A (Partial) Documentation of RBF & AdaBoost$_{Reg}$ Software Packages

Gunnar Rätsch

January 4, 2001

1 Overview

The RBF and AdaBoost$_{Reg}$ packages consists out of eight classes:[1]

- The data storage classes: `data, data_w`,
- the abstract learner classes: `learner, learner_w`,
- an implementation of an RBF network `rbf_net_w` and
- some classes for ensemble learning: `booster_base, adabooster, adabooster_regul`.

All classes are implemented in MATLAB and should work with MATLAB R11 and R12 on almost any platform. The class hierarchy can be found in Figure 1.

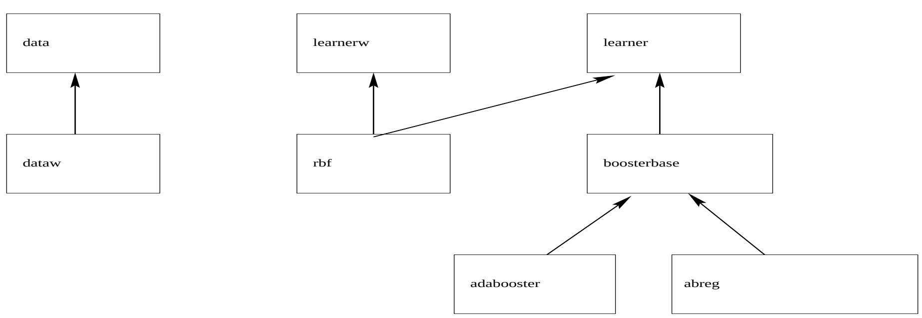

Figure 1: Class-hierarchy of the classes in this package

[1]Please read the licensing and (no) warranty terms in Appendix A.3.

2 The data storage classes

2.1 Class data: training, validation & test set

The class `data` implements the basic functions for managing data sets consisting of training, test and validation set. The methods of the class `data` are:

- `dataset=data(trainpat, traintarg, testpat, testtarg, valpat, valtarg);`
  This constructor creates a `data` object with training set (`trainpat, traintarg`), test set (`testpat, testtarg`) and validation set (`valpat, valtarg`).
  Example:

```
>> X=rand(1,200) ; Y=2*(X<0.3)-1 ;
>> dataset=data(X(1,1:100), Y(1,1:100), ...
                X(1,101:200), Y(1,101:200))
data object
the sname is : none
the nsname is: none_tr100_v0_t100
  number of train patterns     : 100
  number of test patterns      : 100
  number of validation patterns: 0
  input dimension      : 1
  output dimension     : 1
training data has not zero mean
training data has not standard deviation one
```

- `[trainpat, traintarg]=get_train(dataset, order);`
  `[testpat, testtarg]=get_test(dataset,order);`
  `[valpat, valtarg]=get_val(dataset,order);`
  These methods extract the data stored in the `data` object. The first argument is the object created with the constructor above. If the parameter `order` is not specified or `order=1`, then the return arguments are `[{train,test,val}pat, {train,test,val}targ]`. If `order=2`, the only `[{train,test,val}targ]` is returned.

- `numpat=get_train_size(dataset);`
  `numpat=get_test_size(dataset);`
  `numpat=get_val_size(dataset);`
  Returns the number of samples of training, test or validation set for the given `data` object.

- `odim=get_odim(dataset);`
  `idim=get_idim(dataset)`
  Returns the input or output dimension (`odim` should be 1) of the training, test and validation set (should be the same for all three sets).

- `dataset=split_train(dataset, testsize, valsize);`
  Splits up the training set into training, test and validation set. `testsize`

and `valsize` specify the size of the test and validation set after splitting. The rest is used as training data.

- `dataset=normalize(dataset) ;`
  Linearly transforms the data, such that the training set has zero-mean in all dimensions. The same transformation is applied to test and validation data.

- `dataset=standardize(dataset) ;`
  Linearly transforms the data, such that the training set has zero-mean and a standard deviation of 1 in all dimensions. The same transformation is applied to test and validation data.

2.2 Class `data_w`: weighted training data

The class `data_w` extends the functionality of class `data` to allow weighted training sets as often used in Boosting/Ensemble learning methods. The methods of the class `data_w` are:

- `dataset=data_w(dataset)`
  This constructor creates a `data_w` object from an existing `data` object. Example:

```
>> X=rand(1,200) ; Y=2*(X<0.3)-1 ;
>> dataset=data(X(1,1:100), Y(1,1:100), ...
                X(1,101:200), Y(1,101:200)) ;
>> datasetw=data_w(dataset) ;
```

- `datasetw=set_sampl_weights(datasetw, weights)`
  `weights=get_sampl_weights(datasetw)`
  Sets or gets the weights associated to the training set. The parameter `weights` is a row-vector with length `get_train_size(datasetw)`.

