MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CENTER FOR BIOLOGICAL & COMPUTATIONAL LEARNING (CBCL)

GURLS
A LEAST SQUARES-BASED LIBRARY FOR STATE OF THE ART SUPERVISED 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: SEPTEMBER 25, 2014

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Welcome

GURLS – which stands for *Grand Unified Regularized Least Squares* – is a software library for regression and (multiclass) classification based on the Regularized Least Squares (RLS) loss function. The library comprises four main modules. GURLS and BGURLS – both implemented in Matlab – are aimed at solving learning problems with small/medium and big datasets respectively. GURLS^{++} and BGURLS^{++} are their C^{++}counterparts. Due to programming language constraints, there are some differences between BGURLS and BGURLS^{++}. In Figure 1.1 we depicted the different scenarios where the different packages can be applied and show how the two modules are able to deal with big learning scenarios by means of parallel or serial computing.

The library takes advantage of some favorable properties of regularized least squares algorithm and is tailored to deal in particular with multi-category/multi-label problems. The package comprises useful routines to perform automatic parameter selection and can handle computations with very large matrices by means of both memory-mapped storage and distributed task execution.

GURLS is distributed under the simplified BSD license. Source code, and binaries can be downloaded from `https://github.com/CBCL/GURLS`.

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

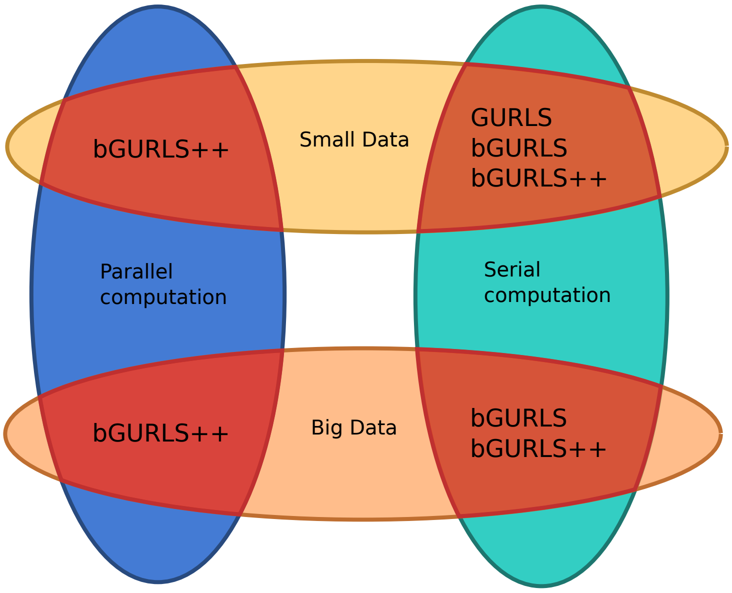

Figure 1.1: GURLS scenarios.

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 the GURLS project, the authors and a link to the research group where the library 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 `GURLSROOT`. Then open MATLAB and execute:

```
>> run('GURLSROOT/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 bGURLS

bGURLS is a pure Matlab library and has no specific dependencies on external libraries, made exception for the GURLS library and 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 `GURLSROOT`. Then open MATLAB and execute:

```
>> run('GURLSROOT/bgurls/utils/bgurls_install.m');
```

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

1.2.4 Installing GURLS^{++}/BGURLS^{++}

GURLS^{++}and BGURLS^{++}are part of the same project, called `gurls`. Users may choose what libraries will be built during project configuration (See section **Configuring** GURLS^{++}/BGURLS^{++} for details).

In the following we assume that the directory where *"gurls++"* and *"bgurls++"* directories reside is named `GURLSROOT`.

GURLS^{++}dependencies

GURLS^{++}depends on several external libraries:

- A Blas/Lapack implementation. Currently we support:
  - AMD's ACML: `http://developer.amd.com/libraries/acml`;
  - ATLAS: `http://math-atlas.sourceforge.net`;
  - 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/`.
  - OpenBLAS: `http://xianyi.github.io/OpenBLAS/`;
- Boost's (v1.46.0 or higher) libraries serialization, date_time, filesystem, unit_test_framework, system, signals.

BGURLS^{++} dependencies

In addition to the GURLS^{++} dependencies, BGURLS^{++}also depends on:

- An MPI implementation. BGURLS^{++}has been successfully tested with MPICH `http://www.mpich.org/`;
- zlib `http://www.zlib.net/`;
- LibHDF5 v1.8.9 `http://www.hdfgroup.org/HDF5/` compiled enabling parallel support and zlib (Libhdf5 v1.8.10 or higher is proved to not work properly);

Automatic dependencies building

The GURLS^{++}/BGURLS^{++}CMake configurator suppports automatic downloading and building of all dependencies by setting the `GURLS_USE_EXTERNAL_*` variables to `ON` (See section **Configuring** GURLS^{++}/BGURLS^{++} below for details).

Installing GURLS^{++}/BGURLS^{++} on Linux

Below we describe how to build and install GURLS^{++}on Ubuntu Linux(tested on Ubuntu 12.04 and Ubuntu 13.10). 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 cmake-curses-gui
   ```

2. To link against some Blas and Lapack implementations you may need a fortran compiler e.g. for gfortran:

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

3. Create a build directory (e.g. "build") for the project

   ```
   $ cd GURLSROOT
   $ mkdir build
   ```

4. 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 the libraries. See the section **Configuring** GURLS^{++}/BGURLS^{++} below for more information on these variables and how to set them to appropriate values.

5. Start building

   ```
   $ make
   ```

6. Install the library(ies) to the path defined at configuration time

   ```
   $ make install
   ```

   The command wil also install to the same path all the dependencies that user chose to build automatically.

Installing GURLS^{++}/BGURLS^{++} on Windows

Below we describe how to build and install GURLS^{++}and BGURLS^{++}on Windows with Visual Studio (tested with VS Express 2010 and 2012).

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

2. Create a build directory (e.g. `GURLSROOT/build`).

3. Run the CMake GUI.
   You will have to set the source directory to the `GURLSROOT` directory, and the build directory to the directory created at the previous step.
   After pressing the configure button you will have to chose the generator for the project (e.g. Visual Studio 10). On Windows you may encounter the error message "error in configuration process, project files may be invalid", check that the you have writing rights to the path specified in the variable `CMAKE_INSTALL_PREFIX`. If this is not the case, change such a variable to a folder to which you have writing right. Now you have to set the values of some variables used for building and installing the libraries according to your preferences (see the section **Configuring** GURLS^{++}/BGURLS^{++} 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.

