Technical Report

Advanced Multimedia Research Laboratory
Information Communications Technology Research Institute
University of Wollongong
Wollongong 2525
Australia
Phone: +61 (2) 4221 5321
Fax: +61 (2) 4221 4170
E-mail: alc456@uow.edu.au
URL: http://www.uow.edu.au/

AMRL-TR-2008-PO05 23rd January 2009

AdaBoost Toolbox: A MATLAB Toolbox for Adaptive Boosting

Alister Cordiner, MCompSc Candidate
School of Computer Science and Software Engineering
University of Wollongong

Abstract

AdaBoost is a meta-learning algorithm for training and combining ensembles of base learners. This technical report describes the AdaBoost Toolbox, a MATLAB library for designing and testing AdaBoost-based classifiers and regressors. The toolbox implements some existing boosting algorithms and provides the flexibility to implement others. Functions are also provided for assessing the performance of trained classifiers and regressors.

Contents

1 Introduction

1.1 Symbols used

M	number of weak learner/rounds of boosting
N	number of training samples
S	number of features per training sample
x_n	Training sample n
y_n	Training class label n
F	strong learner
f_m	weak learner m
G_m	set of candidate weak learners at round m
b	strong learner bias
α_m	weak learner m weight
w_n	sample weight n
C_ℓ	class ℓ
L	number of classes

1.2 What is AdaBoost?

Adaptive Boosting (AdaBoost) is a popular method of increasing the accuracy of any supervised learning technique through resampling and arcing (adaptive reweighting and combining). AdaBoost itself is not a learning algorithm, but rather a meta-learning technique that "boosts" the performance of other learning algorithms, known as *base learners* or *weak learners*, by weighting and combining them. The basic premise is that multiple weak learners can be combined to generate a more accurate ensemble, known as a *strong learner*, even if the weak learners perform little better than random. From a high level, the AdaBoost ensemble can be viewed as a form of neural network as shown in Figure 1. Similar to the concept of bagging (bootstrap aggregation), subsets of the training set are used to train the individual weak learners which are combined using a voting scheme, however in AdaBoost the subsets are selected based on the performance of the previous weak learners rather than being selected randomly.

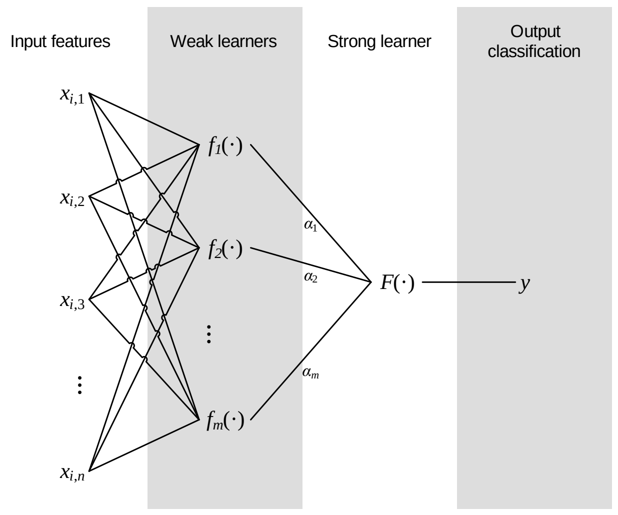


Figure 1: AdaBoost can be viewed as a neural network. The weak learners and strong learner are hidden layers and the output classification is the output layer. The activation functions are determined by the weak learners and the variant of AdaBoost employed.

First pioneered by Freund & Schapire [8], AdaBoost has generated much interest in the machine learning community in recent years, and has seen use in fields including image processing [16], data mining [15], bioinformatics [12] and quantitative analysis [3].

Boosting addresses two problems: choosing the training samples, and combining multiple weak learners into a single strong learner. During training, incorrectly classified training samples are weighted higher to redirect the focus of the training upon them in subsequent weak classifiers. Combining multiple weak learners is extremely effective, because if each weak learner is slightly better than random and they each make different errors, then by combining them the training error will drop exponentially [13]. Resampling of the training samples ensures diversity in the weak learners so that they make errors on different parts of the data.

A large number of extensions and modifications to the original AdaBoost algorithm have been proposed. Adaptions are available for both classification and regression problems. For classification, they can address either binary or multi-class problems. Many different objective functions have also been proposed for the learning.

It was claimed by [5] that AdaBoost using decision trees as the weak learners is the best "off-the-shelf" classifier available. Some of the avantages of AdaBoost are that it is simple, can be efficiently implemented, has few tuning parameters and has certain theoretical performance "guarantees". However, the No Free Lunch theorem states that no classification algorithm is universally superior to others [13]. The main disadvantages of AdaBoost are that it can overfit, is susceptible to noise (although some variants partially overcome this) and it generally requires large amounts of training data.

1.3 What does this toolbox do?

The Boost Toolbox is a pure MATLAB object-oriented library that implements several AdaBoost-derived algorithms. Although other MATLAB implementations of AdaBoost do exist, this toolbox has been designed with two goals in mind: ease of use and flexibility.

It has been designed to be flexible by offering a generic AdaBoost class that is easily extendable. Multiple aspects of the algorithm can be customised, such as the features, weak learners, sample weight initialisation and update rules, objective function, transfer functions and weak classifier weighting scheme. Although some variants of AdaBoost are already provided in the toolbox, others can be easily added without having to modify any of the library's source code.

The Boost Toolbox has also been implemented to mimic the MATLAB Neural Network Toolbox. Users familiar with this toolbox will find the syntax familiar. A simple code example is shown in Listing 1.

