bGURLS $^{++}$ User's Guide

July 1, 2013

1 Introduction

Modules bGURLS^{++}has been specifically designed to deal with the so-called *big learning* scenario, where big is meant either in terms of memory or in terms of computing time. In terms of memory, we consider to be big those data that cannot fully reside in RAM without any memory mapping techniques – such as swapping. Conversely data that can fully reside in RAM are considered to be small/medium. Learning with small/medium data can be carried out via GURLS^{++}. Learning with big data can be carried out with the dedicated modules bGURLS^{++}, under the conditions described in the following. Without the ambition to develop a good solution for all the infinte possible variants of big learning, we decided to focus specifically on those situations where one seeks a linear model on a large set of (possibly non linear) features. A more accurate specification of what "large" means in bGURLS^{++} is directly related to the number d of features: we require it must be possible to store a $d \times d$ matrix in memory. In practice, this roughly means we can train models with up-to $25k$ features on machines with $8Gb$ of RAM, and up-to $50k$ features on machines with $36Gb$ of RAM. It is important to remark we *do not* require the data matrix itself to be stored in memory. Indeed, in bGURLS^{++} it is possible to manage an arbitrarily large set of training examples.
In terms of computing time, big learning problems are those problems for which the exact solution requires a large amount of time, unless parallel computing is performed.

2 bGURLS^{++}**design**

bGURLS^{++} include all the design patterns described for GURLS^{++}, and has been complemented with additional big data and distributed computation capabilities. Big data support is obtained using a data structure called *bigarray*, which allows to handle data matrices as large as a machine's available space on hard drive instead of its RAM: we store the entire dataset on disk and load only small chunks in memory when required. bGURLS^{++} is designed to rely on the MPI protocol to distribute computations. Therefore, it allows for a full distribution within every single task of the pipeline. All the processes read the input data from a shared filesystem over the network and then start executing the same pipeline. During execution, each process' task communicates with the corresponding ones running over the other processes. Every process maintains his local copy of the options. Once the same task is completed by all processes, the local copies of the options are synchronized. This advanced architecture, which is shown in Fig. 1, allows for the creation of hybrid pipelines comprising serial one-process-based tasks from GURLS^{++}.

The usage of bGURLS $^{++}$ is very similar to that of GURLS $^{++}$, with the following exceptions:

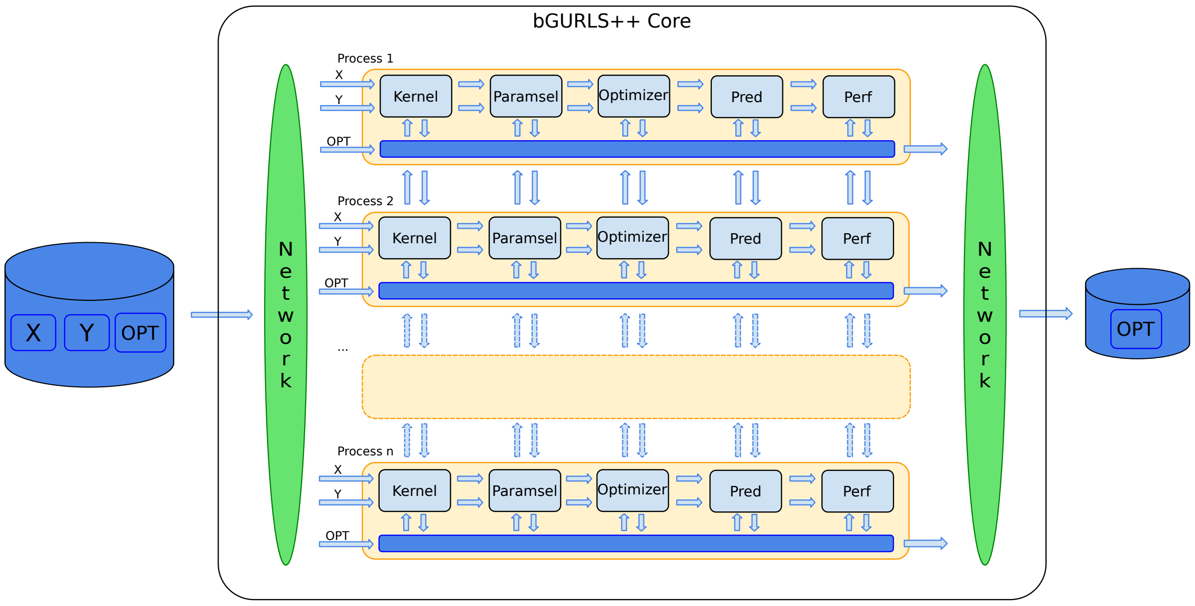

Figure 1: BGURLS^{++} design.

- the Gurls Core is implemented via the `BGURLS` class instead of the `GURLS`;
- The first two inputs of `BGURLS` must be bigarrays rather than matrices;
- The options structure must be of class `BGurlsOptionsList` rather than `GurlsOptionsList`;
- The only allowed "big" task categories for BGURLS $^{++}$ are `bigsplit`, `bigparamsel`, `bigoptimizer`, `bigpred` and `bigperf`;

3 BGURLS $^{++}$ example

Let us consider the demo bigmedmo.cpp in the demo subdirectory to better understand the usage of the BGURLS $^{++}$ module. The data set used in the demo is the `bio` data set used in [1], which is saved in the demo directory as a .zip file, 'bio_traintest_csv.zip', containing four files:

- 'Xtr.csv': containing the input $ntr \times d$ data matrix, where ntr is the number of training samples and d is the number of variables;
- 'Ytr.csv': containing the input $ntr \times 1$ label vector;
- 'Xte.csv': containing the input $nte \times d$ data matrix, where nte is the number of test samples and d is the number of variables;
- 'Yte.csv': containing the input $nte \times 1$ label vector;

Differently from GURLS $^{++}$, we chose the HDF5 data format to store matrices as it easily allows to read the content of the files by blocks. Let us now examine the salient and distinctive part of the demo.

The data is loaded as bigarray (actually only the information realtive to the data, not the data itself) with the following fragment of code:

```
BigArray<T> Xtr(path(shared_directory / "Xtr.h5").native(), 0, 0);
Xtr.readCSV(path(input_directory / "Xtr.csv").native());

BigArray<T> Xte(path(shared_directory / "Xte.h5").native(), 0, 0);
Xte.readCSV(path(input_directory / "Xte.csv").native());

BigArray<T> ytr(path(shared_directory / "ytr.h5").native(), 0, 0);
ytr.readCSV(path(input_directory / "ytr.csv").native());

BigArray<T> yte(path(shared_directory / "yte.h5").native(), 0, 0);
yte.readCSV(path(input_directory / "yte.csv").native());
```

The options' structure is built with default values via the following line of code:

```
BGurlsOptionList opt("bio_demoB", shared_directory.native(),true);
```

The pipeline is built as in GURLS $^{++}$, though with the BGURLS $^{++}$ task categories

```
OptTaskSequence *seq = new OptTaskSequence();
*seq<<"bigsplit:ho"<<"bigparamsel:hoprimal"<<"bigoptimizer:rlsprimal";
*seq << "bigpred:primal" << "bigperf:macroavg";
opt.addOpt("seq",seq);
```

The two sequences of actions identifying the training and test processes are defined exactly as in GURLS $^{++}$, whereas the processes are run through the BGURLS method as in the following:

```
BGURLS G;
G.run(Xtr,ytr,opt,jobid1);
G.run(Xte,yte,opt,jobid2);
```

4 Available methods

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

Category	Class	subclasses (task)
split	BigSplit	Ho
paramsel	BigParamSelection	HoPrimal
rls	BigOptimizer	RLSPrimal
pred	BigPrediction	PredPrimal
perf	BigPerformance	MacroAvg

Table 1: List of BGURLS++ task classes and subclasses.

Bibliography

[1] F. Lauer and Y. Guermeur. Msvmpack: A multi-class support vector machine package. *The Journal of Machine Learning Research*, 12:2293–2296, 2011.