4. Open the generated solution under Visual Studio and build it.

5. If you used external version of Blas-Lapack, the Openblas DLLs will be automatically copied in the `GURLSROOT/build/bin` folder, please add it to your path to run executables.

6. Optionally install the libraries by explicitly building the install project included in the solution (it is not automatically built when building the solution). Ensure that you have writing rights to the path specified in the variable `CMAKE_INSTALL_PREFIX`.

Configuring GURLS^{++}/BGURLS^{++}

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 'configure', and CMake will try to determine the correct values for all variables. After the first configuration the following variables should be checked:

   - `CMAKE_INSTALL_PREFIX`: The path where the library will be installed to;

- GURLS_BUILD_GURLSPP (ON): Build GURLS^{++}. If set to ON CMake also evaluates the variables
  - GURLSPP_BUILD_DEMO (ON): Enable the building of the GURLS^{++}demo programs;
  - GURLSPP_BUILD_DOC (OFF): Enable the building of the GURLS^{++}documentation using doxygen;
- GURLS_BUILD_BGURLSPP (OFF): Build BGURLS^{++}. If set to ON CMake also evaluates the variables
  - BGURLSPP_BUILD_DEMO (ON): Enable the building of the BGURLS^{++} demo programs;
  - BGURLSPP_BUILD_DOC (OFF): Enable the building of the BGURLS^{++} documentation using doxygen;
- GURLS_USE_BINARY_ARCHIVES (ON): If set to ON, all data structures are stored in binary (rather than text) files, saving storage space and time;
- GURLS_USE_EXTERNAL_BLAS_LAPACK (ON): Enable automatic building of Blas and Lapack using OpenBLAS, under Linux, or automatic linking to the provided OpenBLAS pre-built libraries, under Windows.
- GURLS_USE_EXTERNAL_BOOST (ON): Enable automatic building of boost; if set to OFF you typically need to specify only the variable BOOST_INCLUDE_DIR.
- GURLS_USE_EXTERNAL_HDF5 (ON): Enable automatic building of libHDF5 and its dependencies(MPICH and zlib). Used only if GURLS_BUILD_BGURLSPP is set to ON.

3. For each variable GURLS_USE_EXTERNAL_* which is set to OFF, you must specify the path to the corresponding library. If CMake does not find some required library, an error message will be displayed.

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, OPENBLAS. 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.

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 'generate'. 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.

1.2.5 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);
```

1.2.6 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.readCSV("../data/Xtr.txt");
    ytr.readCSV("../data/ytr_onecolumn.txt");
```

and the test data

```
    Xte.readCSV("../data/Xte.txt");
    yte.readCSV("../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);
```

1.2.7 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 library, 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.

As both GURLS and GURLS++ are similarly designed, in the following we start describing GURLS design and 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. In the remainder of the chapter, a number of commonly used experiments will be described.

2.1 GURLS design

GURLS (GURLS++) basically consists of a set of tasks, each one belonging to a predefined category, and of a method called GURLS *Core* (the `gurls` routine in Matlab, and the `GURLS` class in C++) 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.2.1.

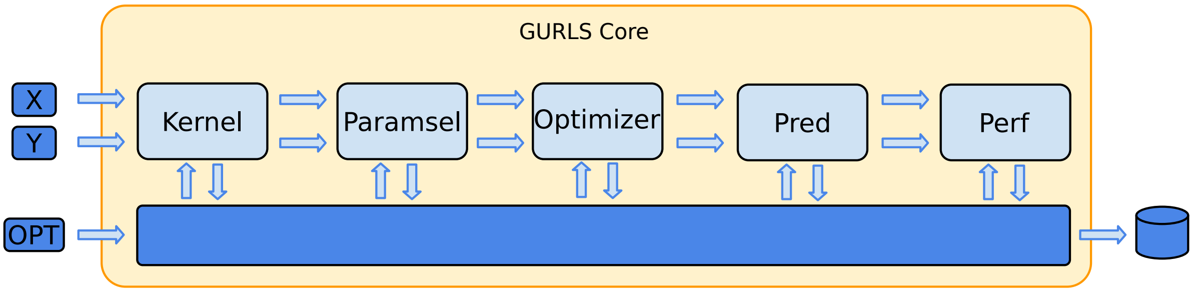

Figure 2.1: GURLS process.

2.2 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 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.

2.3 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 2.3. Type

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

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

2.3.1 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`.

2.3.2 Linear regression, primal case, leave-one-out cv

```
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 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.

2.3.3 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.

2.3.4 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`.

2.3.5 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 [35]. 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.

2.3.6 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 [44]. 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.

2.4 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 2.3.2, 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)
```

2.5 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 GURLS`loocvprimal.cpp`, which implements exactly the first example described in Section 2.2. 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)
    {
        std::cout << "====================================="<< std::endl
        << " Wrong parameters number ("<<argc <<")." << std::endl
        << " Provide a valid path for training, test and output files"
        << "using the following syntax:" << std::endl
        << " \n\n\t " << argv[0] << " xtr xte ytr yte" << std::endl
        <<"==========================================="
        << std::endl << std::endl;
        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.readCSV(xtr_file);
        Xte.readCSV(xte_file);
        ytr.readCSV(ytr_file);
        yte.readCSV(yte_file);

        // specify the task sequence
        OptTaskSequence *seq = new OptTaskSequence();
        *seq << "paramsel:loocvprimal" << "optimizer:rlsprimal"
             << "pred:primal" << "perf:macroavg" << "perf:precrec";

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

        // defines instructions for training process
        OptProcess* process1 = new OptProcess();
        *process1 << GURLS::computeNsave << GURLS::computeNsave
                  << GURLS::ignore << GURLS::ignore << GURLS::ignore;
        process->addOpt("one", process1);

        // defines instructions for testing process
        OptProcess* process2 = new OptProcess();
        *process2 << GURLS::load << GURLS::load << GURLS::computeNsave
                  << GURLS::computeNsave << GURLS::computeNsave;
        process->addOpt("two", process2);