Listing 1: Simple example of using the Boost Toolbox

```
% instantiate a GentleBoost object
bst = newgab;

% train the classifier for 10 rounds
bst = train(bst,x,y,10);

% simulate the trained classifier
ysim = sim(bst,x);
```

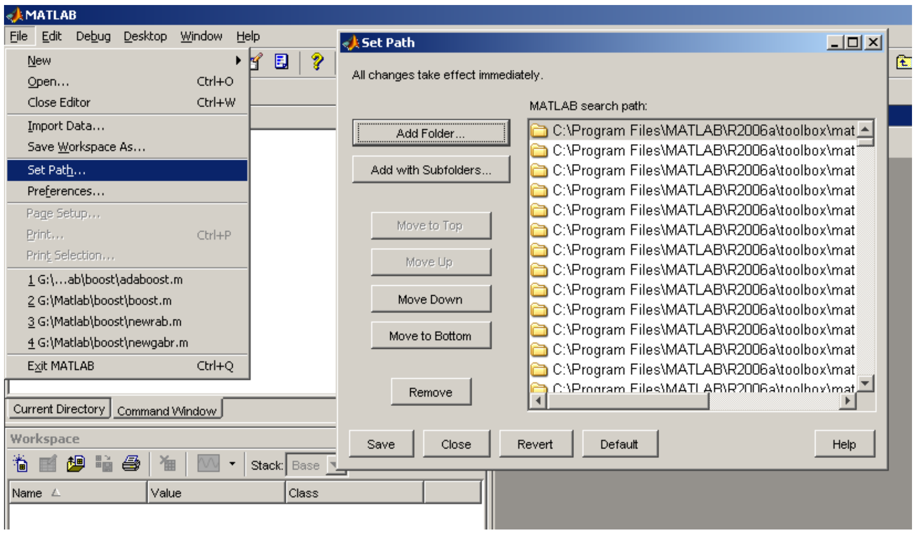
If it is installed, many of the functions provided in the Neural Network Toolbox can be used together with this toolbox, such as to analyse the correlation between the classifier output and the ground truth. If available, functionality from the Bioinformatics Toolbox can also be used together with this toolbox. See Section 5 for information on using the Boost Toolbox together with other toolboxes. Note that no other toolboxes need be installed to use the Boost Toolbox.

Note that this library has not been designed for efficiency. Instead it is intended to allow researchers to quickly prototype and test AdaBoost-based machine learning algorithms. There exist highly optimised libraries such as the Intel OpenCV C++ library [1] which should be used instead if efficiency is a primary requirement.

1.4 Terms of use

There are no restrictions on the use of this toolbox but please cite this report if you do use it in your research:

```
@TECHREPORT{AMRL-TR-2008-P005,
  author = {Alister Cordiner},
  title = {AdaBoost Toolbox: A {MATLAB} Toolbox for Adaptive Boosting},
  institution = {University of Wollongong},
  year = {2008},
  number = {AMRL-TR-2008-P005}
}
```



All changes take effect immediately.

```
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
C:\Program Files\MATLAB\R2006a\toolbox\mat
```

Figure 2: Adding the toolbox to MATLAB's path

2 Using the toolbox

2.1 Installation

The steps to install the AdaBoost Toolbox are:

1. Download the zip file and unzip to somewhere on your computer, taking note of the full directory name you unzip to

2. Add this directory to your MATLAB path by clicking *File* → *Set Path...* → *Add Folder...* and selecting the directory (see Figure 2)

3. Click *Save* and then *Close*

To upgrade an existing installation of this toolbox, simply copy the new files over the old ones.

2.2 Getting started

All of the AdaBoost functionality is provided via the `boost` object. Different variants of AdaBoost inherit from this object. This object is used in two steps: first, the object is trained using the training data, and second, the object is simulated using testing data.

Training data is stored in the variables `x` and `y`. The variable `x` represents $\mathbf{X}^{S\times N}$ where each row vector $\mathbf{x}_n$ is an input sample. The variable `y` represents $\mathbf{Y}^{L\times N}$ where each row vector $\mathbf{y}_n$ is an output or target value. In the case of a binary classifier, this is a scalar output value y_n for each training sample $\mathbf{x}_n$. During training, the AdaBoost object attempts to learn the relationship between the input samples $\mathbf{X}$ and target values $\mathbf{Y}$ either through classification or regression. Simple examples of classification and regression are shown in Listings 2 and 3 using some sample training data provided with the toolbox.

Listing 2: Simple example of binary classification

```
% load some toy training data
load databclass2d x y

% instantiate a GentleBoost object
bst = newgab;

% train the classifier for 50 rounds
bst = train(bst,x,y,50);

% simulate the trained classifier
ysim = sim(bst,x);

% find the training error rate of the classifier
er = perfer(ysim,y);
fprintf('Error rate = %f\n', er);
```

Listing 3: Simple example of regression

```
% load some toy training data
load datareg3d x y

% instantiate a GentleBoost object
bst = newgabr;

% train the classifier for 50 rounds
% (use zero-mean shifted data)
bst.b = mean(y);
bst = train(bst,x,y-bst.b,50);

% simulate the trained classifier
ysim = sim(bst,x);

% find the training error rate of the classifier
er = perfmse(ysim,y);
fprintf('Error rate = %f\n', er);
```

Displaying the properties reveals useful information about the current `boost` object, as shown in Figure 3. These properties can be modified to implement different extensions of the AdaBoost algorithm.

