Tutorial to GURLS^{++} wrappers

1 The `GurlsWrapper` class

We developed a set of simple API's for specific learning pipelines, that simplify GURLS^{++} usage for specific cases. All the available API's have been implemented through the class `GurlsWrapper` which has two main methods:

- **train**

  ```
  void train(const gMat2D<T> &X, const gMat2D<T> &y);
  ```

  which takes in input the $n \times d$ inputs matrix, `X` and the $n \times T$ labels matrix, `Y`, and performs parameter selection and computes the model.

- **eval**

  ```
  T eval(const gVec<T> &Xte, unsigned long *index = NULL);
  ```

  takes in input the $n_{te} \times d$ test inputs matrix, `Xte`, and returns an $n_{te} \times T$ matrix (vector) of the estimated labels.

The design of GURLS^{++} wrapper is sketched in Figure 1.

1.1 Available wrappers

The available GURLS^{++} wrappers are organized by class hierarchy in Figure 2.

Subclasses `RLSWrapper` and `RecursiveRLSWrapper` allow training (generalized) linear models. The subclass `KernelWrapper` includes all wrappers that make use of kernel methods and thus allow training nonlinear models.

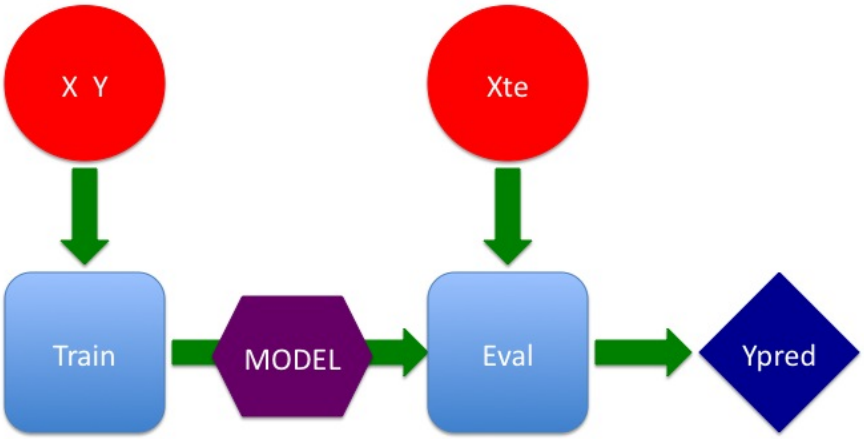

Figure 1: Representation of GURLS^{++}wrappers design

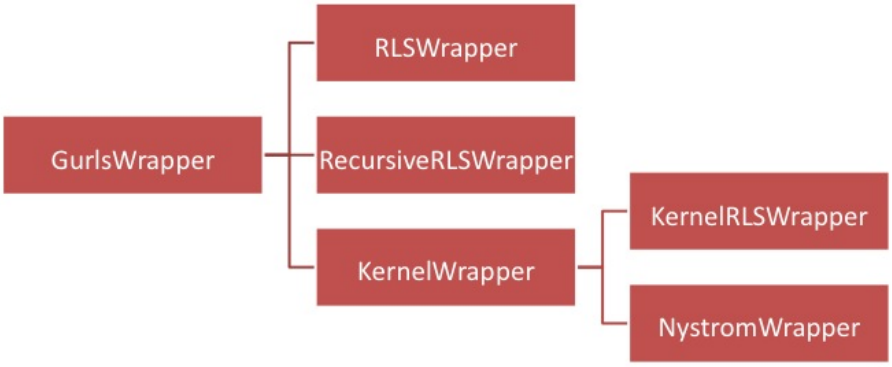

Figure 2: Available GURLS^{++}wrappers

2 Usage

Let us consider a generic subclass of the class `GurlsWrapper`, namely `<WrapperSubClass>`. Training on a set of input–output matrices `(X,Y)` is performed as follows:

```
WrapperSubClass<T> wrapper ("wrapper_name");
wrapper.train(X,Y);
```

then, the obtained estimator can be used to predict the labels for a new set of input data `Xte` through the following line of code

```
gMat2D<T>* Ypred =  wrapper.eval(Xte);
```

2.1 Configuration methods

Class `GurlsWrapper` has a set of additional methods that allow configuring the parameters of the learning process. The methods the user needs to know are the following:

- `void setSplitProportion(double value(0.2));`
  sets the fraction of data to be used as hold-out set in parameter selection.

- `void setProblemType(typename value(CLASSIFICATION));`
  sets the problem to either regularization or classification.

- `void setNparams(unsigned long value(20));`
  sets the number of regularization parameter to be assessed in parameter selection.

- `void setParam(double value);`
  avoids performing parameter selection and uses as regularization parameter the value given in input.

In addition subclass `KernelWrapper` has four specific methods:

- `void setKernelType(typename value(linear));`
  sets the type of kernel to be used.

- `void setSigma(double value);` avoids performing parameter selection for the kernel parameter and uses as kernel parameter the value given in input.

- `void setNSigma(unsigned long value(25));`
  sets the number of kernel parameter to be assessed in parameter selection.