        // build an options' list
        GurlsOptionsList* opt = new GurlsOptionsList("Gurlslooprimal", true);
        opt->addOpt("seq", seq);
        opt->addOpt("processes", process);

        GURLS G;
        string jobId0("one");
        string jobId1("two");

        // run gurls for training
        G.run(Xtr, ytr, *opt, jobId0);

        // run gurls for testing
        G.run(Xte, yte, *opt, jobId1);

    }
    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 << "paramsel:loocvprimal" << "optimizer:rlsprimal"
     << "pred:primal" << "perf:macroavg" << "perf:precrec";
```

and for the sequence of instructions

```
*process1 << GURLS::computeNsave << GURLS::computeNsave
          << GURLS::ignore << GURLS::ignore << GURLS::ignore;
process->addOpt("one", process1);
```

and

```
*process2 << GURLS::load << GURLS::load << GURLS::computeNsave
          << GURLS::computeNsave << GURLS::computeNsave;
process->addOpt("two", 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 << "split:ho"<<"paramsel:siglamho" <<"kernel:rbf"
     << "optimizer:rlsdual" << "pred:dual" << "predkernel:traintest"
     << "perf:macroavg";
```

the code fragment specifying the sequence of instructions for the training process must be

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

and the code fragment specifying the sequence of instructions for the testing process must be

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

2.6 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.

2.7 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)
```

2.8 Learning from large datasets

Two modules in GURLS, namely BGURLS and BGURLS^{++}, have 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. Due to programming language constraints, there are some differences between the Matlab and C^{++}implementations. In Figure **??** we depicted the different scenarios where the different packages can be applied and show how the two modules are able to deal with big learning scenarios by means of parallel or serial computing.

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 or GURLS^{++}. Learning with big data can be carried out with the dedicated

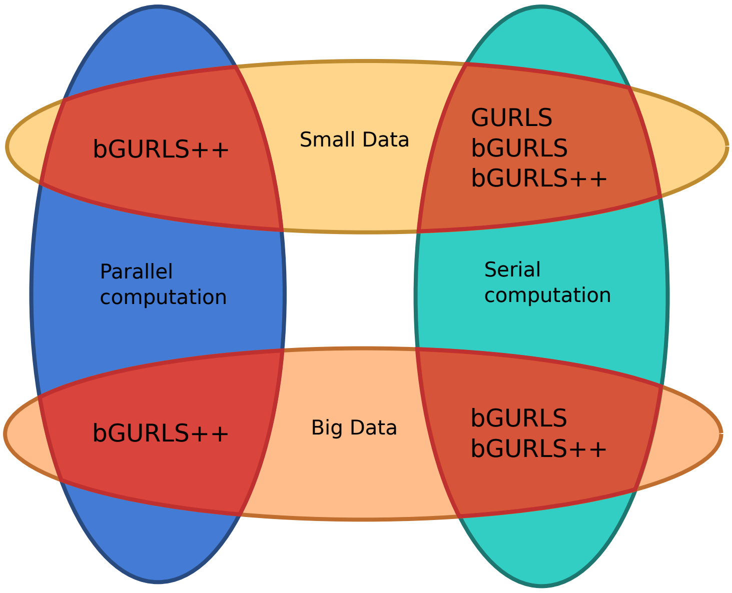

Figure 2.2: GURLS scenarios.

modules BGURLS or 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 GURLS 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. Learning in these conditions can be carried out with the dedicated module BGURLS^{++}.

Both BGURLS and BGURLS^{++} include all the design patterns described for GURLS, and have 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.

2.8.1 BGURLS

BGURLS relies on a simple interface – developed ad-hoc and called *Gurls Distributed Manager* (GDM) – to distribute matrix-matrix multiplications, thus allowing users to perform the important task of kernel matrix computation on a distributed network of computing nodes. After this step, the subsequent tasks behave as in GURLS (cfr. Fig. 2.3).

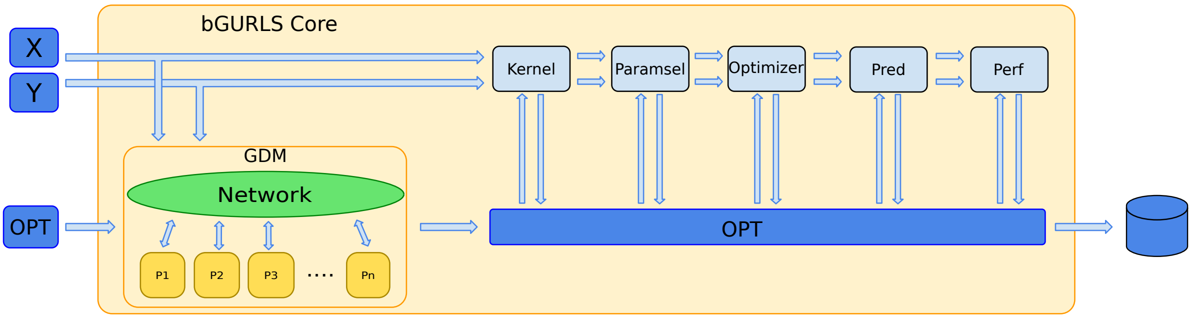

Figure 2.3: BGURLS design.

The BGURLS Core is identified with the `bgurls` command, which behaves as `gurls`. As `gurls` it accepts exactly four arguments:

1. the bigarray of the input data.
2. the bigarray of the labels vector.
3. An options' structure.
4. A job-id number.

