CENTER FOR BIOLOGICAL & COMPUTATIONAL LEARNING (CBCL)

GURLS

A TOOLBOX FOR REGULARIZED LEAST SQUARES LEARNING

BASIC DOCUMENTATION

Authors	**Affiliation**
Andrea Tacchetti	CBCL
Pavan K. Mallapragada	CBCL
Matteo Santoro	CBCL, IIT@MIT Lab
Lorenzo Rosasco	CBCL, IIT@MIT Lab

LAST UPDATE: JULY 18, 2012

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Welcome

GURLS – which stands for *Grand Unified Regularized Least Squares* – is a software package for regression and (multiclass) classification based on the Regularized Least Squares (RLS) loss function. The package is available both in Matlab and C++. The following are among the key features of the package:

- automatic parameter selection;
- wide range of optimization routines;
- modularity, with the possibility of using each method independently;
- ability to handle massive datasets.

The present document describes both the API design and the usage of GURLS framework for machine learning researchers.

One of the main goals of GURLS is to build machine learning applications that scale well to reasonably large data sets. The core algorithms of the package are designed to run on a single desktop machine. However, a significant contribution is the extension of some functionalities to run on a cluster of CPUs. We believe the current version of the library is reasonably optimized to allow for good performance for both non-distributed and distributed applications.

From a user's viewpoint, we tried to design a very intuitive library that can be used without having a lot of expertise in Matlab or C++ programming. From a developer's viewpoint, our attempt was to minimize the effort required to write single machine learning modules and to create an easy environment where it is possible to write and prototype new code and algorithms by exploiting what has been already implemented and made available in the package.

1.2 Getting started

1.2.1 Downloading the GURLS package

The Home Page of the GURLS package is at `http://cbcl.mit.edu/gurls/`, where you will find useful information about GURLS project, the authors and a link to the research group where the toolbox has been designed and developed, and the link to the GITHUB repository (`http://github.com/CBCL/GURLS`) from which the source code can be downloaded.

1.2.2 Installing GURLS

GURLS is a pure Matlab library and has no specific dependencies on external libraries, made exception for the stats toolbox (see Subsection 3.3.1). Once the compressed archive has been downloaded on your machine from the GITHUB repository, you need to save it in the desired `PACKAGE_ROOT`. Then open MATLAB and execute:

```
>> run('PACKAGE_ROOT/gurls/utils/gurls_install.m');
```

This will add all the important directories to your path. Run `savepath` if you want the installation to be permanent.

1.2.3 Installing GURLS++

In the following we assume the the source code of `GURLS++` is located at `$GURLS_ROOT`.

Installing GURLS++ on Linux

Below we describe how to build and install GURLS++ on Ubuntu (tested on Ubuntu 10.04). For other distributions, the same packages must be installed with the distribution-specific method.

1. Install the cmake build system (`www.cmake.org/`)

   ```
   $ sudo apt-get install cmake
   ```

2. Install your favourite Blas/Lapack implementation. Currently we support:

   - ATLAS: `http://math-atlas.sourceforge.net`;
   - AMD's ACML: `http://developer.amd.com/libraries/acml`;
   - Intel's MKL: `http://software.intel.com/en-us/articles/intel-mkl`;
   - Netlib's reference implementation: `http://www.netlib.org/blas` and `http://www.netlib.org/lapack/`.

   As an example, to install Netlib's Blas and Lapack run the following command:

   ```
   $ sudo apt-get install libblas-dev libblas3gf liblapack3gf
   ```

3. Install Boost's libraries date_time and serialization

   ```
   $ sudo apt-get install libboost-serialization1.40.0
   libboost-serialization1.40-dev
   libboost-date-time1.40.0 libboost-date-time1.40-dev
   ```

4. To link against some Blas and Lapack implementations you may need a fortran compiler e.g. for Netlib's Blas and Lapack:

   ```
   $ sudo apt-get install gfortran
   ```

5. Create a build directory (e.g. "build") for `GURLS++`

   ```
   $ cd $GURLS_ROOT
   $ mkdir build
   ```

6. Run cmake into the build directory

   ```
   $ cd build
   $ ccmake ..
   ```

   The last command will show the CMake interface, which must be used to set the values of some variables used for building and installing `GURLS++`. See the section **Configuring GURLS++** below for more information on these variables and how to set them to appropriate values.