3 Abstract Learner Classes

3.1 Class `learner`

The class `learner` implements the abstract functionality of any "learner". The methods are:

- `lrn=learner(idim, odim);`
  This constructor creates a `learner` object for data of input dimension `idim` and output dimension `odim`. `idim` and `odim` default to 1, if not given.

- `lrn=do_learn(lrn, dataset);`
  This is an abstract function which any derived class has to implement/overload. The learner `lrn` is given a dataset and learns from the data. It may use the training and validation set only.

- `output_data=calc_output(lrn, in_data);`
  This is an abstract function which any derived class has to implement/overload. After calling `do_learn`, one may use `calc_output` to compute the predictions of the learner based on the training/validation data.
  Example:

```
>> X=rand(2,200) ; Y=2*(X(1,:)+X(2,:)<0.7)-1 ;
>> dataset=data(X(1,1:100), Y(1,1:100), ...
                X(1,101:200), Y(1,101:200)) ;
>> lrn= ... % some derived class from learner
>> lrn=do_learn(lrn,dataset) ;
>> output=calc_output(lrn, rand(1,200)) ;
```

- `[trErrs, tstErrs, valErrs]=get_class_errors(lrn, dataset);`
  Computes the training, test and validation classification error rates if the learner is used for classification.

- `[trErrs, tstErrs, valErrs]=get_mse(lrn, dataset);`
  Computes the training, test and validation mean squared error if the learner is used for regression.

3.2 Class `learner_w`: Learning weighted data

The class `learner_w` has the functionality needed for weighted training sets. It is not derived from `learner`. Any learner for weighted training sets should be derived from `learner` *and* `learner_w`. The methods are:

- `lrnw=learner_w;`
  Constructs the `learner_w` object.

- `weights=get_distr(lrnw);`
  `lrnw=set_distr(lrnw, weights);`
  Methods for setting and getting the distribution/weighting to the learner.

- `data_w_verify(lrnw, dataset)`
  Checks whether the given data set is a `data_w` object that eventually could be used for learning by an `learner_w` object.

4 The RBF Network Class

4.1 Class `rbf_net_w`

The class `rbf_net_w` implements the algorithm given in Appendix A.1. It has several methods for internal use only. The steps given in the pseudo-code can be mapped to methods as follows: Initialization: `cluster, private/clustknb_new_w`;
1. `calc_weights, private/update, private/ls_solve_w, private/design_rbf`;

2a. private/rbfgrad_w; 2b. private/optimize; 3a. private/linmin, private/mnbrak, private/brent and 3b. private/optimize.

The class rbf_net_w is derived from learner and learner_w. The methods to be used are:

- lrn=rbf_net_w(numcen, lambda, idim, odim);
  This constructor creates a rbf_net_w object with numcen centers and λ =lambda.
- lrn=set_max_iter(lrn, maxiter);
  Sets the maximum number of CG iterations (cf. Figure 2). This is the parameter which influences the learning speed most (default: 10).
- lrn=do_learn(lrn, dataset, do_cluster);
  This method overloads the abstract function defined in class learner. The learner lrn is given a dataset and learns from the given data. The parameter do_cluster is a boolean variable determining whether the centers should be initialized via K-means clustering (strongly recommended).
- output_data=calc_output(lrn, in_data);
  This method overloads the abstract function defined in class learner. After calling do_learn, one may use calc_output to compute the predictions of the learner based on the training/validation data.

Example:

```
>> X=rand(2,200) ; Y=2*(X(1,:)+X(2,:)<0.7)-1 ;
>> dataset=data(X(1,1:100), Y(1,1:100), ...
                X(1,101:200), Y(1,101:200)) ;
>> lrn=rbf_net_w(3, 1e-3, 2, 1) ; % 3 centers and lambda=0.001
>> lrn=do_learn(lrn, dataset, 1) ;
>> [trErr,teErr]=get_class_errors(lrn, dataset)
trErr =
    0.1500
teErr =
    0.2300
```

5 The Ensemble Learning Classes

5.1 Class booster_base: the basis

The class booster_base implements the basic functionality for all ensemble learning classes. An object of this class stores a prototype ("base learner") of a learner object (base learner), an array of learner objects that are already trained ("base hypothesis") and some additional parameters like the number of iterations. The most important methods are:

- `bb=booster_base(prototype, boost_steps, param1, param2);`
  Creates a `booster_base` object. The parameter `prototype` is an object derived from `learner` and optionally derived from `learner_w` (e.g. `rbf_net_w`). The parameter `boost_steps` determines the number of base hypothesis that should be combined. `param1` and `param2` are optional parameters that are given to the method `do_learn` of the base learner.