The options' structure is built through the `bigdefopt` function with default fields and values. Most of the main fields in the options' structure are the same as in GURLS, however `bgurls` requires the options' structure to have the additional field `files`, which must be a structure with fields:

- `Xva_filename`: the prefix of the files that constitute the bigarray of the input data used for validation
- `yva_filename`: the prefix of the files that constitute the bigarray of the labels vector used for validation
- `pred_filename`: the prefix of the files that constitute the bigarray of the predicted labels for the test set
- `XtX_filename`: the name of the files where pre-computed matrix $X'X$ is stored
- `Xty_filename`: the name of the files where pre-computed matrix $Xt'y$ is stored
- `XvatXva_filename`: the name of the files where pre-computed matrix $X_{va}'X_{va}$ is stored
- `Xvatyva_filename`: the name of the files where pre-computed matrix $X_{va}'y_{va}$ is stored

BGURLS **example**

Let us consider the demo `bigdemoA.m` in the demo directory to better understand the usage of BGURLS. The demo computes a linear classifier with the regularization parameter chosen via hold-out validation, and then evaluate the prediction accuracy on a test set. The data set used in the demo is the `bio` data set used in [26], which is saved in the demo directory as a .zip file, 'bio_unique.zip', containing two files:

- 'X.csv': containing the input $n \times d$ data matrix, where n is the number of samples (24,942) and d is the number of variables (68)
- 'Y.csv': containing the input $n \times 1$ label vector

Note that the bio data is not properly a big data set, as it could reside in memory, however it is large enough to make it reasonable to use BGURLS.

In the following we examine the salient part of the demo in details. First unzip the data file

```
unzip('bio_unique.zip','bio_unique')
```

and set the name of the data files

```
filenameX = 'bio_unique/X.csv'; %nxd input data matrix
filenameY = 'bio_unique/y.csv'; %nx1 or 1xn labels vector
```

Now set the size of the blocks for the bigarrays (matrices of size `blocksize`× `d` must fit into memory):

```
blocksize = 1000;
```

the fraction of total samples to be used for testing:

```
test_hoproportion = .2;
```

the fraction of training samples to be used for validation:

```
va_hoproportion = .2;
```

and the directory where all processed data is going to be stored:

```
dpath = 'bio_data_processed';
```

Now set the prefix of the files that will constitute the bigarrays

```
mkdir(dpath)
files.Xtrain_filename = fullfile(dpath, 'bigarrays/Xtrain');
files.ytrain_filename = fullfile(dpath, 'bigarrays/ytrain');
files.Xtest_filename = fullfile(dpath, 'bigarrays/Xtest');
files.ytest_filename = fullfile(dpath, 'bigarrays/ytes');
files.Xva_filename = fullfile(dpath, 'bigarrays/Xva');
files.yva_filename = fullfile(dpath, 'bigarrays/yva');
```

and the name of the files where pre-computed matrices will be stored

```
files.XtX_filename = fullfile(dpath, 'XtX.mat');
files.Xty_filename = fullfile(dpath, 'Xty.mat');
files.XvatXva_filename = fullfile(dpath,'XvatXva.mat');
files.Xvatyva_filename = fullfile(dpath, 'Xvatyva.mat');
```

We are now ready to prepare the data for BGURLS. The following line of command reads files `filenameX` and `filenameY` blockwise – thus avoiding to load all file at the same time– and stores them in the bigarray format, after having split the data into train, validation and test set

```
bigTrainTestPrepare(filenameX, filenameY,files,blocksize,...
va_hoproportion,test_hoproportion)
```

Bigarrays are now stored in the file names specified in the structure `files`. We can now precompute matrices that will be recursively used in the training phase, and store them in the file names specified in the structure `files`

```
bigMatricesBuild(files)
```

The data set is now prepared for running the learning pipeline with the `bgurls` command. This phase behaves almost completely as in GURLS. The only differences are that:

- we need not to load the data into memory, but simply "load" the bigarray, that is load the information necessary to access the data blockwise.
- we have to specify in the opt structure the path where the already computed matrix multiplications, and bigarrays for validation data are stored.

Let us first define the option structure as in GURLS

```
name = fullfile(wpath,'gurls');
opt = bigdefopt(name);
opt.seq = {'paramsel:dhoprimal','rls:dprimal','pred:primal','perf:macroavg'};
opt.process{1} = [2,2,0,0];
opt.process{2} = [3,3,2,2];
```

Note that no task is defined for the `split` category, as data has already been split in the preprocessing phase and bigarrays for validation were built. In the following fragment of code we add to the options' structure the information relative to the already computed matrix multiplications and to the validation bigarrays

```
opt.files = files;
opt.files = rmfield(opt.files,{'Xtrain_filename';'ytrain_filename';...
'Xtest_filename';'ytest_filename'}); %not used by bgurls
opt.files.pred_filename = fullfile(dpath, 'bigarrays/pred');
```

Note that we have also defined where the predicted labels shall be stored as bigarray.

Now we have to "load" bigarrays for training

```
X = bigarray.Obj(files.Xtrain_filename);
y = bigarray.Obj(files.ytrain_filename);
X.Transpose(true);
y.Transpose(true);
```

and run bgurls on the training set

```
bgurls(X,y,opt,1)
```

In order to run the testing process, we first have to "load" bigarrays variables for test data

```
X = bigarray.Obj(files.Xtest_filename);
y = bigarray.Obj(files.ytest_filename);
X.Transpose(true);
y.Transpose(true);
```

and then we can finally run bgurls on the test set

```
bgurls(X,y,opt,2);
```

Now you should have a mat file named gurls.mat in your path. This file contains all the information about your experiment. If you want to see the mean accuracy, for example, load the file in your workspace and type

```
>> mean(opt.perf.acc)
```

If you are interested in visualizing or printing stats and facts about your experiment, check the documentation about the summarizing functions in the gurls package.

Dealing with other data formats

Other two demos can be found in the 'demo' directory. The three demos differ in the format of the input data, as we tried to provide examples for the most common data formats. The data set used in bigdemoB is again the bio data set, though in a slightly different format as it is already split into train and test data. The bigTrainPrepare and bigTestPrepare take care of preparing the train and test set separately.

The data set used in bigdemoC is the ImageNet data set, which is automatically downloaded from http://bratwurst.mit.edu/sbow.tar, when running the demo. This data set is stored in 1000 .mat files where the i-th file contains the variable x which is a $d \times n_i$ input data matrix for the n_i samples of class i. The bigTrainTestPrepare_manyfiles takes care of preparing the bigarrays for the ImageNet data format. Note that, while the bio data is not properly a big data set, the ImageNet occupies about 1G of RAM and can thus be called a big data set.

In order to run BGURLS on other data formats, one can simply use bigdemoA after having substituted the line

```
bigTrainTestPrepare(filenameX, filenameY,files,blocksize,...
va_hoproportion,test_hoproportion)
```

with a suitable fragment of code. The remainder of the data preparation, that is the computation and storage of the relevant matrices, can be left unchanged.

