Tutorial to GURLS usage for recursive RLS

1 Introduction

Recursive RLS is an efficient algorithm for efficient update of the exact RLS estimator when the data are given sequentially as in online learning. In fact, for any new input-output pair the (regularized) inverse of the kernel matric in the primal space is updated just via matrix/vector multiplication. The basic operations of recursive RLS are sketched in Algorithm 1.1.

Algorithm 1.1 Recursive RLS

```
given:
    the initial RLS estimator $\mathbf{w_{n_0}}$
    the inverse of the (regularized) kernel matrix, $\mathbf{C_{n_0}^{-1}}$
    an order sequence of input-output pairs $\{(\mathbf{x_{n_0+1}}, \mathbf{y_{n_0+1}}), \dots, (\mathbf{x_{n_{max}}}, \mathbf{y_{n_{max}}})\}$
for $\mathbf{n = n_0, ..., n_{max} - 1}$ do
    $$\mathbf{w_{n+1} = w_n + \frac{C_n^{-1}}{1 + x_n^T C_n^{-1} x_n} x_n (y_n - x_n^T w_n)}$$
end for
return $\mathbf{w_{n_{max}}}$
```

2 Design

Recursive RLS can be performed in GURLS (GURLS^{++}) through the tasks (subclasses in GURLS^{++}) `primalrecinit` and `primalrecupdate` of the category (class in GURLS^{++}) `rls`. The workflow for rank 1 update of the RLS estimator in GURLS and GURLS^{++} is depicted in Figure 2. Initial parameter selection and training are carried out on a initial set of samples. The computation of the RLS estimator is carried out by the task `primalrecinit` of the category `rls`, which stores all information necessary for efficient recursive update in the options structure. Once the information about initial training is stored in the options' structure, `opt`, given a new set of input–output pairs (possibly also just one pair), the RLS estimator can be efficiently updated via the task `primalrecupdate` of the category `rls`. Every time a new set of input–output pairs is available, task `primalrecupdate` can be invoked again. Parameter selection and RLS estimation (Retraining in Figure 2) can be repeated after any number of online updates. Finally, the ordinary task `primal` of the prediction category can be used on test data.

Rank 1 update of the RLS estimator is thus efficiently implemented in GURLS thanks to the two new tasks of the category `rls`, `primalrecinit`, and `primalrecupdate`. As all GURLS tasks, `primalrecinit` and `primalrecupdate` take in input:

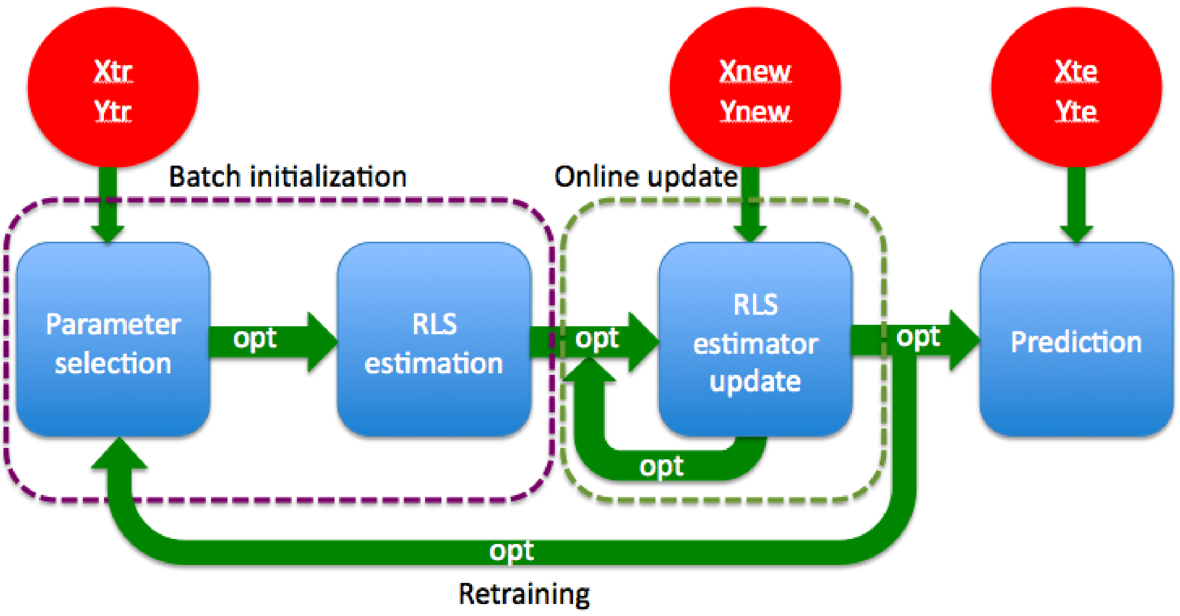

Figure 1: Representation of GURLS design for RLS via recursive rank 1 update or the RLS estimator