2.3 Training

The first step to using the `boost` object is training. This is done by calling the `train` method of the object. All AdaBoost variants follow the same sequence of steps shown in Algorithm 1.

2.4 Simulating

Once the `boost` object has been trained, it can be simulated by applying it to testing data. This is done by calling the `sim` method of the object. Again, all AdaBoost variants

```
>> bst

    AdaBoost object:

   properties:

         quietMode: false
renormaliseWeights: true
 singleFeatLearner: true
      resampleSize: 0.000000

    functions:

     initWeightFcn: initeqc
         outputFcn: outcombth
          inputFcn: hardlims
    featExtractFcn: @(x) x
   weightUpdateFcn: wupexp
          alphaFcn: @(y,ysim,bst) 1
        weakSimFcn: stumpfunc
      weakTrainFcn: stumplearn
           perfFcn: perfer
          adaptFcn: adapt

    variables:

                 w: [] (sample weights)
                 b: 0.000000 (output bias)
             alpha: [] (weak learner weights)
            stages: 0
          userdata: []
```

Figure 3: Example object properties displayed in the MATLAB console

Algorithm 1 Pseudocode for training. * Optional steps.

```
1. Initialise $\mathbf{w}$ (initWeightFcn)
2. For $m = 1, \ldots, M$
    (a) Extract features $\mathbf{x}_m$ (featExtractFcn)
    (b) Input transfer function applied to $\mathbf{x}_m$ (inputFcn)
    (c) Train weak learner $f_m$ on $\mathbf{y}_m$ and $\mathbf{w}$ (weakLearnFcn)
    (d) Call alpha update function (alphaFcn) *
    (e) Renormalise weights $\mathbf{w}$ *
    (f) Calculate error rate and performance (perfFcn) *
```

Algorithm 2 Pseudocode for simulation.

```
1. For $m = 1, \ldots, M$
    (a) Extract features $\mathbf{x}_m$ (featExtractFcn)
    (b) Input transfer function applied to $\mathbf{x}_m$ (inputFcn)
    (c) Calculate weak learner $f_m$ output $\mathbf{y}_m$ (weakSimFcn)
    (d) Weight the weak learner by its $\alpha_m$ value
2. Calculate the output based on the combination rule (outputFcn)
```

follow the same sequence of steps shown in Algorithm 2.

2.5 Initialisation

The different extensions to AdaBoost are implemented as `boost` objects with customised properties. Some extensions have already been implemented in the toolbox, and the objects are instantiated with the following initialisers:

- `newdab` – Discrete AdaBoost
- `newrab` – Real AdaBoost
- `newgab` – Gentle AdaBoost (classification)
- `newgabr` – Gentle AdaBoost (regression)

Once initialised, the objects can be further customised if required, such as by modifying the weak learners. The properties that can be customised are described in the following sections.

2.6 Properties

2.6.1 List of properties

Property name	Type
`renormaliseWeights` (see section 2.6.2)	`logical`
`resampleSize` (see section 2.6.3)	`double` (values from 0 to 1)
`singleFeatLearner` (see section 2.6.4)	`logical`
`quietMode` (see section 2.6.5)	`logical`
`userdata` (see section 2.6.6)	any type

2.6.2 Distribution normalisation (`renormaliseWeights`)

For many AdaBoost algorithms, the sample weights are interpreted as being a probability distribution function. If this is the case, the weights should be re-normalised to sum to unity after each round:

$$w_n \rightarrow \frac{w_n}{\sum_{i=1}^{N} w_i} \text{ for } n = 1, \ldots, N$$

If `renormaliseWeights` is set to true, this step will be performed after each round of boosting. In the case that they are not a PDF (e.g. in regression problems, the weights may represent the target residuals),`renormaliseWeights` should be set to false.

2.6.3 Resampling subset size (`resampleSize`)

If resampling is used, $N\times$`resampleSize` training samples are drawn based on their weightings to be used for each training round. It is expressed as a ratio and thus `resampleSize` $\in (0, 1)$. This can speed up training and reduce memory requirements (which can help overcome the limitation that MATLAB numeric matrices require continguous blocks of memory [2]) since less samples are used for each training round, however lower values of `resampleSize` will generally result in a less accurate classifer. Setting `resampleSize` to zero will disable resampling and use the entire training set for each round of training.

2.6.4 Single feature weak learners (`singleFeatLearner`)

In some weak learner schemes, a single weak learner corresponds to a single feature, such as stump classifiers. In others, such as decision trees or neural networks, a single weak learner may use many or even all of the available features. The `singleFeatLearner` property is used to specify whether the weak learner uses a single feature or all features. See section 2.7.2 for details on specifying different types of weak learners.

2.6.5 Quiet mode (`quietMode`)

When the AdaBoost object is trained, a dialog window will display the current progress of the training. To disable the window, set the `quietMode` property to false before training.

2.6.6 User data (`userdata`)

User data can be used to store any additional data required.