2.8.2 BGURLS^{++}

BGURLS^{++} offers more interesting features since it's 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. 2.4, allows for the creation of hybrid pipelines comprising serial one-process-based tasks from GURLS^{++}.

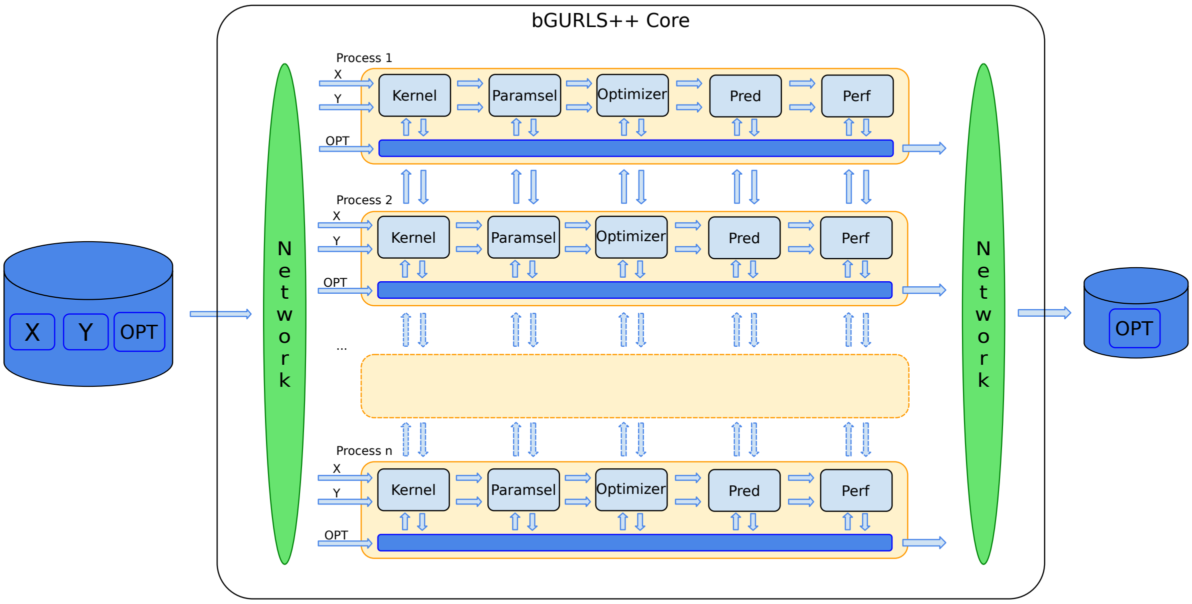

Figure 2.4: BGURLS^{++} design.

BGURLS $^{++}$ example

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

- the Gurls Core is implemented via the BGURLS class instead of the GURLS one;
- 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;

Let us consider the demo bigmedmo.cpp in the demo subdirectory to better understand the usage of the BGURLS $^{++}$ module. This demo implements the same example of the bigdemoB.m of the BGURLS module. The data set used in the demo is the `bio` data set used in [26], 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" << "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 `run` method of the `BGURLS` class as in the following:

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

2.9 Available methods

In this section we summarize all the available tasks that have been implemented in the 4 modules, GURLS, GURLS^{++}, BGURLS 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 2.1: List of GURLS tasks organized by category.

Category	Class	subclasses (task)	
split	Split	Ho	
paramsel	ParamSelection	LoocvDual	LoocvPrimal
		HoDual	HoPrimal
		SiglamHo	Siglam
		FixLambda	FixSigLam
		HoPrimalr	HoDualr
		LooGPRegr	HoGPRegr
		SigLamLooGPRegr	SigLamHoGPRegr
		CalibrateSGD	
kernel	Kernel	ChisquaredKernel	LinearKernel
		RBFKernel	
rls	Optimizer	RLSPrimal	RLSDual
		RLSAuto	RLSPegasos
		RLSPrimalr	RLSDualr
		RLSGPRegr	
predkernel	PredKernel	TrainTest	
pred	Prediction	PredPrimal	PredDual
		PredGPRegr	
perf	Performance	MacroAvg	PrecisionRecall
		Rmse	AbsErr
conf	Confidence	ConfMaxScore	ConfGap
		ConfBoltzmanGap	ConfBoltzman

Table 2.2: List of GURLS^{++} task classes and subclasses.

task category	description	available tasks
bigparamsel	performs selection of the regularization parameter λ	hoprimal calibratesgd
bigrls	solves RLS optimization problem	primal pegasos
bigpred	predicts the labels	primal
bigperf	assess prediction performance	macroavg

Table 2.3: List of BGURLS tasks organized by category.

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

Table 2.4: List of BGURLS++ task classes and subclasses.

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 2.3).

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 2.3). 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 2.3. 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

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

abserr Computes the average absolute error for the predictions, taken as the mean absolute difference 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 (2.0): 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
```

Note that in the above scheme, all the information relative to a field <TASK> previously saved in the input OPT is lost. In order to maintain such information – as in the perf_* routines – the old OPT field must be copied in a new struct, which can then be partially modified by adding or substituting some of its fields.

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 2.3).

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 GurlsOptionsList 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 2.3). 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 << "<CATEGORY1>:<TASK1>" << ... << "<CATEGORYn>:<TASKn>";
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 2.2 and with the only difference that classes and subclasses in C++ are defined in CamelCase.

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

X : input data matrix

y : labels matrix

OPT : options' structure

and have only one method, namely `execute`, that returns a GurlsOptionsList.

4.4 The `GurlsOptionsList` 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
- OptProcess

The `GurlsOptionsList` 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 the string given in input to the `GurlsOptionsLi` constructor): experiment identifier.

- todisk (`OptNumber(1)`): sets whether to save to file the results (1) or not (0), for standard pipeline usage leave it unchanged.
- savefile (`OptString(ExpName.bin)`): 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(20)`): 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 stochastic gradient descent.
- subsize (`OptNumber(50)`): training set size used for parameter selection when using stochastic 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("2.0")`): 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 tasks for the already available categories.

Say you want to add `<NEWTASK>`, for task category `<CATEGORY>`.

4.5.1 Step 1: Updating the factory function of <CATEGORY>

In `<CATEGORY>.h`, where the main class `<CLASS>` is defined, add these lines of code:

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

right before the following fragment of code:

```
        } else
            throw Bad<CATEGORY>Creation(id);
```

in the factory function of the category;

and the following lines:

```
template <typename T>
class <NEWSUBCLASS>;
```

right before:

```
/**
 * \ingroup Exceptions
```

4.5.2 Step 2: Creating the subclass `<NEWSUBCLASS>`

You should now create the subclass `<NEWSUBCLASS>` of `<CLASS>` and modify `gurls.h` in order to include the new class header (`<NEWSUBCLASSFILE>.h`), by adding the following instruction in the proper group of includes:

```
#include "gurls++/<NEWSUBCLASSFILE>.h"
```

The new subclass must be defined in `<NEWSUBCLASSFILE>.h` in the following way:

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

#include // HERE INCLUDE THE HEADER WHERE <CLASS> IS DEFINED
#include // required files

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

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

public:
  /**
   * Default constructor
   */
<NEWSUBCLASS>():<CLASS><T>("<NEWTASK>"){}

 /**
  * Clone method
  */
TaskBase *clone()
{
return new <NEWSUBCLASS><T>();
}
    /**
     * ...
     * \param X input data matrix
     * \param Y labels matrix
     * \param opt options with the following:
     *  - ...
     *
     * \return ret a GurlsOptionsList with the following fields:
     * - ...
     */
    void execute(const gMat2D<T>& X, const gMat2D<T>& Y,
        GurlsOptionsList& opt);
};

template <typename T>
GurlsOptionsList* <NEWSUBCLASS><T>::execute(const gMat2D<T>& X,
     const gMat2D<T>& Y, const GurlsOptionsList &opt) throw(gException)

{

GurlsOptionsList* ret = new GurlsOptionsList("<CLASS>");

/*
 new code here
 ...
 ...
 */

return ret;
}

}
#endif // _GURLS_<NEWSUBCLASSFILE>_H_
```

Note that in the above scheme, all the information relative to a field <CLASS> previously saved in the input opt is lost. In order to maintain such information – as in the Performance class – the old OPT field must be copied in a new GurlsOptionList, which can then be partially modified by adding or substituting some of its fields.

4.5.3 Using GURLS++ custom factory

If you don't want to modify GURLS++ source code while adding new functionalities, you can use Gurls++ custom factory capabilities. In this case, you can skip both the first step previously described, and the modifications to `gurls.h` shown in the second step.

In order to let GURLS++ pipeline understands your newly implemented task, a custom factory can be registered at run-time. To do so, you must define a new factory, and use the method `TaskFactory<T>::registerFactory` before running the pipeline containing the new task.

Here you can find an example of a new factory definition for two new tasks called `MyOptimizer1` and `MyOptimizer2`:

```
template<typename T>
class MyAdditionalOptimizersFactory: public TaskFactory<T>
{
public:
MyAdditionalOptimizersFactory() :TaskFactory<T>(){}
Task<T>* operator()(const std::string& id) throw(BadTaskCreation)
{
            if(id == "myoptimizer1")
                  return new MyOptimizer1<T>();
           if(id == "myoptimizer2")
                  return new MyOptimizer2<T>();
             throw BadTaskCreation(id);
}
};
```

This factory, can be easily registered at run-time calling:

```
TaskFactory<T>::registerFactory("optimizer",
                                new MyAdditionalOptimizersFactory<T>());
```

After these two steps, GURLS++ will accept a pipeline sequence containing either `optimizer:myoptimizer1` or `optimizer:myoptimizer2`.

Remember that if you want to share your new tasks with the GURLS++ community sending a Pull request, you should integrate them into GURLS++ source code following Step 1 and Step 2.

CHAPTER 5

EXPERIMENTS

We focus our experimental analysis on the assessment of GURLS' performance both in terms of accuracy and time. Indeed, we believe it is fair to question whether or not a least square algorithm – classically thought of as an approach to regression problems – can perform well in a classification setting. From a theoretical point of view, the use of a least squares estimator is justified since it provides a plug-in estimator: i.e. least squares estimate conditional probabilities that can be used for classification. From an empirical point of view, there is evidence in the literature that a least squares estimator can perform remarkably well in a variety of tasks.

In order to asses the performance of GURLS we compared it with two popular software packages for classification, namely LIBSVM for we we used the python modular interface [7]- and the matlab toolbox LS-SVM [46]. Due to peculiarities of each package, we separately compared GURLS with each of the aforementioned classification packages. In our experiments we considered 5 popular data sets, briefly described in Table5.1(a).

data set	# of samples	# of classes	# of variables
optdigit	3800	10	64
landsat	4400	6	36
pendigit	7400	10	16
letter	10000	26	16
isolet	6200	26	600

Table 5.1: Data sets description

5.1 GURLS vs LIBSVM

In a first experiment we set up different pipelines with different optimization routines available in GURLS, and compared the performance to SVM, for which we used the python modular interface to LIBSVM [7]. Automatic selection of the optimal regularization parameter is implemented identically in all experiments: (i) split the data; (3) define a set of 20 regularization parameter values on a regular grid; (3) perform hold-out validation. The variance of the Gaussian kernel has been fixed by looking at the statistics of the pairwise distances among training examples. The prediction accuracy of GURLS and GURLS^{++} is identical – as expected – but the latter module is significantly faster. The prediction accuracy of standard RLS-based methods is in many cases higher than SVM. However the computing performance seems to favor SVM. By exploiting the wide range of optimization procedures handled in our library, we further run the experiments with the random features approximation [35] implemented in GURLS. As show in Figure **??**(b), with a relatively small number of random features namely 500, the performance of such method is comparable to that of SVM at much lower computational cost in the majority of the tested data sets.