7. Start `GURLS++` building

   ```
   $ make
   ```

8. Install `GURLS++` to the path defined at configuration time

   ```
   $ make install
   ```

Installing GURLS++ on Windows

Below we describe how to build and install GURLS++ on Windows with Visual Studio (tested with VS Express 2010 on Windows 7 and with VS Express 2008 on Windows Vista).

1. Install the CMake build system downloading the installer from `http://cmake.org/cmake/resources/software.html`.

2. Install your favourite Blas/Lapack implementation. Currently we support:

   - ATLAS: `http://math-atlas.sourceforge.net`;
   - AMD's ACML: `http://developer.amd.com/libraries/acml`;

- Intel's MKL: `http://software.intel.com/en-us/articles/intel-mkl`.

Under Windows ACML is probably the easiest choice, since they provide the library binaries for free.

3. Install Boost's `date_time` and `serialization` libraries following the instructions on Boost's website: `http://www.boost.org/doc/libs/1_50_0/more/getting_started/windows.html`.

4. Create a build directory (e.g. `$GURLS_ROOT/build`) for `GURLS++`.

5. Run the CMake GUI.
   You will have to set the source directory to the root directory holding GURLS++ source code, and the build directory to the directory created at the previous step.
   After pressing the configure button, you have to set the values of some variables used for building and installing `GURLS++`. See the section **Configuring GURLS++** below for more information on these variables and how to set them to appropriate values. After having configured the build options, press the generate button to create the solution file.

6. Build `GURLS++` by opening and building the solution under Visual Studio.

7. Install `GURLS++` by explicitely building the install project included in the solution (it is not automatically built when building the solution).

Configuring GURLS++

The configuration step is carried out using CMake.

Using the command-line interface you will be presented with a textual menu, where you can change the values of some variables and configure the building settings accordingly.

1. At the beginning no variable is set, and a message `EMPTY CACHE` is shown.

2. Press `[c]` for 'configure', and CMake will try to determine the correct values for all variables, such as the location of required libraries.

3. If CMake does not find some required library, an error message will be displayed. In this case press `[e]` to exit help and go to the main screen.

4. In the main screen you may change a number of variables. Most of them can be left unchanged, but some must be set to appropriate values. The following are the variables whose values should be checked:

   - `BLAS_LAPACK_IMPLEMENTATION`. Allows user to specify an implementation of the Blas/Lapack routines. Available choices are: `ACML`, `ATLAS`, `MKL`, `NETLIB` (under linux). Depending on the choice you make, CMake will try to find the libraries in standard locations in the system. Normally this process should run fine, however, in case the libraries have been installed in some non-standard directory, you may have to manually specify their location.

- `USE_BINARY_ARCHIVES`. If set to `ON`, all data structures in GURLS are stored in binary (rather than text) files, saving storage space and time.
- `BUILD_DEMO` and `BUILD_TEST`. These two options enable the building of the demo and test programs.
- `CMAKE_INSTALL_PREFIX`. This is the path where the library will be installed to.

5. Once all variables have been set, press `[c]` again, and CMake will check the settings. As in step (3), if something is wrong an error message will be displayed and you will have to go back to the main screen to tweak the configuration.
6. When the settings are correct, the option to 'generate' the files required for the actual build will appear. Press `[g]`. CMake will generate the files and exit.

After the build files (e.g. the Makefile under Linux) have been generated, you can proceed as explained above.

The same procedure outlined above is used when using the GUI of CMake, e.g. under Windows or Mac.

1.2.4 Hello World in GURLS

Have a look, and run gurls_helloworld.m in the 'demo' subdirectory. Below we describe the demo in details. We first have to load the training data

```
>> load('data/quickanddirty_traindata;')
```

and train the classifier

```
>> [opt] = gurls_train(Xtr,ytr);
```

now we load the test data

```
>> load('data/quickanddirty_testdata');
```

then we predict the labels for the test set and asses prediction accuracy

```
>> [yhat,acc] = gurls_test(Xte,yte,opt);
```

1.2.5 Hello World in GURLS++

Have a look, and run helloworld.cpp in the 'demo' subdirectory. Below we describe the salient parts of demo in details.
First we have to load the training data

```
    Xtr = readFile<T>("../data/Xtr.txt");
    ytr = readFile<T>("../data/ytr_onecolumn.txt");
```

and the test data

```
Xte = readFile<T>("../data/Xte.txt");
yte = readFile<T>("../data/yte_onecolumn.txt");
```

then we train the classifer

```
GurlsOptionsList* opt = gurls_train(*Xtr, *ytr);
```

finally we predict the labels for the test set and asses prediction accuracy

```
gurls_test(*Xte, *yte, *opt);
```

1.2.6 License

GURLS is distributed under the BSD license. This means that it is free for both academic and commercial use.