2.7 Functions

2.7.1 List of functions

Function	Parameters	Examples
`weakSimFcn` (see section 2.7.2)	`w = fcn(y,ysim,bst)`	`stumpfunc` `treefunc` `regressfunc`
`weakTrainFcn` (see section 2.7.2)	If `singleFeatLearner` is false: `[fm,feats] = fcn(x,y,w)` If `singleFeatLearner` is true: `fm = fcn(x,y,w)`	`stumplearn` `treelearn` `regresslearn`
`initFcn` (see section 2.7.3)	`w = fcn(y)`	`initeq` `initeqc` `inittarg`
`weightUpdateFcn` (see section 2.7.4)	`w = fcn(y,ysim,bst)`	`wupexp` `wupsub`
`featExtractFcn` (see section 2.7.5)	`x = fcn(x)`	
`alphaFcn` (see section 2.7.6)	`x = fcn(y,ysim,bst)`	`wgtlogit`
`perfFcn` (see section 2.7.7)	`p = fcn(y,ysim)`	`perfer` `perffpr` `perffnr` `perfmse`
`inputFcn` (see section 2.7.8)	`y = fcn(y)`	`hardlim` `hardlims` `purelin`
`outputFcn` (see section 2.7.9)	`y = fcn(y)`	

2.7.2 Weak learners (`weakSimFcn` and `weakTrainFcn`)

The weak learners are specified by two functions: the training function `weaktrainFcn` and the simulation function `weakSimFcn`. The training function accepts the training samples and weights and returns a weak learner object `fm` and the features used by this object `feats`. Some training functions always use only a single feature whereas others may use many or even all of the available features. See section 2.6.4 for details on specifying whether a weak learner uses a single feature or multiple features. The simulation function only receives the subset of features `feats` from the input samples returned during training as its input and should return the classification results (which can be real-valued if the AdaBoost variant allows real-valued weak learner outputs).

Three common forms of weak learners are provided by the toolbox: simple threshold classifiers (`threshfunc` and `threshlearn`) and decision stumps (`stumpfunc` and `stumplearn`) for classification problems, and linear regression learners (`regressfunc` and `regresslearn`) for regression problems. An example of using decision stumps is shown in Listing 4.

Listing 4: Using a decision tree weak learner

```
% instantiate a GentleBoost object
bst = newgab;

% use a decision stump weak learner
bst.weakSimFcn = @stumpfunc;
bst.weakTrainFcn = @stumplearn;
bst.singleFeatLearner = true;

% train the classifier for 50 rounds
bst = train(bst,x,y,50);
```

Simple threshold classifier The weak learner output is:

$$f_m(x) = \text{sign}\left(x - \phi\right)$$

where ϕ is a threshold selected which minimises the weighted error rate:

$$\epsilon(f_m) = Pr_{n \sim w_m}\left[f_m(x_n|\theta) \neq y_n\right]$$

This is a discrete binary classifier that uses single features. The learner function is `threshlearn` and the simulation function is `threshfunc`.

Decision stumps The weak learner output is:

$$f_m(x) = m \times \text{sign}\left(x - \phi\right) + b$$

where m, b and θ are selected to minimise the weighted squared error:

$$\epsilon(f_m) = \sum_{n=1}^{N} w_n(y_n - f_m(x_n|m, b, \theta))^2$$

This is a real-valued binary classifier that uses single features. The learner function is `stumplearn` and the simulation function is `stumpfunc`.

Linear regression learners The weak learner output is:

$$f_m(x) = mx + b$$

where m and b are selected to minimise the weighted squared error:

$$\epsilon(f_m) = \sum_{n=1}^{N} w_n (y_n - f_m(x_n|m, b))^2$$

This is a regressor that uses single or multiple features. The learner function is `regresslearn` and the simulation function is `regressfunc`.

2.7.3 Sample weight initialisation (`initFcn`)

Sample weight initialisation is controlled by assigning a function handle to the `initFcn` property. Some common initialisation schemes are provided by the toolbox.

Two sample weight initialisation schemes are commonly used for classification problems. One is to weight all samples equally (implemented by `initeq`), such that:

$$\sum_{n=1}^{N} w_n = 1$$

The other approach is to weight them based on the number of samples in each class, so that the sample weights for each class sum to $1/L$ (`initeqc`):

$$\sum_{n=1}^{N} w_n = \frac{1}{L} \text{ where } y_n \in C_\ell \text{ for } \ell = 1, \ldots, L$$

This will give each class equal weighting rather than each sample to prevent the training being biased towards classes with larger numbers of training samples.

For regression problems, the weights can represent the target residual values. At initialisation, they can simply be set to the target values (`inittarg`):

$$w_n = y_n$$

Other custom weight initialisation rules can be created by creating a function of the form `w = func(y)` where w is the initialised weights and y is the training target values.

2.7.4 Sample weight update rule (`weightUpdateFcn`)

The sample weights are modified at each round to shift the focus of the training to different training samples and so encourage the generation of diverse ensembles. For AdaBoost classification variants, a common update rule is to exponentially grow or decay the weight coefficients at each round (`wupexp`):

$$w_i \leftarrow w_i \exp\left[-\alpha_m y_i f_m(x_i)\right]$$

For regression variants where $\mathbf{w}$ represents the update target values, the update rule is usually given by a subtractive rule (`wupsub`):

$$w_i \leftarrow w_i - \alpha_m f_m(x_i)$$

2.7.5 Feature extraction (**featExtractFcn**)

The input data `x` can be of practically any data type (matrix of doubles, cell array, etc). In most cases, it will be a `double` matrix containing the features that are used for training and simulation. However, the feature extraction function can be used so that `x` can be a representation of the input samples and then the actual feature values that are used are only calculated during the training or simulation. The feature extraction function is expected to return a matrix where each row represents a sample (correct?) and each column represents a feature. For each sample, the function must generate the same number of features S. Some examples could include:

- `x` contains image filenames and `featExtractFcn` is a function to read in an image to a vector.
- `x` contains primary keys and `featExtractFcn` is a function to read a row of data from a database into a vector based on the key.
- `x` contains vectors and `featExtractFcn` is a function to project each vector into a lower-dimensional subspace.