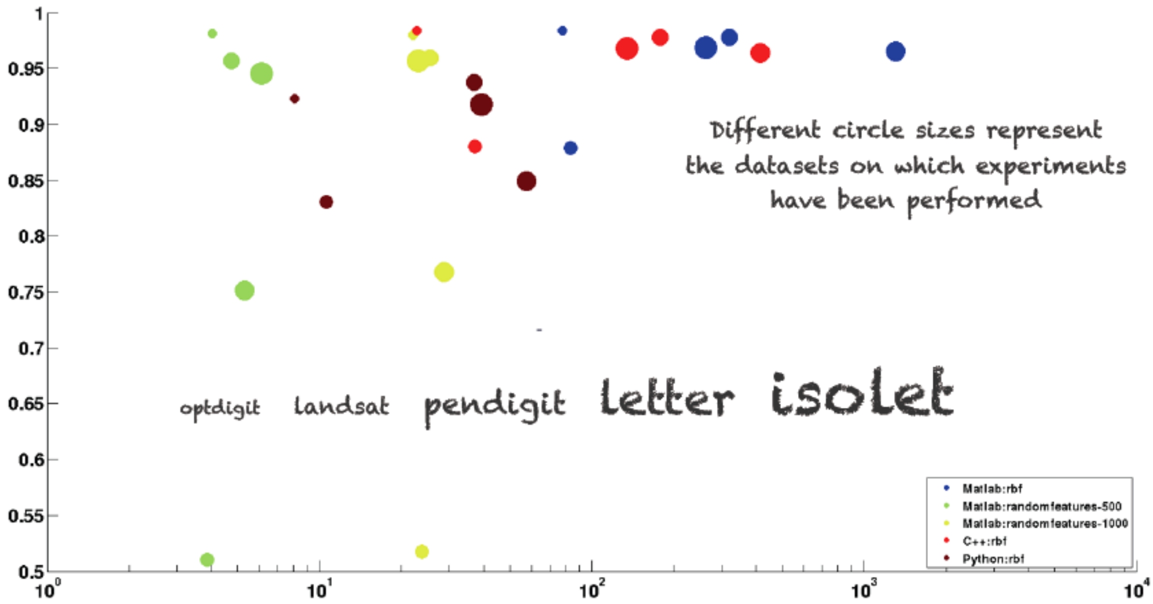

Figure 5.1: Experiments results. For each data set and algorithm/software we plotted a circle is drawn in a 2d-space where the x-axis represents computing time in seconds and the y-axis represents the average prediction accuracy over the classes. For each circle, the size corresponds to the data set, and the color to the algorithm.

5.2 GURLS vs LS-SVM

In a second experiment we compared the performance of GURLS with that of the LS-SVM toolbox [46]. As in LS-SVM one is not allowed to fix a priori the parameter σ or the data splitting, we cannot fairly compare results obtained in the previous experiment with those obtained with LS-SVM. In fact it would not have been fair to use only the optimizer of the LS-SVM toolbox without exploiting its highly optimized routines for parameter selection. As a consequence we run a different comparison where we compare the entire learning pipeline of GURLS with the entire learning pipeline of LS-SVM, using, for each toolbox, its parameter selection routines. For GURLS we performed an exhaustive search over a one- or two-dimensional grid of 20 equispaced values for each parameter. The number of evaluation is thus 20 for linear kernel and 400 for gaussian kernel. For LS-SVM we used the combination of Coupled Simulated Annealing and line/grid search for linear/gaussian, where the former is used to select the range of the parameter values defining the borders of the grid used in the latter step. Differently from GURLS, the grid search is performed recursively on a coarse to fine basis, with stopping rules based on total number of evaluations and tolerance. For the line search parameters, we used the default values, precisely a grid size of 10 elements per dimension, a zooming factor of 2, a tolerance of 0.01, and a maximum number of evaluation set to 20. Similarly, for the grid search parameters, we used the default values, precisely a grid size of 7 elements per dimension, a zooming factor of 5, a tolerance of 0.0001, and a maximum number of evaluation set to 70.

The data sets used for the comparison are the same ones that were used in the first experiment. Experiments were run on a Intel Xeon 5140 @ 2.33GHz processor with 8GB of RAM, and operating system Ubuntu 8.10 Server (64 bit).

Results are reported in Table 5.2, in terms of prediction accuracy and computing time. Note that, we compared only the Matlab implementation of GURLS, as results show that it already has a better computational performance than LS-SVM in most cases. Note that, for sake of completeness we used both primal and dual GURLS implementation of RLS with linear kernel. As expected, the primal formulation, which exploits the lower dimensionality of the input space, is the fastest, however both are significantly faster than LS-SVM. With gaussian kernel, GURLS RLS and LS-SVM have comparable computing time though with a large variability due to the fact that, while in GURLS the number of evaluated parameters pairs is set to 400, in LS-SVM it may vary (and is limited to 70). Moreover, on two of the three tested data sets, with a small loss in prediction performance, the GURLS implementation of the random features algorithm again represents a faster alternative to the highly performing (in terms of prediction) Gaussian kernel RLS and Gaussian kernel LS-SVM.

(a) percentage prediction accuracy

	optdigit	landsat	pendigit
GURLS linear (primal)	92.29	63.68	82.24
GURLS linear (dual)	92.29	66.34	82.46
LS-SVM linear	92.28	64.59	82.31
GURLS 500 rand. feats.	96.75	63.46	96.72
GURLS 1000 rand. feats.	97.54	63.54	95.85
GURLS gauss	98.32	90.37	98.39
LS-SVM gauss	98.26	90.51	98.36

(b) computing time in seconds

	optdigit	landsat	pendigit
GURLS linear (primal)	0.49	0.2215	0.227
GURLS linear (dual)	726	1,148	5,590
LS-SVM linear	7,190	6,526	46,240
GURLS 500 rand. feats.	25.6	27.97	31.6
GURLS 1000 rand. feats.	207	187	199
GURLS gauss	13,500	20,796	100,600
LS-SVM gauss	26,100	18,430	120,170

Table 5.2: Results of experimental comparison between GURLS and LS-SVM.

BIBLIOGRAPHY

[1] Apache Mahout. http://mahout.apache.org/.

[2] Amazon Mechanical Turk. Artificial Artificial Intelligence. https://www.mturk.com/mturk/welcome, 2011.

[3] E.L. Allwein, R.E. Schapire, and Y. Singer. Reducing multiclass to binary: A unifying approach for margin classifiers. *The Journal of Machine Learning Research*, 1:113–141, 2001.

