using a stochastic gradient descent algorithm, namely Pegasos [2]. Note that the above pipeline uses the default value for option 'subsize' (50). If such a value is used with data sets with less than 50 samples the following error will be displayed:

```
GURLS usage error: the option subsize of the option list must be
smaller than the number of training samples!!
```

Set

```
opt.subsize = subsize;
```

with `subsize` smaller than the number of training samples to avoid errors.

4 Customizing the options' structure

The options structure passed as third input to `gurls` is built by function `defopt` with a set of default fields and values. Some of these fields can be manually customized by adding the line

```
opt.<FIELD> = <VALUE>;
```

before calling `gurls`, and after having built `opt` with `defopt`. In the example of Subsection 3.2, 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 (20): number of values for the regularization parameter
- nsigma (25): number of values for the kernel parameter.
- nholdouts (1): number of data splits to be used for hold-out cross validation.
- hoproportion (0.2): proportion between training and validation set in parameter selection
- hoperf (function `@perf_macroavg`): objective function to be used for parameter selection.
- epochs (4): number of passes over the training set for stocastic gradient descent
- subsize (50): training set size used for parameter selection when using stocastic gradient descent.
- singlelambda (function `@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).

As an example, in order to perform parameter selection on 5 different hold-out splits of the training set, with validation/training proportion set to 0.4, and with 20 and 10 values for the regularization and kernel parameter respectively, one has to run the following lines of code

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.seq = {'split:ho', 'paramsel:siglamho', 'kernel:rbf', ...
 'rls:dual','predkernel:traintest', 'pred:dual', 'perf:macroavg'};
opt.process{1} = [2,2,2,2,0,0,0];
opt.process{2} = [3,3,3,3,2,2,2];
```

```
opt.nlambda = 20;
opt.nsigma = 10;
opt.hoproportion = 0.4;
opt.nholdouts = 5;
gurls(Xtr, ytr, opt,1)
gurls(Xte, yte, opt,2)
```

5 Normalization functions

The `norm` set of functions allow to normalize the data. This is a preprocessing step, therefore it has not implemented as a GURLS task, and has to be called explicitly before running the pipeline. There are two possible ways to call these functions, that we describe in the following.

In the first example we separately normalize train and test data.

```
[Xtr] = norm_l2(Xtr,ytr,opt);
[Xte] = norm_l2(Xte,yte,opt);
```

In the following example the training set is first normalized and the column-wise means and covariances are saved to file. Then the test data are normalized according to the stats computed with the training set.

```
[Xtr] = norm_zscore(Xtr, ytr, opt);
[Xte] = norm_testzscore(Xte, yte, opt);
```

6 Results visualization

You can visualize the results of one or more GURLS pipelines using the `summary_*` functions. Below we show the usage of these set of functions for two sets of experiments (i.e. GURLS pipelines) each one run 5 times.

First we have to run the experiments. `nRuns` contains the number of runs for each experiment, and `filestr` contains the names of the experiments.

```
nRuns = {5,5};
filestr = {'hoprimal'; 'hodual'};

for i = 1:nRuns{1};
  opt = defopt(filestr{1} '_' num2str(i)];
  opt.seq = {'split:ho','paramsel:hoprimal','rls:primal',...
   'pred:primal','perf:macroavg','perf:precrec'};
  opt.process{1} = [2,2,2,0,0,0];
  opt.process{2} = [3,3,3,2,2,2];
  gurls(Xtr, ytr, opt,1)
  gurls(Xte, yte, opt,2)
end

for i = 1:nRuns{2};
  opt = defopt(filestr{2} '_' num2str(i)];
  opt.seq = {'split:ho', 'kernel:linear', 'paramsel:hodual', ...
  'rls:dual', 'pred:dual', 'perf:macroavg', 'perf:precrec'};
```

```
  opt.process{1} = [2,2,2,2,0,0,0];
  opt.process{2} = [3,3,3,3,2,2,2];
  gurls(Xtr, ytr, opt,1)
  gurls(Xte, yte, opt,2)
end
```

In order to visualize the results we have to specify in `fields` which fields of opt are to be displayed (as many plots as the elements of fields will be generated)

```
>> fields = {'perf.ap','perf.acc'};
```

we can generate "per-class" plots with the following command:

```
>> summary_plot(filestr,fields,nRuns)
```

and “global” plots with:

```
>> summary_overall_plot(filestr,fields,nRuns)
```

this generates “global” table:

```
>> summary_table(filestr, fields, nRuns)
```

This plots times taken by each step of the pipeline for performance reference:

```
>> plot_times(filestr,nRuns)
```

7 Available methods

In this section we summarize all the available tasks that have been implemented in the 2 modules, GURLS and BGURLS.

task category	description	available tasks
split	Splits data into one or more pair of training and validation sets	ho
paramsel	performs selection of the regularization parameter λ and, if using Gaussian kernel, also of the kernel parameter σ	fixlambda loocvprimal loocvdual hoprimal hodual siglam siglamho bfprimal bfdual calibratesgd hoprimalr hodualr horandfeats gpregrLambdaGrid gpregrSigLambGrid loogpregr hogpregr siglamhogpregr siglamloogpregr
kernel	builds the symmetric kernel matrix to be used for training	chisquared linear load randfeats rbf
rls	solves RLS optimization problem	primal dual auto pegasos primalr dualr randfeats gpregr
predkernel	builds the train-test kernel matrix	traintest
pred	predicts the labels	primal dual randfeats gpregr
perf	assess prediction performance	macroavg precrec rmse abserr
conf	computes a confidence for the highest scoring class	maxscore gap boltzmangap boltzman

Table 1: List of GURLS tasks organized by category.

Bibliography

[1] Ali Rahimi and Ben Recht. Random features for large-scale kernel machines. In *Neural Information Processing Systems (NIPS*, 2007.

[2] S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal estimated sub-gradient solver for svm. In *Proceedings of the 24th international conference on Machine learning*, pages 807–814. ACM, 2007.