Used in conjunction with `resampleSize`, the feature extraction function is useful if each input sample contains a large number of features, since they will be calculated only as they are required rather than pre-calculating them all before training and then loading them into memory. This is significantly slower since features are potentially recalculated numerous times, but this approach has lower memory requirements. Listing 5 shows an example of a feature extraction function where `x` is a cell array of image filenames and `featExtractFcn` is set to a function that reads in and vectorises the images.

Listing 5: Using the feature extraction function to read in images using functions from the Image Processing Toolbox

```
% cell array containing list of image filenames
imgs = dir('*.jpg');

% instantiate a GentleBoost object
bst = newgab;

% function to read in and vectorise images (the features
% will be the pixel values)
bst.featExtractFcn = @(x) reshape(im2double(imread(x)),1,[]);

% train the classifier for 50 rounds
bst = train(bst,{imgs.name},y,50);
```

In situations where feature extraction is not required, the features can simply be set to the input samples using an anonymous function such as `featExtractFcn = @(x) x`.

2.7.6 Weak learner weights (`alphaFcn`)

The weak learners can be assigned weights. For example, half logit function (`wgtlogit`) is given by:

$$\alpha_m = \frac{1}{2} \ln \left(\frac{1 - \epsilon_m}{\epsilon} \right) \text{ where } \epsilon_m = \sum_{i=1}^{N} w_i (y_i - f(x_i))^2$$

The half log-likelihood function (`wgthalflike`) is given by:

$$\alpha_m = \frac{1}{2} \ln \left(\frac{\sum_{i|y_i=1} w_i f_m(x_i)}{\sum_{i|y_i=-1} w_i f_m(x_i)} \right)$$

Many other variants simply use a constant value.

2.7.7 Performance function (`perfFcn`)

The performance function is the function which is used to determine the performance of an AdaBoost classifier. These are described in Section 4.1.

2.7.8 Input transfer function (`inputFcn`)

The input transfer functon will normally simply be a mapping function to ensure that the training class labels take on the correct values. In binary classification problems, most AdaBoost algorithms expect that the class labels are either 0 and 1, or -1 and $+1$. For example, an input transfer function may be (`hardlims`):

$$f : \chi \rightarrow \{-1, +1\}$$

Or (`hardlim`):

$$f : \chi \rightarrow \{0, 1\}$$

To perform no mapping (e.g. for regression problems), use the `purelin` transfer function.

2.7.9 Output combination rule (`outputFcn`)

For AdaBoost regression algorithms, the output is generally simply the weighted linear combination of the weak learners (`outcomb`):

$$F(x) = \left(\sum_{m=1}^{M} \alpha_m f_m(x) \right) + b$$

For most AdaBoost classification algorithms, the output classification is found by thresholding a linear combination of the weak classifiers using a threshold b (`outcombth`):

$$F(x) = \begin{cases} \sum_{m=1}^{M} \alpha_m f_m(x) \geq b & +1 \\ \sum_{m=1}^{M} \alpha_m f_m(x) < b & -1 \end{cases}$$

Algorithm 3 Discrete AdaBoost (adapted from [9])

```
Given: $(x_1, y_1), \ldots, (x_N, y_N)$ where $x_i \in X$, $y_i \in Y = \{-1, +1\}$

  1. Initialise $w_i = 1/N$

  2. For $m = 1, \ldots, M$

       (a) Train weak learner using distribution $w$

       (b) Generate weak learner $f_m : \chi \rightarrow \{-1, +1\}$ with minimum weighted error:

             $f_m(x) = \arg\min_{f \in F} \{\epsilon(f) = Pr_{i \sim w_m}[f_m(x_i) \neq y_i]\}$

       (a) Choose $\alpha_m = \frac{1}{2} \ln\left(\frac{1-\epsilon_m}{\epsilon}\right)$

       (b) Set $w_i \leftarrow w_i \exp[-\alpha_m y_i \cdot f_m(x_i)]$ for $i = 1, 2, \ldots, N$

       (c) Renormalise such that $\sum_{i=1}^{N} w_i = 1$

  3. Output the final classification:

       $F(x) = \begin{cases} \sum_{m=1}^{M} \alpha_m f_m(x) \geq 0 & +1 \\ \sum_{m=1}^{M} \alpha_m f_m(x) < 0 & -1 \end{cases}$
```

2.8 Help documentation

The help documentation for this toolbox can be accessed through MATLAB using the `doc` or `help` commands. For example, to access the help documentation for the `newgab` function, in the help browser type `doc initgab`, or in the command window type `help initgab`.

3 AdaBoost variants

3.1 Discrete AdaBoost

Discrete AdaBoost was the original AdaBoost algorithm proposed by [8]. It generates an ensemble of weighted binary classifiers that take a discrete value $f_m(x) \in \{-1, +1\}$. Training samples are weighted to indicate their probability of being included in the next training subset, with incorrectly classified samples being weighted higher.