If you are going to use GURLS in your scientific work, please cite the toolbox, the main website and the paper

> Tacchetti, A., P. Mallapragada, M. Santoro, and L. Rosasco
> *GURLS: a Toolbox for Large Scale Multiclass Learning,*
> presented at Workshop: "Big Learning: Algorithms, Systems, and Tools for Learning at Scale" at NIPS 2011, December 16-17 2011, Sierra Nevada, Spain.

CHAPTER 2

USER'S GUIDE

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. In the remainder of the chapter, a number of commonly used experiments will be described.

As both GURLS and GURLS++ are similarly designed, in the following we start describing GURLS usage, and later explain what changes in the `C++` implementation. Then we described some additional functionalities of the package, precisely the normalization and visualization commands.

2.1 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 (+1,−1) stored (for One-Vs-All) in a $N \times T$ matrix, where T is the number of classes.
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 to run the training process on a dataset `(Xtr,ytr)` and then test on a different dataset `(Xte,yte)`. We are interested in the precision-recall performance measure as well as the average classification accuracy. 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:precrec','perf:macroavg'};
opt.process{1} = [2,2,0,0,0];
opt.process{2} = [3,3,2,2,2];
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 two aforementioned performance measures and save them.

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.

2.2 Further examples

The gurls command executes an ordered sequence of tasks, specified in the option seq of the options' structure as

```
{'<TASK1>:<TASK1_CHOICE>';'<TASK2>:<TASK2_CHOICE>;...}
```

These tasks can be combined in order to build different train-test pipelines. A list of GURLS tasks, along with the list of the available choices already implemented, is summarized in Table 2.1. Type

```
help <TASK>_<TASK_CHOICE>
```

(ex. help paramsel_hoprimal) for further reference on each task choice. The most popular learning pipelines are outlined in the following.

2.2.1 Linear classifier, primal case, hold-out cv

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.seq = {'split:ho','paramsel:hoprimal','rls:primal',...
  'pred:primal','perf:macroavg','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)
```

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.

2.2.2 Linear classifier, primal case, hold-out cv, randomized SVD

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.seq = {'split:ho','paramsel:hoprimalr','rls:primalr',...
  '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)
```

Here a randomized version of the singular value decomposition (SVD).

2.2.3 Linear classifier, dual case, leave one out cv

```
name = 'ExampleExperiment';
opt = defopt(name);
```

task type	**description**	**available choices**
split	Splits data into one or more pair of	ho
paramsel	training and validation sets 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
kernel	builds the symmetric kernel matrix to be used for training	chisquared linear load rbf
rls	solves RLS optimization problem	primal dual auto pegasos primalr dualr
predkernel	builds the train-test kernel matrix	traintest
pred	predicts the labels	primal dual
perf	assess prediction performance	macroavg precrec rmse
conf	computes a confidence for the highest scoring class	maxscore gap boltzmangap boltzman

Table 2.1: List of GURLS tasks along with description and list of available choices for each task.

```
opt.seq = {'kernel:linear', 'paramsel:loocvdual', 'rls:dual', ...
   'pred:dual', '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)
```

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

2.2.4 Linear classifier, dual case, hold-out cv

```
name = 'ExampleExperiment';
opt = defopt(name);
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)
```

Here parameter selection is performed via hold-out cross validation.

2.2.5 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 choice `siglam` for the task `paramsel`. Once the value for kernel parameter σ has been chosen, the gaussian kernel is built through the `kernel` task with option `rbf`.

2.2.6 Gaussian Kernel, dual case, hold-out cv

```
name = 'ExampleExperiment';
opt = defopt(name);
```

```
opt.seq = {’split:ho’, ’paramsel:siglamho’, ’kernel:rbf’, ...
 ’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];
gurls(Xtr, ytr, opt,1)
gurls(Xte, yte, opt,2)
```

Here multiple parameter selection for gaussian kernel is performed with hold-out cross validation with the option `siglamho` for the task `paramsel`.

2.2.7 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.

2.2.8 Multiple output regression

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

Here GURLS is used for a multiple output regression problem. 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.

2.3 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 2.2.8, 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 (100): 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)
```

2.4 Examples in GURLS++

In C++ the counterpart of the gurls function is the GURLS class, with its only method run, whereas function defopt has its equivalent in the class GurlsOptionsList. In the 'demo' directory you will find GURLSloocvprimal.cpp, which implements exactly the first example described in Section 2.1. In the following we report it for completeness.