- `wl=train_weak(bb, dataset);`
  Calls **do_learn** of the prototype and returns the trained `learner` object (base hypothesis).

- `weights=get_vote_weights(bb, idx);`
  `bb=set_vote_weights(bb, weights, idx);`
  Method for getting and setting the weights for linear combination of the base hypotheses.

- `lrn=get_boosted_learner(bb, idx);`
  `bb=set_boosted_learner(bb, lrn, idx);`
  Method for getting and setting the base hypothesis (objects of class `learner`) for linear combination.

5.2 Class `adabooster`: the original AdaBoost algorithm

The class `adabooster` is derived from `booster_base` and implements the original AdaBoost algorithm [2] (cf. pseudo-code in Figure 3). It has several methods for internal use only. The steps given in the pseudo-code can be mapped to methods as follows: Initialization: `init_learn`; 1. `train_week, do_learn`; 2.&3. `comp_weight, do_learn` and 4. `comp_distr, do_learn`.

- `bb=adabooster(proto, booststeps, param1, param2);`
  Constructor for `adabooster` objects (cf. `booster_base`).

- `bb=do_learn(bb, dataset);`
  Implements the AdaBoost algorithm (cf. Figure 3 and `learner/do_learn`).

- `weights=comp_distr(bb, b_t, output, dataset, weights, Prot, t);`
  Computes the new pattern distribution using the previous weights (`weights`), the output of the previous base hypothesis (`output`) and the weight of the last base hypothesis (`b_t`).

- `[bb, b_t]=comp_weight(bb, t, output, dataset, weights, EpsT);`
  Computes the weight `b_t` of the current base hypothesis based on its output (`output`) on the training set, the previous pattern weights (`weights`) and the weighted classification error (`EpsT`).

- `Prot=report(bb, t, EpsT, weights, dataset, Prot);`
  This function is called in each iteration and can be used to make some outputs and/or to record some variables stored in the variable `Prot` for later analysis.

- `id=get_use_sign_output(bb);`
  `bb=set_use_sign_output(bb, id);`
  Sets or gets how the outputs of the base hypothesis are transformed: 0 - no transformation; 1 - signum function mapping to $\{-1, +1\}$ and 2 - sigmoidal transformation to [-1..+1].

- `bb=finish_learn(bb);`
  Cleans up after learning.

Example:

```
>> X=rand(2,200) ; Y=2*(X(1,:)+X(2,:)<0.7)-1 ;
>> dataset=data(X(1,1:100), Y(1,1:100), ...
                X(1,101:200), Y(1,101:200)) ;
>> lrn=rbf_net_w(3, 1e-3, 2, 1) ; % 3 centers and lambda=0.001
>> bb=adabooster(lrn, 30, 1) ; % 30 iterations
>> bb=do_learn(bb,dataset) ;
>> [trErr,teErr]=get_class_errors(bb, dataset)
trErr =
    0.1100
teErr =
    0.2100
```

5.3 Class `adabooster_regul`: the regularized algorithm

This class is derived from `adabooster` and just adds/overloads some functionality. The algorithm implemented in this class is given in Figure 4 as pseudo-code.

- `bb=adabooster_regul(proto, booststeps, phi, C, param1, param2);`
  The constructor for this class. The parameter `phi` modifies the error function (details are given in [7], $\phi = \frac{1}{2}$ is a reasonable choice). The parameter C is the regularization parameter: $C = 0$ leads to the original AdaBoost algorithm. Large C means a "very soft margin".

The other functions e.g. `do_learn, comp_distr` and `comp_weight` work as before – they just contain slightly different formulas.

A Appendix

A.1 RBF nets with adaptive centers

The RBF nets used in the experiments are an extension of the method of [3], since centers and variances are also adapted (see also [1, 4]). The output of the network is computed as a linear superposition of K basis functions

$$f(\mathbf{x}) = \sum_{k=1}^{K} w_k g_k(\mathbf{x}) \ , \tag{1}$$

where w_k, $k = 1, \ldots, K$, denotes the weights of the output layer. The Gaussian basis functions g_k are defined as

$$g_k(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mu_k\|^2}{2\,\sigma_k^2}\right), \tag{2}$$

where μ_k and σ_k^2 denote means and variances, respectively. In a first step, the means μ_k are initialized with K-means clustering and the variances σ_k are determined as the distance between μ_k and the closest μ_i ($i \neq k, i \in \{1, \ldots, K\}$). Then in the following steps we perform a gradient descent in the regularized error function (weight decay)

$$E = \frac{1}{2}\sum_{i=1}^{l}(y_i - f(\mathbf{x}_i))^2 + \frac{\lambda}{2l}\sum_{k=1}^{K} w_k^2. \tag{3}$$