Algorithm 4 RealBoost (adapted from [11])

```
1. Start with weights $w_i = 1/2a$ for the $a$ examples where $y_i = +1$ and $w_i = 1/2b$ for
   the $b$ examples where $y_i = -1$
2. Repeat for $m = 1, 2, \ldots, M$
    (a) Train weak learner using distribution $w$
    (b) Generate weak learner $f_m : \chi \to \{-1, +1\}$ which minimises the half log
        likelihood:
        $$f_m(x) = \frac{1}{2} \log \frac{P\left(y = +1 | x, w^{(M-1)}\right)}{P\left(y = -1 | x, w^{(M-1)}\right)} = \frac{1}{2} \log \left( \frac{\sum_{i|y_i=1} w_i f_m(x_i)}{\sum_{i|y_i=-1} w_i f_m(x_i)} \right)$$
    (c) Set $w_i \leftarrow w_i \exp\left[-y_i \cdot f_m(x_i)\right]$ for $i = 1, 2, \ldots, N$
    (d) Renormalise such that $\sum_i w_i = 1$
3. Output the final classification:
    $$F(x) = \begin{cases} \sum_{m=1}^{M} f_m(x) \geq 0 & +1 \\ \sum_{m=1}^{M} f_m(x) < 0 & -1 \end{cases}$$
```

3.2 Real AdaBoost

Real AdaBoost [14] is a variant of AdaBoost for binary classification where the weak learners are allowed to output a real value $f_m(x) \in \mathbb{R}$. The sign of this output gives the predicted label $\{-1, +1\}$ and its magnitude gives a measure of confidence in this prediction.

3.3 Gentle AdaBoost (classification)

Similar to Real AdaBoost, Gentle AdaBoost [10] permits the weak classifiers to output confidence-weighted real values for binary classification. However, it uses the half log-ratio to update the weighted class probabilities.

3.4 Gentle AdaBoost (regression)

Gentle AdaBoost can also be adapted to apply to regression problems. The main difference is that, rather than updating the sample weights between rounds to focus the training, the target values themselves are updated [6]. This technique is from a broader class of AdaBoost-based *relabelling algorithms* which approach regression problems by iteratively subtracting from the target values during to minimise the residual [4].

Note: In the implementation, w is used instead of y^{λ}.

Algorithm 5 Gentle AdaBoost (adapted from [7])

```
1. Start with weights $w_i = 1/N$, $i = 1, \ldots, N$, $F(x) = 0$
2. Repeat for $m = 1, 2, \ldots, M$
    (a) Generate weak learner $f_m : \chi \to \{-1, +1\}$ which minimises the weighted squared error:
        $$f_m = \arg\min_f \left( \epsilon_m = \sum_{i=1}^{N} w_i (y_i - f(x_i))^2 \right)$$
    (b) Set $w_i \leftarrow w_i \exp\left[-y_i \cdot f_m(x_i)\right]$ for $i = 1, 2, \ldots, N$
    (c) Renormalise such that $\sum_{i=1}^{N} w_i = 1$
3. Output the final classification:
    $$F(x) = \begin{cases} \sum_{m=1}^{M} f_m(x) \geq 0 & +1 \\ \sum_{m=1}^{M} f_m(x) < 0 & -1 \end{cases}$$
```

Algorithm 6 Gentle AdaBoost (adapted from [6])

```
1. Start with regression target values $y^\lambda = y$, $i = 1, \ldots, N$
2. Repeat for $m = 1, 2, \ldots, M$
    (a) Generate weak learner $f_m : \chi \to \{-1, +1\}$ which minimises the squared error:
        $$f_m = \arg\min_f \left( \epsilon_m = \sum_{i=1}^{N} (y_i^\lambda - f(x_i))^2 \right)$$
    (b) Set $y_i^\lambda \leftarrow y_i^\lambda - f_m(x_i)$, $i = 1, 2, \ldots, N$
3. Output the final result:
    $$F(x) = \sum_{m=1}^{M} f_m(x)$$
```

4 Classification performance

4.1 Performance functions

Performance functions compare the simulated output of the AdaBoost classifier or regressor to the ideal target values. The toolbox provides a number of performance functions to evaluate the accuracy of a classifier or regressor:

- `perfer` – error rate of a classifier
- `perffpr` – false positive rate (FPR) of a classifier, inverse of the true negative rate (TNR)
- `perffnr` – false negative rate (FNR) of a classifier, inverse of the true positive rate (TPR)
- `perfmse` – mean-squared error (MSE) of a regressor or classifier

4.2 Receiver operating characteristic curve

The receiver operating characteristic (ROC) curve plots the TPR versus the FPR of a classifier. This applies only to binary classification versions of AdaBoost using a linear combination output function. The output classification is computed as:

$$F(x) = \begin{cases} \sum_{m=1}^{M} \alpha_m f_m(x) \geq b & +1 \\ \sum_{m=1}^{M} \alpha_m f_m(x) < b & -1 \end{cases}$$

The ROC curve can be calculated by varying the value of the threshold b to obtain different values for the TPR and FPR. Generation of ROC curves is provided by the `roc` function, which generates a list of corresponding FPR and TPR values for the classifier. It can also optionally return the list of thresholds used to obtain each pair of FPR and TPR values. An example is shown in Listing 6.