```
#include <iostream>
#include "gurls.h"
#include "exceptions.h"
#include "gmat2d.h"
#include "options.h"
#include "optlist.h"

using namespace gurls;
using namespace std;

typedef double T;

int main(int argc, char *argv[])
{
    string xtr_file, xte_file, ytr_file, yte_file;

// check that all inputs are given
    if(argc<4)
    {
        printHelp(argc, argv);
        return 0;
    }

// get file names from input
    xtr_file = argv[1];
    xte_file = argv[2];
    ytr_file = argv[3];
    yte_file = argv[4];

    try
    {
        gMat2D<T> *Xtr, *Xte, *ytr, *yte;

// load data from file
        Xtr = readFile<T>(xtr_file);
        Xte = readFile<T>(xte_file);
        ytr = readFile<T>(ytr_file);
        yte = readFile<T>(yte_file);

// specify the task sequence
```

```
        OptTaskSequence *seq = new OptTaskSequence();

        seq->addTask("paramsel:loocvprimal");
        seq->addTask("optimizer:rlsprimal");
        seq->addTask("pred:primal");
        seq->addTask("perf:macroavg");
        seq->addTask("perf:precrec");

        GurlsOptionsList *process = new GurlsOptionsList("processes",false);
// defines instruction sequence for training process
        std::vector<double> process1;
        process1.push_back(GURLS::computeNsave);
        process1.push_back(GURLS::computeNsave);
        process1.push_back(GURLS::ignore);
        process1.push_back(GURLS::ignore);
        process1.push_back(GURLS::ignore);
        process->addOpt("one", new OptNumberList(process1));

// defines instruction sequence for testing process
        std::vector<double> process2;
        process2.push_back(GURLS::load);
        process2.push_back(GURLS::load);
        process2.push_back(GURLS::computeNsave);
        process2.push_back(GURLS::computeNsave);
        process2.push_back(GURLS::computeNsave);
        process->addOpt("two", new OptNumberList(process2));

// build an options' structure
        GurlsOptionsList* opt = new GurlsOptionsList("GURLSlooprimal",false);
        opt->addOpt("seq", &seq);
        opt->addOpt("processes", process);

        GURLS G;
// run gurls for training
        G.run(Xtr, ytr, *opt, jobIds[0]);
// run gurls for testing
        G.run(Xte, yte, *opt, jobIds[1]);

    }
    catch (gException& e)
    {
        cout << e.getMessage() << endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
```

```
}
```

In order to run the other examples you just have to substitute the code fragment for the task pipeline

```
seq->addTask("paramsel:loocvprimal");
seq->addTask("optimizer:rlsprimal");
seq->addTask("pred:primal");
seq->addTask("perf:macroavg");
seq->addTask("perf:precrec");
```

and for the sequence of instructions

```
GurlsOptionsList* process = new GurlsOptionsList("processes",false);
// defines instructions for training process
std::vector<double> process1;
process1.push_back(GURLS::computeNsave);
process1.push_back(GURLS::computeNsave);
process1.push_back(GURLS::ignore);
process1.push_back(GURLS::ignore);
process1.push_back(GURLS::ignore);
process->addOpt("one", new OptNumberList(process1));

// defines instructions for testing process
std::vector<double> process2;
process2.push_back(GURLS::load);
process2.push_back(GURLS::load);
process2.push_back(GURLS::computeNsave);
process2.push_back(GURLS::computeNsave);
process2.push_back(GURLS::computeNsave);
process->addOpt("two", new OptNumberList(process2));
```

with the desired task pipeline and instructions sequence. For example, for the case 'Gaussian Kernel, dual case, hold-out cv' the code for defining the task pipeline must be

```
seq->addTask("split:ho");
seq->addTask("paramsel:siglamho");
seq->addTask("kernel:rbf");
seq->addTask("optimizer:rlsdual");
seq->addTask("pred:dual");
seq->addTask("predkernel:traintest");
seq->addTask("perf:macroavg");
seq->addTask("perf:precrec");
```

and the code fragment specifying the sequence of instructions must be

```
GurlsOptionsList* process = new GurlsOptionsList("processes",false);
// defines instructions for training process
std::vector<double> process1;
process1.push_back(GURLS::computeNsave);
process1.push_back(GURLS::computeNsave);
process2.push_back(GURLS::computeNsave);
process2.push_back(GURLS::computeNsave);
process1.push_back(GURLS::ignore);
process1.push_back(GURLS::ignore);
process1.push_back(GURLS::ignore);
process1.push_back(GURLS::ignore);
process->addOpt("one", new OptNumberList(process1));