Taking the derivative of (3) with respect to RBF means $\mu_{\mathbf{k}}$ and variances σ_k we obtain

$$\frac{\partial E}{\partial \mu_k} = \sum_{i=1}^{l}(f(\mathbf{x}_i) - y_i)\frac{\partial}{\partial \mu_k} f(\mathbf{x}_i), \tag{4}$$

with $\frac{\partial}{\partial \mu_k} f(\mathbf{x}_i) = w_k \frac{\mathbf{x}_i - \mu_k}{\sigma_k^2} g_k(\mathbf{x}_i)$ and

$$\frac{\partial E}{\partial \sigma_k} = \sum_{i=1}^{l}(f(\mathbf{x}_i) - y_i)\,\frac{\partial}{\partial \sigma_k} f(\mathbf{x}_i), \tag{5}$$

with $\frac{\partial}{\partial \sigma_k} f(\mathbf{x}_i) = w_k \frac{\|\mu_k - \mathbf{x}_i\|^2}{\sigma_k^3} g_k(\mathbf{x}_i)$. These two derivatives are employed in the minimization of (3) by a conjugate gradient descent with line search, where we always compute the optimal output weights in every evaluation of the error function during the line search. The optimal output weights $\mathbf{w} = [w_1, \ldots, w_K]^\top$ in matrix notation can be computed in closed form by

$$\mathbf{w} = \left(G^T G + 2\frac{\lambda}{l}\mathbf{I}\right)^{-1} G^T \mathbf{y}, \quad \text{where} \quad G_{ik} = g_k(\mathbf{x}_i) \tag{6}$$

and $\mathbf{y} = [y_1, \ldots, y_l]^\top$ denotes the output vector, and $\mathbf{I}$ an identity matrix. For $\lambda = 0$, this corresponds to the calculation of a pseudo-inverse of G.

So, we simultaneously adjust the output weights and the RBF centers and variances (see Figure 2) for pseudo-code of this algorithm). In this way, the network fine-tunes itself to the data after the initial clustering step, yet, of course, overfitting has to be avoided by careful tuning of the regularization parameter, the number of centers K and the number of iterations (cf. [1]). In our experiments we always used $\lambda = 10^{-6}$ and up to ten CG iterations.

A.2 AdaBoost & AdaBoost-Reg

I am not going to explain these algorithms here and just give the pseudo-code of them. For details see e.g. [2] and [7].

Algorithm RBF-Net(K, λ, O)

Input:

Sequence of labeled training patterns $\mathbf{Z} = \langle(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_l, y_l)\rangle$

Number of RBF centers K

Regularization constant λ

Number of iterations O

Initialize:

Run K-means clustering to find initial values for μ_k and determine $\sigma_k, k = 1, \ldots, K$, as the distance between μ_k and the closest μ_i $(i \neq k)$.

Do for $o = 1 : O$,

1. Compute optimal output weights $\mathbf{w} = \left(G^\top G + 2\frac{\lambda}{l}\mathbf{I}\right)^{-1} G^\top \mathbf{y}$

2a. Compute gradients $\frac{\partial}{\partial \mu_k} E$ and $\frac{\partial}{\partial \sigma_k} E$ as in (5) and (4) with optimal $\mathbf{w}$ and form a gradient vector $\mathbf{v}$

2b. Estimate the conjugate direction $\overline{\mathbf{v}}$ with Fletcher-Reeves-Polak-Ribiere CG-Method [5]

3a. Perform a line search to find the minimizing step size δ in direction $\overline{\mathbf{v}}$; in each evaluation of E recompute the optimal output weights $\mathbf{w}$ as in line 1

3b. Update μ_k and σ_k with $\overline{\mathbf{v}}$ and δ

Output: Optimized RBF net

Figure 2: Pseudo-code description of the RBF net algorithm

A.3 Notes

A.3.1 Licensing Terms

A.3.2 No warranty

Algorithm AdaBoost$(\mathbf{Z}, T)$

Input: l examples $\mathbf{Z} = \langle(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_l, y_l)\rangle$