2.2 I/O methods

All classes share two methods for saving and loading models:

- **saveModel**

  ```
  void saveModel(const std::string &fileName);
  ```

  saves the computed model to the file specified in `fileName`.

- **loadModel**

  ```
  void loadModel(const std::string &fileName);
  ```

  loads a previously computed model saved in the file specified in `fileName`.

2.3 Regularized Least Squares

Subclass `RLSwrapper` implements a linear Regularized Least Squares model.

2.3.1 Example

Train a linear classifier by learning the optimal regularization parameter via hold-out parameter selection on a hold-out set which is 10% of the input training data by choosing amongst 10 values of the regularization parameter. Then predict the label of a new set of input data

```
RLSWrapper<T> wrapper ("wrapper_name");
wrapper.setNparams(10)
wrapper.setSplitProportion(.1)
wrapper.train(X, y);
wrapper.eval(Xte);
```

2.4 Recursive RLS through rank-one update

Subclass `RecursiveRLSwrapper` implements a linear Regularized Least Squares, where optimization is carried out recursively on the training data through rank-one update. 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 matrix in the primal space is updated just via matrix/vector multiplication. The basic operations of recursive RLS are sketched in Algorithm 1.

Algorithm 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 $\{(\mathbf{x_{n_0+1}}, \mathbf{y_{n_0+1}}), \ldots, (\mathbf{x_{n_{max}}}, \mathbf{y_{n_{max}}})\}$
for $\mathbf{n = n_0, ..., n_{max} - 1}$ **do**

$$\mathbf{w_{n+1}} = \mathbf{w_n} + \frac{\mathbf{C_n^{-1}}}{\mathbf{1 + x_n^T C_n^{-1} x_n}} \mathbf{x_n}(\mathbf{y_n} - \mathbf{x_n^T w_n})$$

end for
return $\mathbf{w_{n_{max}}}$

2.4.1 `RecursiveRLSwrapper`

In addition to methods `Train` and `Eval`, class `RecursiveRLSWrapper` has the following specific methods

- **Update**

  ```
  void update(const gVec<T> &X, const gVec<T> &y);
  ```

  which takes in input a $1 \times d$ inputs vector, `X`, and a $1 \times T$ labels vector, `Y`, and update the model via rank-one update

- **Retrain**

  ```
  void retrain();
  ```

  which performs parameter selection on all the received data (the initial training data and the data used to update the model) and returns a new model

2.4.2 Example

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

Initial training is performed through the following code fragment

```
RecursiveRLSWrapper<T> wrapper ("recursiveRLS");
wrapper.train(Xtr, ytr);
```

which stores all information necessary for efficient, then, given a new input–output pair `(Xnew,ynew)`, the estimator can be updated with the following line of code

```
wrapper.update(Xnew,Ynew);
```

whereas retraining can be carried out invoking

```
wrapper.retrain();
```

without any input, as parameter selection is carried out on all the input–output pairs which have been previously given as input to the wrapper either via method `train` or method `update`. Finally, the obtained estimator can be used to predict the label of a new input matrix `Xte`

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

2.5 KernelRLSWrapper

Subclass `KernelRLSwrapper` trains a possibly non linear model by resorting to kernel methods. The default kernel is the linear kernel. Parameter selection is carried out for both the regularization parameter λ and the kernel parameter σ (if any).

2.5.1 Example

Train a nonlinear classifier by learning both the the optimal regularization parameter and kernel parameter via hold-out parameter selection by choosing amongst 20 values of the regularization pa- rameter (default) and 10 values of the kernel parameter (set through method `setNsigma`). The Guassian kernel is used (set through method `setKernelType`. Then predict the label of a new set of input data

```
KernelRLSWrapper<T> wrapper ("wrapper_name");
wrapper.setKernelType(RBF);
wrapper.setNsigma(10);
wrapper.train(X, y);
wrapper.eval(Xte);
```

2.6 NystromWrapper

Subclass `NystromWrapper` allows to train a possibly non linear model for large data sets, for which the complete $n \times n$ kernel matrix may not fit into RAM. `NystromWrapper` implements Nystrom Reguarization, which solves the kernel Least Square problem approximately, by replacing the kernel matrix with a randomized low rank approximation. The default kernel is the Gaussian kernel. The regularization parameter coincides with the rank, m of the kernel approximation. Parameter selection is carried out for the regularization parameter m, whereas the kernel parameter σ must be manually set, otherwise the default value (d, that is the number of features) is used.

2.6.1 Example

Train a nonlinear classifier with gaussian kernel for a manually given value of the kernel parameter. The Guassian kernel is used (set through method `setKernelType`. The optimal rank is chosen by hold-out cross-validation on the range [1,`MaxParam`], where `MaxParam` is given in input using `setParam`. Then predict the label of a new set of input data

```
NystromWrapper<T> wrapper ("wrapper_name");
wrapper.setKernelType(RBF);
wrapper.setParam(100);
wrapper.train(X, Y);
wrapper.eval(Xte);
```