// defines instructions for testing process
std::vector<double> process2;
process2.push_back(GURLS::load);
process2.push_back(GURLS::load);
process2.push_back(GURLS::load);
process2.push_back(GURLS::load);
process2.push_back(GURLS::computeNsave);
process2.push_back(GURLS::computeNsave);
process2.push_back(GURLS::computeNsave);
process2.push_back(GURLS::computeNsave);
process->addOpt("two", new OptNumberList(process2));
```

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)
```

In GURLS++ normalization is implemented through the classes Norm, NormZScore and NormTestZScore. We refer to the doxygen documentation of each class for further reference.

2.6 Results visualization (only for GURLS)

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)
```

CHAPTER 3

MATLAB DEVELOPER'S GUIDE

3.1 Package Organization

The GURLS toolbox contains two user functions `gurls` and `defopt`, and a number of supporting functions and additional functionalities which the developer needs to know about. These functions can be divided as:

task functions tasks are the elements of the learning pipeline, for each task several choices are available (see Table 2.1).

utility functions functions called by the task functions.

quickanddirty functions pair of simplified calls of a train-test GURLS pipeline.

normalization functions set of functions for normalizing the data.

summary functions set of functions that help in visualizing the results.

3.2 The `gurls` function

The `gurls` function

```
function [opt] = gurls (X, y, opt, jobid)
```

is, beyond `defopt`, the only function the user would directly call. It executes an ordered sequence of tasks. For each task several choices are available (see Table 2.1). We recall that the choice for each task has to be specified in the field `seq` of the input options' structure as

```
{<TASK1>:<TASK1_CHOICE>'; <TASK2>:<TASK2_CHOICE>';...}
```

Within the `gurls` function, each task is executed by calling the function corresponding to the task choice specified in the options' structure as `{'<TASK1>:<TASK1_CHOICE>';...}`. After the `gurls` function has executed the task, the returned variable is stored in the field of the options' structure named after the task. After all tasks have been executed, it removes the fields of options'structure that must not be saved, as specified in the option `process` of the input options'structure.

3.3 GURLS tasks

The `gurls` command executes an ordered sequence of tasks, specified in the option `seq` of the input options' structure. The available tasks and task choices are listed in Table 2.1. Each task choice is implemented as a function `<TASK>_<TASK_CHOICE>.m`, e.g. `paramsel_hoprimal.m`. The functions that implement different choices of the same task are all saved in the same directory, named after the task (e.g. directory `perf` contains the files `perf_macroavg`, `perf_precrec`, and `perf_rmse`) and containing also the tasks' utility functions.

All task functions have the same input-outpt structure as they take three inputs:

X : input data matrix

y : labels matrix

OPT : options' structire

and return only one output.

3.3.1 Description of the available tasks choices

split Splits data into traing and validation sets

splits data into one pair of training and validation sets

ho Splits data into one (default) or more pairs of training and validation sets

paramsel Performs parameter selection

fixlambda Sets the regularization parameter to the value set in OPT

loocvprimal Performs parameter selection when the primal formulation of RLS is used. The leave-one-out approach is used.

loocvdual Performs parameter selection when the dual formulation of RLS is used. The leave-one-out approach is used.

hoprimal Performs parameter selection when the primal formulation of RLS is used. The hold-out approach is used.

hodual Performs parameter selection when the dual formulation of RLS is used. The hold-out approach is used.

- **siglam** Performs parameter selection when the dual formulation of RLS is used. The leave-one-out approach is used. It selects both the regularization parameter lambda and the kernel parameter sigma (requires the stats toolbox).
- **siglamho** Performs parameter selection when the dual formulation of RLS is used (requires the stats toolbox). The hold-out approach is used. It selects both the regularization parameter lambda and the kernel parameter sigma.
- **bfprimal** Performs parameter selection when the primal formulation of RLS is used. The hold-out approach is used in a brute force way, i.e. the RLS problem is solved from scratch for each value of the regularizer.
- **bfdual** Performs parameter selection when the dual formulation of RLS is used. The hold-out approach is used in a brute force way, i.e. the RLS problem is solved from scratch for each value of the regularizer.
- **calibratesgd** Performs parameter selection when one wants to solve the problem using `pegasos` as `rls` task.

kernel Computes the symmetric Kernel matrix

- **linear** Computes the Kernel matrix for a linear model.
- **rbf** Computes the kernel matrix for a Gaussian kernel.
- **chisquared** Computes the Kernel matrix for chi-squared kernel.
- **load** Loads the kernel matrix from disk.

rls Solves RLS optimization problem

- **primal** Computes a classifier for the primal formulation of RLS.
- **dual** Computes a classifier for the dual formulation of RLS.
- **auto** Computes a RLS classifier, with automatic selection of primal/dual procedure.
- **pegasos** Computes a classifier for the primal formulation of RLS. The optimization is carried out using a stochastic gradient descent algorithm.

predkernel Computes the Kernel matrix for prediction

- **traintest** Computes the kernel matrix between the training points and the test points.

pred Predicts the labels

- **primal** Computes the predictions of the linear classifier stored computed with the primal formulation of RLS, on the samples passed in the X matrix.
- **dual** Computes the predictions of the classifier computed with the dual formulation of RLS, on the samples passed in the X matrix.

perf Assesses prediction performance

macroavg Computes the average classification accuracy per class.

precrec Computes the average precision per class

rmse Computes the root mean squared error for the predictions, taken as the Frobenius norm of the distance matrix between the predicted labels specified in the field pred of opt and the true labels y

conf Computes a confidence for the highest scoring class

maxscore Computes a confidence estimation for the predicted class (i.e. highest scoring class). The highest score is considered

gap Computes a confidence estimation for the predicted class (i.e. highest scoring class). The difference between the highest scoring class and the second highest scoring class is considered.

boltzman Computes the probability of belonging to the highest scoring class. The scores are converted in probabilities using the Boltzman distribution.

boltzmangap Computes a confidence estimation for the predicted class (i.e. highest scoring class). The scores are converted in probabilities using the Boltzman distribution and the difference between the highest scoring class and the second highest scoring class is used as an estimate.

3.4 `defopt` function

The `defopt` function:

```
function opt = defopt(expname)
```

builds a defualt options' structure. The options' structure, given in input to `gurls`, and passed to the task functions, contains all the information relative to the learning pipeline and to the tasks of the pipeline that have already been executed. The `defopt` function assignes default values to the following fields (in parenthesis the default values):

Experiment options:

- name (`expname`): experiment identifier.
- savefile (`expname.mat`) name of the file where results will be saved.

Data options:

- 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.
- nlambda (100): number of values for the regularization parameter.

- nsigma (25): number of values for the kernel parameter.

General algorithm options:

- kernel (structure with field type set to `'rbf'`): kernel for dual formulation.
- singlelambda (function `@mean`): function for obtaining one value for the regularization parameter, given the selected parameter value for each class in multiclass classification (for each output in multiple output regression).
- smallnumber (1e-8): sets the lower limit for the values of the regularization parameter.
- hoperf (function `@perf_macroavg`): objective function to be used for parameter selection.

Pegasos options:

- 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.
- calibfile ('foo'): name of the file used by `paramsel_calibratesgd` to save temporary results in parameter selection for pegasos.

Version info:

- version (0.1): GURLS version.

The options' structure saved by `gurls` contains both the above default values and some additional fields: the field `times`, where computing times for each task are saved, and the fields corresponding to the tasks which corresponding instruction code has been set to 2 (=`CSV`), in at least one job.

3.5 Other supporting functions

In addition to the two user functions `gurls` and `defopt` and the tasks functions, the GURLS toolbox comprises a number of supporting functions.

3.5.1 Installation functions

gurls_install Adds all the important directories to your path.

gurls_root Returns the directory where the calling M-files is saved

3.5.2 Tasks utility functions

GInverseDiagonal Utility function for the `loocvdual` choice of `paramsel` task.

rls_eigen Returns the RLS optimizer given the data singular value decomposition. It is called by the `paramsel` and `rls` tasks.

precrec_driver Utility function for the `precrec` choice of the `perf` task.

distance Computes the Euclidean squared distance matrix. It is used to compute the kernel in the `paramesel`, `kernel` and `predkernel` tasks.

paramsel_lambdaguesses Returns a geometric series of values for the regularization parameter. It is a utility function for the `paramsel` task.

tygert_svd Computes randomized singular value decomposition. It is called by the `primalr` and `dualr` options for both `paramesel` and `rls` tasks.

rls_pegasos_driver Utility function for the `pegasos` choice of `rls` task. Computes a single pass for pegasos algorithm, performing the stochastic gradient descent over all training samples once.

rls_primal_driver Utility function for the `primal` choice of `rls` task.

3.5.3 `enums` functions

ign Returns the instruction code for `'IGNORE'` in a GURLS process sequence

cpt Returns the instruction code for `'COMPUTE'` in a GURLS process sequence

csv Returns the instruction code for `'COMPUTE AND SAVE'` in a GURLS process sequence

ldf Returns the instruction code for `'LOAD FROM FILE'` in a GURLS process sequence

del Returns the instruction code for `'EXPLICITLY DELETE'` in a GURLS process sequence

3.6 Designing and Implementing a new functionality

Thanks to its great modularity you can extend the GURLS package by adding new choices for the already available tasks. You can add a new task choice, say `<NEWTASKCHOICE>`, for task `<TASK>`, by implementing the function `<TASK>_<NEWTASKCHOICE>.m` (to be saved in the `<TASK>` folder) with the following structure