- an input data matrix
- a labels matrix
- an options' structure

and return just one output variable, which is a structure containing the fields

- `W`: matrix of coefficient vectors of rls estimator for each class
- `C`: empty matrix
- `X`: empty matrix
- `Cinv`: inverse of the regularized kernel matrix in the primal space

Task `primalrecinit` is indeed almost identical to task `primal` of the `rls` category, which implements the RLS estimator for batch learning. The only difference is in the last two fields of the output structure, `XtX` and `Cinv`, which are not returned by task `primal` of the `rls` category.

2.1 GURLS usage

Two examples of GURLS usage for rank 1 update of the RLS estimator can be found in the demos `demo_recursiveRLS.m` and `demo_recursiveRLSwRetrain.m` in the `demo` directory. The difference between the two demos is in the fact that in the latter retraining is performed after the set of recursive updates, as sketched in Figure 2.1. Let us examine `demo_recursiveRLS.m` first. The demo is structured in three steps, of which the salient part are reported inthe following:

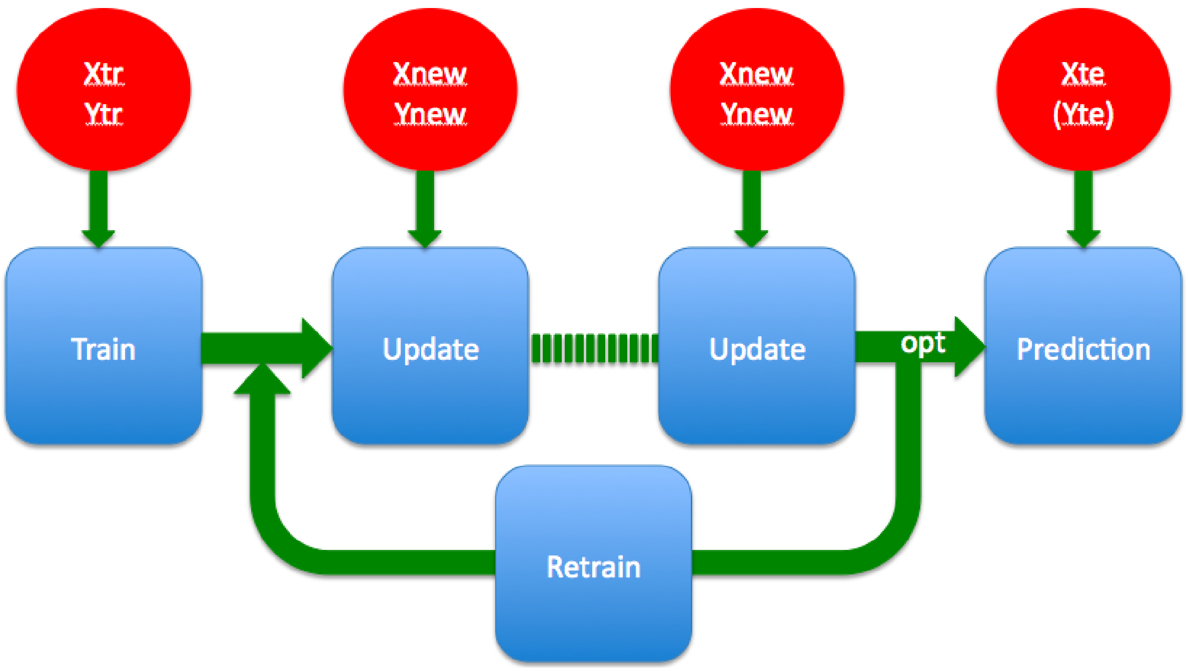

Figure 2: The Train–update–Retrain–Eval workflow for RLS via recursive rank 1 update or the RLS estimator

1. TRAIN: initial parameter selection and training

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.seq = {'split:ho','paramsel:hoprimal','rls:primalrecinit'};
opt.process{1} = [2,2,2];
opt = gurls (Xtr, ytr, opt,1);
```

Notice that while parameter selection can be carried out via any of the tasks for batch parameter selection available in GURLS, for initial estimation of the RLS estimator the specific task `primalrecinit` has to be invoked.

2. UPDATE: recursive update of the estimator, to be repeated every time a new (set of) input–output pairs is available

```
opt.rls = rls_primalrecupdate(Xnew, ynew, opt);
```

3. EVAL: estimate label for a new input point

```
ypred = pred_primal(Xte, yte, opt);
```

Let us now examine `demo_recursiveRLSwRetrain.m` which allows to implement the workflows sketched in Figure 2.1. The demo is structured in four steps, of which the salient part are reported inthe following:

1. TRAIN