Listing 6: Plotting an ROC curve

```
% load some toy training data
load databclass2d x y

% instantiate a GentleBoost object
bst = newgab;

% train the classifier for 50 rounds
bst = train(bst,x,y,50);

% simulate the trained classifier
ysim = sim(bst,x);

% plot the ROC curve
[tpr,fpr] = roc(bst,x,y);
plot(fpr,tpr,'LineWidth',2)
```

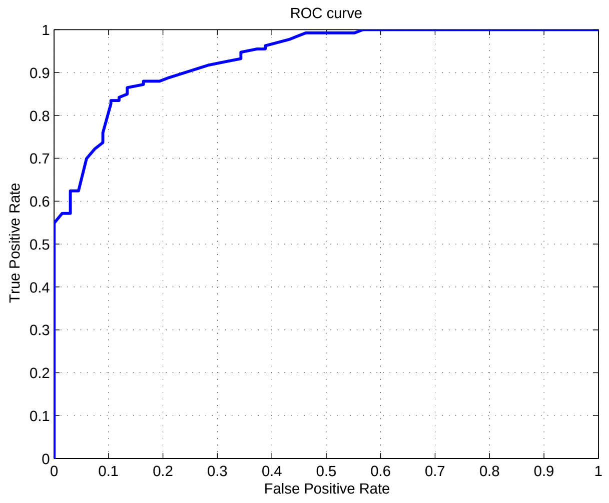


Figure 4: ROC curve generated by Listing 6

```
grid on
title('ROC curve')
xlabel('False Positive Rate')
ylabel('True Positive Rate')
```

4.3 Summary statistics

Because adjusting the threshold can simultaneously alter the true positive rate, false positive rate and error rate, providing any of these statistics alone as the measure of a classifier's accuracy insufficient. Providing all of them is more useful, but there is no easy way to compare classifiers measured at different points on their respective ROC curves. Two summary statistics are available to allow fair and simple comparison of the performance of different classifiers with a single statistic: the Area Under Curve (AUC) and Equal Error Rate (EER).

The AUC refers to the integral of the ROC curve and provides a convenient summary statistic. After calculating the TPR and FPR values using the `roc` function, the AUC can be calculated using the `roc_auc` function.

The EER, sometimes also referred to as the Crossover Error Rate (CER), is the error rate at the point on the ROC at which the proportion of false positives is equal to the proportion of false negatives. The EER can be calculated using the function `roc_eer`. This function can also optionally return the threshold that was used to obtain the EER, as shown in Listing 7.

Listing 7: Calculating the EER and AUC

```
% load some toy training data
load databclass2d x y

% instantiate a GentleBoost object
bst = newgab;
```

```

% train the classifier for 50 rounds
bst = train(bst,x,y,50);

% simulate the trained classifier
ysim = sim(bst,x);

% find the classifier performance with b=0
fprintf('b = %f:\n', bst.b);
fprintf('\tFNR = %f, FPR = %f, ER = %f\n', ...
    perffnr(ysim,y), perffpr(ysim,y), perfer(ysim,y));

% find the value of b that yields the EER
[tpr,fpr,th] = roc(bst,x,y);
[eer,b] = roc_eer(tpr,fpr,th);

% find the AUC
auc = roc_auc(tpr,fpr);

% find the new classifier performance
bst.b = b;
ysim = sim(bst,x);
fprintf('b = %f:\n', bst.b);
fprintf('\tFNR = %f, FPR = %f, ER = %f\n', ...
    perffnr(ysim,y), perffpr(ysim,y), perfer(ysim,y));
fprintf('EER = %f, AUC = %f\n', eer, auc);
```

5 Using functionality from other toolboxes

5.1 Weak learners

The toolboxes available for MATLAB allow the possibility of using a large variety of weak learners for the AdaBoost algorithm. Some possible classifiers include support vector machines (`svmtrain` and `svmclassify` from the Bioinformatics Toolbox) and decision trees (`treefit` and `treeval` from the Statistics Toolbox). Some possible regressors include stepwise regression (`stepwisefit` from the Statistics Toolbox) and robust linear regression (`robustfit` from the Statistics Toolbox).

Listing 8: Neural network weak learner

```
% Weak learner training function
function [fm,feats] = nnet_train(x,y,w)

    % create a NN object and train it
    net = newff(minmax(x),[5 1]);
    fm = train(net,x,y);

    % this weak classifier uses all of the features
    feats = 1:size(x,1);

    close gcf %close the training window

end

% Weak learner simulaton function
function ysim = nnet_sim(fm,x)

    ysim = hardlims(sim(fm,x));

end
```

Listing 9: Using weak learners from the Neural Network Toolbox

```
% load some toy training data
load databclass2d x y

net = nnet_train(x,y,[]);
ysim_nobst = nnet_sim(net,x);

% instantiate a GentleBoost object
bst = newgab;

% use half the training samples for each round
% of training
bst.resampleSize = .5;

% specify the NN as the weak learner
bst.weakTrainFcn = @nnet_train;
bst.weakSimFcn = @nnet_sim;

% train the classifier for 10 rounds
bst = train(bst,x,y,10);

% simulate the trained classifier
ysim_bst = sim(bst,x);

% find the training error rate of the 2 classifiers
% and display them both to compare
er_nobst = perfer(ysim_nobst,y);
fprintf('Error (no boosting) = %f\n', er_nobst);
er_bst = perfer(ysim_bst,y);
fprintf('Error (boosting) = %f\n', er_bst);
```

In Listing 8, weak learner training and simulation functions are shown that use functions from the Neural Network Toolbox. An example script that uses these as the AdaBoost weak learners is shown in Listing 9.

5.2 Cross-validation

Cross-validation involves involves partitioning the data set into training and testing sets. The `crossvalind` function in the Bioinformatics Toolbox can be used to perform holdout, k-folds, leave-m-out and resubstitution cross-validation. An example of k-folds cross-validation is shown in Listing 10.