```
function [out] = <TASK>_<NEWTASKCHOICE>(X, y, opt)
\%<TASK>_<NEWTASKCHOICE>(X,Y,OPT)
\%Computes ....
\%INPUTS:
\%-X: input data matrix
```

```
\%-y: labels matrix
\%-OPT: structure of options with the following fields (and subfields):
\%    fields that need to be set through previous gurls tasks:
\%     - ... (set by the ...* routine)
\%    fields with default values set through the defopt function:
\%     - ...
\%
\%   For more information on standard OPT fields
\%   see also defopt
\%
\% OUTPUT: ...

\% new code
\% ...
\% ...
end
```

CHAPTER 4

C++ DEVELOPER'S GUIDE

4.1 Package Organization

The GURLS++ toolbox comprises two main classes, GURLS and GurlsOptionsList, and a number of supporting classes and additional functionalities which the developer needs to know about. These functions can be divided as:

task classes tasks are the elements of the learning pipeline, for each task several subclasses are available (see Table 2.1).

utilities functions called by the task classes .

quickanddirty classes pair of simplified calls of a train-test GURLS pipeline.

normalization classes set of classes for normalizing the data.

In the following we describe the user's classes GURLS and GurlsOptionList and the set of tasks classes. For all other classes we refer to the doxygen documentation.

4.2 The **GURLS** class

The GURLS class has only one method, run, which executes an ordered sequence of tasks. For each task several choices are available (see Table 2.1). We recall that the choice for each task has to be specified in the field seq of the input options' structure, say opt, as