```
name = 'ExampleExperiment';
opt = defopt(name);
opt.kernel.XtX = Xtr'*Xtr;
opt.kernel.Xty = Xtr'*ytr;
opt.kernel.n = size(ytr,1);
opt.seq = {'split:ho','paramsel:hoprimal','rls:primalrecinit'};
opt.process{1} = [2,2,2];
opt = gurls (Xtr, ytr, opt,1);
Xva = Xtr(opt.split{1}.va,:);
yva = ytr(opt.split{1}.va,:);
```

Notice that, differently from `demo_recursiveRLSw.m` we here have to store the matrices `XtX, Xty, Xva, yva` later necessary for retraining.

2. UPDATE

```
opt.rls = rls_primalrecupdate(Xnew, ynew, opt);
opt.kernel.XtX = opt.kernel.XtX + Xnew'*Xnew;
opt.kernel.Xty = opt.kernel.Xty + Xnew'*ynew;
opt.kernel.n = opt.kernel.n + 1;

if mod(opt.kernel.n,round(1/hoproportion))==0
        Xva = [Xva; Xnew];
        yva = [yva; ynew];
end
```

3. EVAL

```
ypred = pred_primal(Xte, yte, opt);
```

4. RETRAINING

```
optRetrained = defopt(name);
optRetrained.kernel = opt.kernel;
nva = size(yva);
optRetrained.split{1}.tr = zeros(opt.kernel.n - nva,1);
optRetrained.split{1}.va = 1:nva;
optRetrained.seq = {'paramsel:hoprimal','rls:primalrecinit',...
                   'pred:primal','perf:macroavg'};
optRetrained.process{1} = [2,2,0,0];
optRetrained.process{2} = [3,3,2,2];
optRetrained = gurls (Xva, yva, optRetrained,1);
optRetrained = gurls (Xte, yte, optRetrained,2);
```

Notice that selection of the new regularization parameter is performed via hold-out validation using the subset `Xva,yva` of the total training set as validation set.

2.2 GURLS^{++}usage

As for GURLS, recursive RLS can be performed in GURLS^{++} using the the same pipelines and tasks described in Subsection 2.1. The GURLS tasks `primalrecinit` and `primalrecupdate` of the category `rls` have been implemented in GURLS^{++} as subclasses of the class `rls`.

However, in order to simplify GURLS^{++}usage for recursive RLS, we have implemented two specific classes, namely `RecursiveRLSWrapper` and `RecursiveRLSRetrainWrapper`, which methods `train`, `update`, `eval`, and `retrain` (the last one only for `RecursiveRLSRetrain`) are user friendly wrappers of the set of pipelines and tasks that implement the 3 and 4 steps described in Subsection 2.1.

2.2.1 Usage of Class **`RecursiveRLSWrapper`**

For performing recursive RLS without retraining class `RecursiveRLSWrapper` can be used (class `RecursiveRLSRetrainWrapper` can be used as well but unnecessary computations and data storage are performed). Initial training on a set of input–output matrices `(Xtr,ytr)` is performed as follows:

```
RecursiveRLSWrapper* wrapper;
wrapper = new RecursiveRLSWrapper("recursiveRLS");
wrapper->train(Xtr, ytr);
```

then, given a new input–output pair `(Xnew,ynew)`, the estimator can be updated with the following line of code

```
wrapper->update(Xnew,Ynew)
```

finally, the obtained estimator can be used to predict the label of a new input matrix `Xte`

```
gMat2D<T>* rec_result =  wrapper->eval(Xte)
```

Prediction can be also performed iteratively row-by-row

```
for(unsigned long i=0; i<nte; ++i)
{
       getRow(Xte_tot.getData(), nte, d, i, Xnew.getData());

       unsigned long index;
       double ret = wrapper->eval(Xnew, &index);
}
```

2.2.2 Usage of Class **`RecursiveRLSRetrainWrapper`**

For allowing retraining, class `RecursiveRLSRetrainWrapper` shall be used. Methods `train`, `update`, `eval` can be used as the corresponding methods of calss `RecursiveRLSWrapper`:

```
RecursiveRLSWrapper* wrapper;
 wrapper = new RecursiveRLSRetrainWrapper("recursiveRLSRetrain");
 wrapper->train(Xtr, ytr);
 wrapper->update(Xnew,Ynew)
gMat2D<T>* rec_result =  wrapper->eval(Xte)
```

whereas retraining can be carried out invoking