[4] F. Bauer, S. Pereverzev, and L. Rosasco. On regularization algorithms in learning theory. *Journal of complexity*, 23(1):52–72, 2007.

[5] M. Bertero, T.A. Poggio, and V. Torre. Ill-posed problems in early vision. *Proceedings of the IEEE*, 76(8):869–889, 1988.

[6] S. Canu, Y. Grandvalet, V. Guigue, and A. Rakotomamonjy. Svm and kernel methods matlab toolbox. Perception Systèmes et Information, INSA de Rouen, Rouen, France, 2005.

[7] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[8] O. Chapelle and V. Vapnik. Model selection for support vector machines. *Advances in neural information processing systems*, 12:230–236, 1999.

[9] K. Crammer and Y. Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *The Journal of Machine Learning Research*, 2:265–292, 2002.

[10] Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *J. Mach. Learn. Res.*, 2:265–292, March 2002.

[11] T.G. Dietterich and G. Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2(263):286, 1995.

[12] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[13] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[14] R.E. Fan, K.W. Chang, C.J. Hsieh, X.R. Wang, and C.J. Lin. Liblinear: A library for large linear classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

[15] R.E. Fan, K.W. Chang, C.J. Hsieh, X.R. Wang, and C.J. Lin. Liblinear: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[16] Gene H. Golub and Charles F. van Van Loan. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)(3rd Edition)*. The Johns Hopkins University Press, 3rd edition, October 1996.

[17] N. Halko, P. G. Martinsson, and J. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *Society for Industrial and Applied Mathematics*, 53:217–288, 2011.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[19] T. Hastie, S. Rosset, R. Tibshirani, and J. Zhu. The entire regularization path for the support vector machine. *The Journal of Machine Learning Research*, 5:1391–1415, 2004.

[20] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning Theory*. Springer, 2009.

[21] T. Joachims. Making large-scale support vector machine learning practical. 1998.

[22] T. Joachims. Text categorization with support vector machines: Learning with many relevant features. In *European Conference on Machine Learning (ECML)*, pages 137–142, Berlin, 1998. Springer.

[23] T. Joachims. Svmlight: Support vector machine. *SVM-Light Support Vector Machine http://svmlight. joachims. org/, University of Dortmund*, 19, 1999.

[24] T. Joachims. Svmstruct-support vector machine for complex outputs. *URL {http://www. cs. cornell. edu/People/tj/svm_light/svm_struct. html}*, 2011.

[25] Thomas Kailath, Ali H. Sayed, and Babak Hassibi. *Linear Estimation*. Prentice Hall, 2000.

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

[27] F. Lauer and Y. Guermeur. MSVMpack: a multi-class support vector machine package. *Journal of Machine Learning Research*, 12:2269–2272, 2011. http://www.loria.fr/ lauer/MSVMpack.

[28] Yann Lecun and Corinna Cortes. The MNIST database of handwritten digits.

[29] Youssef Mroueh, Lorenzo Rosasco, and Jean-Jacques E. Slotine. Learning multiple classes: Simplex coding and relaxation error analysis. 2011.

[30] E. Osuna and F. Girosi. Reducing the run-time complexity of support vector machines. In *International Conference on Pattern Recognition (submitted)*. Citeseer, 1998.

[31] N. Pinto, Z. Stone, T. Zickler, and D.D. Cox. Scaling-up biologically-inspired computer vision: A case-study on facebook. In *Proc. of Workshop on Biologically Consistent Vision*. CVPR, 2011.

[32] T. Poggio and F. Girosi. Networks for approximation and learning. *Proceedings of the IEEE*, 78(9):1481–1497, 1990.

[33] T. Poggio and S. Smale. The mathematics of learning: Dealing with data. *Notices of the AMS*, 50(5):537–544, 2003.

[34] T. Poggio, V. Torre, and C. Koch. Computational vision and regularization theory. *Nature*, 317(6035):314–319, 1985.

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

[36] Carl Edward Rasmussen and Christopher K. I. Williams. Gaussian processes for machine learning. MIT Press, 2006.

[37] R. Rifkin and A. Klautau. In defense of one-vs-all classification. *The Journal of Machine Learning Research*, 5:101–141, 2004.

[38] R. Rifkin and R. A. Lippert. Notes on regularized least squares. *CSAIL Technical Report*, 2007.

[39] R. Rifkin, G. Yeo, and T. Poggio. Regularized least-squares classification. *Nato Science Series Sub Series III Computer and Systems Sciences*, 190:131–154, 2003.

[40] R.M. Rifkin. *Everything Old Is New Again: A Fresh Look at Historical Approaches in Machine Learning*. PhD thesis, Massachusetts Institute of Technology, 2002.

[41] L. Rosasco, E.D. Vito, A. Caponnetto, M. Piana, and A. Verri. Are loss functions all the same? *Neural Computation*, 16(5):1063–1076, 2004.

[42] Bryan Russell, Antonio Torralba, and William T. Freeman. Labelme. the open annotation tool. http://labelme.csail.mit.edu/, 2005.

[43] Matthias Seeger. Low rank updates for the Cholesky decomposition. Technical report, Department of EECS - University of California at Berkeley, 2008.

[44] 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.

[45] S. Sonnenburg, G. Raetsch, S. Henschel, C. Widmer, J. Behr, A. Zien, F. de Bona, A. Binder, C. Gehl, and V. Franc. The shogun machine learning toolbox. *Journal of Machine Learning Research*, 11:1799–1802, 2010.

[46] J. A K. Suykens, T. Van Gestel, J. De Brabanter, B. De Moor, and J. Vandewalle. *Least Squares Support Vector Machines*. World Scientific Pub. Co., 2002.

[47] J.A.K. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[48] Antonio Torralba. The importance of data. http://people.csail.mit.edu/torralba/courses/6.870_2011f 2011.

[49] J. Weston and C. Watkins. Multi-class support vector machines. 1998.

[50] J. Yang, K. Yu, Y. Gong, and T. Huang. Linear spatial pyramid matching using sparse coding for image classification. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 1794–1801. IEEE, 2009.

[51] X. Zhou, K. Yu, T. Zhang, and T. Huang. Image classification using super-vector coding of local image descriptors. *Computer Vision–ECCV 2010*, pages 141–154, 2010.