Listing 10: Training with k-folds cross-validation using the Bioinformatics Toolbox

```
% load some toy training data
load databclass2d x y

% number of samples
nsamps = numel(y);

% array to store the classifier outputs
ysim = zeros(1,nsamps);

% choose the k-folds samples (using a function from the
% Bioinformatics Toolbox)
kfolds = 10;
ix = crossvalind('Kfold', nsamps, kfolds);

% train and simulate the classifiers
for k = 1:kfolds

    % select the next fold
    test_set = (ix==k);
    train_set = ¬test_set;

    % instantiate a GentleBoost object for this fold
    bst = newgab;

    % train a classifier on the next fold
    bst = train(bst,x(:,train_set),y(train_set),50);

    % simulate the trained classifier
    ysim(test_set) = sim(bst,x(:,test_set));

end

% find the error rate of the classifier
er = perfer(ysim,y);
fprintf('Error rate = %f\n', er);
```

5.3 Classifier accuracy

This toolbox provides a number of performance functions that can be used for evaluating the accuracy of AdaBoost classifiers (see Section 2.7.7). Other MATLAB toolboxes also provide other functions that can be used to further investigate performance.

For classification problems, the `classperf` object in the Bioinformatics Toolbox can be used to display a number of extra performance measures. Listing 11 shows how to display the confusion matrix for a classifier.

Figure 5: Confusion matrix generated by Listing 11

Listing 11: Classifier performance evaluated using the Bioinformatics Toolbox

```
% load some toy training data
load databclass2d x y

% instantiate a GentleBoost object
bst = newgab;

% train the classifier for 50 rounds
bst = train(bst,x,y,50);

% simulate the trained classifier
ysim = sim(bst,x);

% create a classifier performance object
cp = classperf(hardlim(y));
cp = classperf(cp, hardlim(ysim));
cp.DiagnosticTable
```

For regression problems, the `postreg` function in the Neural Network Toolbox can be used to assess the error of the regressor. The function both calculates the R-value of the regression and displays the regression results graphically. An example is shown in Listing 12.

Listing 12: Evaluating regression performance with the Neural Network Toolbox

```
% load some toy training data
load datareg3d x y

% instantiate a GentleBoost object
bst = newgabr;

% train the classifier for 50 rounds
% (use zero-mean shifted data)
bst = train(bst,x,y,50);

% simulate the trained classifier
ysim = sim(bst,x);

% regression analysis
postreg(ysim,y);
```

References

[1] Intel OpenCV. `http://opencv.sourceforge.net`.

[2] Mathworks technical support guide 1106. http://www.mathworks.com/support/tech-notes/1100/1106.html.

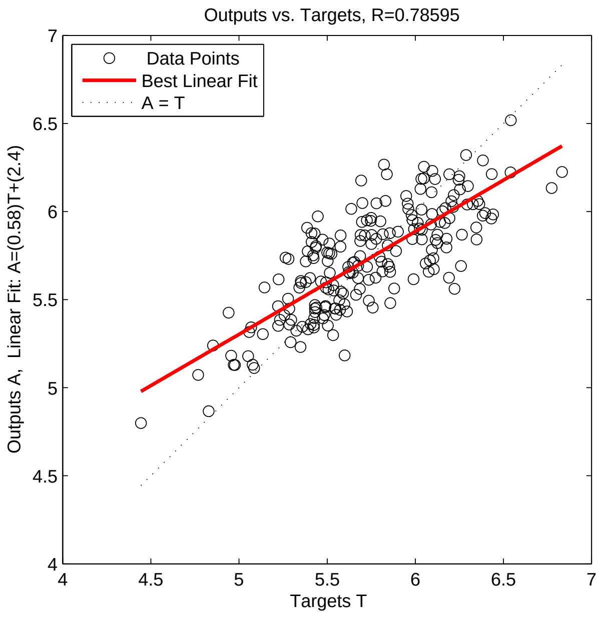


Figure 6: Regression analysis plot generated by Listing 12

[3] Esteban Alfaro, Noelia García, Matías Gámez, and David Elizondo. Bankruptcy forecasting: An empirical comparison of AdaBoost and neural networks. *Decision Support Systems*, 45(1):110–122, 2008.

[4] Zafer Barutçuoğlu and Ethem Alpaydın. A comparison of model aggregation methods for regression. In *Artificial Neural Networks and Neural Information Processing - ICANN/ICONIP 2003*, page 180. Springer Berlin / Heidelberg, 2003.

[5] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[6] David Cristinacce and Tim Cootes. Boosted regression active shape models. In *British Machine Vision Conference*, 2007.

[7] Cem Demirkir and Bülent Sankur. Face detection using look-up table based Gentle AdaBoost. In *Audio- and Video-Based Biometric Person Authentication (AVBPA)*, volume 3546, pages 339–345, Hilton Rye Town, NY, USA, July 2005. Springer Berlin / Heidelberg.

[8] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[9] Yoav Freund and Robert E. Schapire. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771–780, September 1999.

[10] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28(2):337–407, 2000.

[11] Stan Z. Li, Long Zhu, Zhen Qiu Zhang, Andrew Blake, Hong Jiang Zhang, and Harry Shum. Statistical learning of multi-view face detection. In *European Conference on Computer Vision*, pages 67–81, London, UK, 2002. Springer-Verlag.

[12] Shinya Nabatame and Hitoshi Iba. Integrative estimation of gene regulatory network by means of AdaBoost. In *International Conference on Genome Informatics*, page 119, 2006.

[13] Robi Polikar. Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*, 6(3):21–45, 2006.

[14] Robert E. Schapire and Yoram Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.

[15] Robert E. Schapire and Yoram Singer. Boostexter: A boosting-based system for text categorization. *Machine Learning*, 39(2-3):135–168, 2000.

[16] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pages 511–518. IEEE Computer Society, 2001.