```
    dynamic_cast<RecursiveRLSRetrainWrapper*>(wrapper)->retrain();
```

without any input, as all the input–output pairs which have been previously fed to the wrapper either via method `train` or method `update` are used for parameter selection.

A demo showing the usage of these wrappers can be found in `RecursiveRLS.cpp` in the demo directory.

3 Experiments

We have used the GURLS implementation of the rank 1 update of the RLS estimator on the `bio` data set used in [1], which is stored into four files:

- 'Xtrain.csv': containing the training input $n \times d$ data matrix, where n is the number of samples (12,471) and d is the number of variables (68)
- 'Ytrain.csv': containing the input $n \times 1$ label vector
- 'Xtest.csv': containing the test input $n_{test} \times d$ data matrix, with $n_{test} = 12,471$.
- 'Ytest.csv': containing the input $n_{test} \times 1$ label vector

We used the first n_0 samples of the training data for initial parameter selection and training. Recursive rank 1 update of the RLS estimator is performed on the remaining training samples. Online learning behavior is simulated by reading blocks of new samples from the files 'Xtrain.csv' and 'Ytrain.csv', and sequentially invoking task `rls:primalrecupdate` on each set of samples, until all lines of the files have been read. Online learning is performed for increasing value of n_0, precisely from 10 to 200 increasing of 10 samples each time. On the same data set, batch learning is performed by training an RLS estimator from the entire training set in batch mode, using the same parameter selected in the initial parameter selection of online RLS. The experiment is repeated 10 times, each time after random permutation of the training set samples. We evaluated the mean difference between accuracies of online and batch RLS, and the mean frobenius norm of the difference of the test labels predicted by the two methods. While accuracies are identical, the values of the the predicted test labels (before class assignment) are slightly different the difference between the estimated test labels, though already very small for small n_0, decreases with n_0, and stabilizes for n_0 is significantly bigger than the number of features, 68. Such a slight difference may be due to the, when $n_0 < d$ inversion of the kernel matrix in the primal space may lead to some instability.

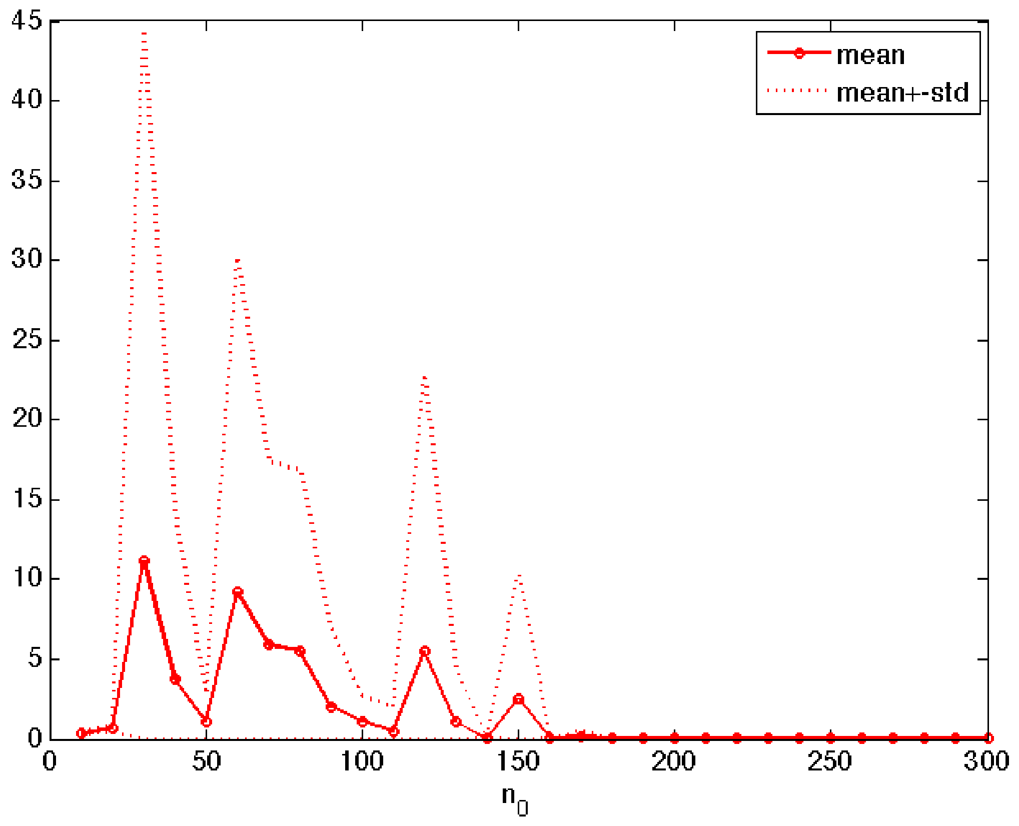

Figure 3: Frobenius norm of the difference between test labels predicted by online and batch RLS estimators.

Bibliography

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