Initialize: $w_1(\mathbf{z}_i) = 1/l$ for all $i = 1 \ldots l$

Do for $t = 1, \ldots, T$,

1. Train classifier with respect to the weighted sample set $\{\mathbf{Z}, \mathbf{w}^t\}$ and obtain hypothesis $h_t : \mathbf{x} \mapsto \{\pm 1\}$
2. Calculate the training error ϵ_t of h_t:

$$\epsilon_t = \sum_{i=1}^{l} w^t(\mathbf{z}_i)\mathrm{I}(h_t(\mathbf{x}_i) \neq y_i) \ , \tag{7}$$

   abort if $\epsilon_t = 0$ or $\epsilon_t \geq \frac{1}{2} - \Delta$, where Δ is a small constant

3. Set

$$b_t = \log \frac{1 - \epsilon_t}{\epsilon_t}. \tag{8}$$

4. Update weights $\mathbf{w}^t$:

$$w^{t+1}(\mathbf{z}_i) = w^t(\mathbf{z}_i) \exp\{-b_t \mathrm{I}(h_t(\mathbf{x}_i) = y_i)\} / Z_t \ , \tag{9}$$

   where Z_t is a normalization constant, such that $\sum_{i=1}^{l} w_{t+1}(\mathbf{z}_i) = 1$.

Output: Final hypothesis

$$f(\mathbf{x}) = \sum_{t=1}^{T} c_t h_t(\mathbf{x}), \quad \text{where} \quad c_t := \frac{b_t}{\sum_{t=1}^{T} |b_t|} \tag{10}$$

Figure 3: The AdaBoost algorithm [2].

References

[1] C.M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[2] Y. Freund and R.E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *EuroCOLT: European Conference*

Algorithm AdaBoost$_{Reg}(\mathbf{Z}, T, C, p)$

Input: l examples $\mathbf{Z} = \langle(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_l, y_l)\rangle$

Initialize: $w_1(\mathbf{z}_i) = 1/l$ for all $i = 1 \ldots l$

Do for $t = 1, \ldots, T$,

1. Train classifier with respect to the weighted sample set $\{\mathbf{Z}, \mathbf{w}_t\}$ and obtain hypothesis $h_t : \mathbf{x} \mapsto [-1 \ldots 1]$
2. Find the weight of the hypothesis b_t:

$$b_t = \operatorname*{argmin}_{b_t \geq 0} \sum_{i=1}^{l} \exp\left\{-\frac{1}{2}\left[\rho(\mathbf{z}_i, \mathbf{b}^t) + C|\mathbf{b}^t|\eta_t(\mathbf{z}_i)\right]\right\}, \qquad (11)$$

where $\eta_t(\mathbf{z}_i) \equiv \left(\sum_{r=1}^{t} c_r w_r(\mathbf{z}_i)\right)^2$ and $\rho(\mathbf{z}_i, \mathbf{b}^t) \equiv y_i \sum_{r=1}^{t} c_r h_r(\mathbf{x}_i)$
abort if

$$b_t = 0 \quad \text{or} \quad b_t \geq \Gamma\ , \qquad (12)$$

where Γ is a large constant

3. Update weights $\mathbf{w}^t$:

$$\mathbf{w}^{t+1}(\mathbf{z}_i) = \frac{1}{Z_t} \exp\left\{-\frac{1}{2}\left[\rho(\mathbf{z}_i, \mathbf{b}^t) + C|\mathbf{b}^t|\eta_t(\mathbf{z}_i)\right]\right\} \qquad (13)$$

where Z_t is a normalization constant, such that $\sum_{i=1}^{l} w_{t+1}(\mathbf{z}_i) = 1$.

Output: Final hypothesis with weights $\mathbf{b}$ as in Figure 3.

Figure 4: The AdaBoost$_{Reg}$ (ABR) algorithm [6, 7], where C is the regularization constant. For $C = 0$ and $h_t \in \{-1, +1\}$ this algorithm is equivalent to AdaBoost.

on Computational Learning Theory. LNCS, 1994.

[3] J. Moody and C. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1(2):281–294, 1989.

[4] K.-R. Müller, A. Smola, G. Rätsch, B. Schölkopf, J. Kohlmorgen, and V. Vapnik. Using support vector machines for time series prediction. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods — Support Vector Learning*. MIT Press, Cambridge, MA, 1998.

[5] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, second edition, 1992.

[6] G. Rätsch. Ensemble learning methods for classification. Master's thesis, Dep. of Computer Science, University of Potsdam, April 1998. In German.

[7] G. Rätsch, T. Onoda, and K.-R. Müller. Soft margins for AdaBoost. *Machine Learning*, 42(3):287–320, March 2001. also NeuroCOLT Technical Report NC-TR-1998-021.