```
OptTaskSequence *seq = new OptTaskSequence();
seq->addTask("<TASK1_TYPE>:<TASK1_CHOICE>");
...
seq->addTask("<TASKn_TYPE>:<TASKn_CHOICE>");
opt->addOpt("seq", &seq);
```

The `run` method executes each task of the sequence. After all tasks have been executed, it removes the fields of the options'structure corresponding to tasks that must not be saved, as specified in the option `process` of the input options'structure.

4.3 GURLS tasks

The `run` method of the `GURLS` class executes an ordered sequence of tasks. For the list of possible task choices we refer to the Matlab developer's guide (precisely Section 3.3) Each task is implemented as a class, with its sub-classes being listed in Table 4.1 and with the only difference that classes and subclasses in C++ are defined in Camel Case.

All the task classes have the same input-outpt structure as they take three inputs:

X : input data matrix

y : labels matrix

OPT : options' structire

and return no output. All task classes have only one method, namely `execute`, that updates or creates a field of the options'structure. This is the main difference with respect to GURLS in Matlab, where the task functions return a variable which is then placed in the options' structure by the `gurls` function. In a later release of the GURLS++, this will be changed in order to conform GURLS++ to GURLS.

task	Class	subclasses	
split	Split	Ho	
paramsel	ParamSelection	LoocvDual HoDual SiglamHo FixLambda BfDual HoPrimalr	LoocvPrimal HoPrimal Siglam CalibrateSGD BfPrimal HoDualr
kernel	Kernel	ChisquaredKernel RBFKernel	LinearKernel
rls	Optimizer	RLSPrimal RLSAuto RLSPrimalr	RLSDual RLSPegasos RLSDualr
predkernel	PredKernel	TrainTest	
pred	Prediction	PredPrimal	PredDual
perf	Performance	MacroAvg Rmse	PrecisionRecall
conf	Confidence	ConfMaxScore ConfBoltzmanGap	ConfGap ConfBoltzman

Table 4.1: List of GURLS++ task classes and subclasses.

4.4 The `GurlsOptionList` class

The options' structure given in input to the `GURLS` class, and passed to the task functions, contains all the information relative to the learning pipeline and to the tasks of the pipeline that have already been executed. All options must be of one of the following `OptTypes`:

- OptNumber
- OptNumberList
- OptList
- OptString
- OptStringList
- OptMatrix
- OptFunction
- OptTaskSequence

The `GurlsOptionList` class builds a default options' structure assigning default values to the following fields (in parenthesis the default values):

Experiment options:

- name (OptString(ExpName) where ExpName is string given in input to GurlsOptionList): experiment identifier.

- savefile (OptString(ExpName.txt): name of the file where results will be saved.

Data options:

- nholdouts (OptNumber(1)): number of data splits to be used for hold-out cross validation.

- hoproportion (OptNumber(0.2)): proportion between training and validation set in parameter selection.

- nlambda (OptNumber(100)): number of values for the regularization parameter.

- nsigma (OptNumber(25)): number of values for the kernel parameter.

General algorithm options:

- singlelambda (OptFunction("median")): function for obtaining one value for the regularization parameter, given the selected parameter value for each class in multiclass classification (for each output in multiple output regression).

- smallnumber (OptNumber(1e-8)): sets the lower limit for the values of the regularization parameter.

- hoperf (macroavg): objective function to be used for parameter selection.

Pegasos options:

- 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.

- calibfile (OptString("foo")): name of the file used by paramsel_calibratesgd to save temporary results in parameter selection for pegasos.

Version info:

- version (OptString("0.1")): GURLS version.

The options' structure saved by the run method of the GURLS class contains both the above default values and some additional fields: the field times, where computing times for each task are saved, and the fields corresponding to the tasks which corresponding instruction code has been set to 2 (=CSV), in at least one job.

4.5 Designing and Implementing a new functionality

Thanks to its great modularity you can extend the GURLS++ package by adding new choices for the already available tasks.You can add a new task choice, say <NEWTASKCHOICE>, for task <TASK>, by adding the following two lines of code

```
        }else if(id == "<NEWTASKCHOICE>"){
            return new <NEWSUBCLASS><T>;
```

right before the following fragment of code:

```
        } else
            throw BadPredictionCreation(id);
```

in the definition of the the class, say <CLASS>, corresponding to <TASK>, and by creating the subclass <NEWSUBCLASS> of <CLASS>. The new subclass must be defined in the following way:

```
#ifndef _GURLS_<NEWSUBCLASSFILE>_H_
#define _GURLS_<NEWSUBCLASSFILE>_H_

#include <cstdio>
#include <cstring>
#include <iostream>
#include <cmath>
#include <algorithm>
#include <set>
#include "options.h"
#include "optlist.h"
#include "gmat2d.h"
#include "gvec.h"
#include "gmath.h"

// HERE INCLUDE THE HEADER WHERE THE TASK CLASS IS DEFINED
#include // other required classes

namespace gurls {

/**
   * \brief <NEWSUBCLASS> is the subclass of <CLASS> that implements ...
   */

template <typename T>
class <NEWSUBCLASS>: public <CLASS><T>{

public:
    /**
     * ...
     * \param X input data matrix
     * \param Y labels matrix
     * \param opt options with the following:
     *  - ...
     *
     * \return adds the following fields to opt:
     *  - ...
     */
    void execute(const gMat2D<T>& X, const gMat2D<T>& Y,
        GurlsOptionsList& opt);
};

template <typename T>
void <NEWSUBCLASS><T>::execute(const gMat2D<T>& X_OMR,
    const gMat2D<T>& Y_OMR, GurlsOptionsList& opt)
{
//transpose X and y
   gMat2D<T> X(X_OMR.cols(), X_OMR.rows());
   X_OMR.transpose(X);
   gMat2D<T> Y(Y_OMR.cols(), Y_OMR.rows());
   Y_OMR.transpose(Y);

// define new GurlsOptions or GurlsOptionsList depending on
// the type of outout

/*
 new code here
 ...
 ...
 */
    // add the new GurlsOptions/GurlsOptionsList to opt
        opt.addOpt("optimizer",optimizer);
}
}
#endif // _GURLS_<NEWSUBCLASSFILE>_H_
```

Note that matrices `X` and `y` must be transposed. Similarly all matrices/vectors that compose the new `GurlsOptions/GurlsOptionsList` must be first transposed. This is a design choice based on the column major order required by BLAS-LAPACK routines.