AdaBoost with Various Weak Classifiers

Andrew Alex, *University of California, Los Angeles*
Patrick La Fontaine, *University of California, Los Angeles*

Abstract

Boosting algorithms are used to combine weak classifiers to form a committee that votes on the final classification. In this paper, we discuss our experimentation with multiple weak classifiers on a classification problem with multiple targets; specifically, we tested using decision stumps, decision trees and logistic regression models. We found that, in terms of both accuracy and speed, decision stumps performed best with a 3.3% error rate on the iris dataset from UC Irvine.

1. Introduction

AdaBoost is a specific type of boosting algorithm that "adapts" on each iteration to misclassified points by updating a set of weights. Each weak model is also given a weight α based on the number of points it correctly classifies. While this algorithm must be run sequentially because of the adaptive structure, weak classifiers are relatively fast to train, so performance can be kept high. As long, as the accuracy of any of the weak classifiers is better than random, AdaBoost can perform fairly accurately and is considered to be one of the best "out-of-the-box" classifiers. AdaBoost is inherently a binary classification tool, so in order to implement it for the multiclass case, we used a 1 vs. K scheme for each class and the final decision is chosen to be the class that the model deemed most likely.

1.1. The Data

The dataset we test and train on consists of 150 samples with 4 attributes each and 3 different target classes. One class is linearly separable from the other two, the others are not. In order to efficiently and accurately test our model, we randomly generated 5 different 80-20 training-testing splits of our data. Each class was assigned a numerical value before being processed by our algorithm. The implementation of the data generation can be found in the shell script *gen-data.sh*, which can be run on RHEL 6 systems to regenerate a new sets.

2. Weak Classifiers

In our experiment, we compared the results of three different weak classifiers. Each classifier had the common goal of optimizing the following error function:

$$J_m = \sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n) \quad (1)$$

where $w_n^{(m)}$ is the weight for the *n-th* point in the *m-th* iteration and I is the indicator function such that $I(y_m(x_n) \neq t_n)$ is 1 when $y_m(x_n) \neq t_n$ and is 0 otherwise. Each weak classifier optimized this in different ways, but each had to take into account that the data was weighted differently, which presented some challenges in using "textbook" implementations of the classifiers.

2.1. Decision Stumps

Decision stumps are perhaps the most naïve weak classifier. Each stump chooses an attribute from the data and attempts to minimize the error function (1). This is done by performing a 10000 step line search over the interval spanned by the attribute we are classifying. In the context of AdaBoost, each iteration of the algorithm chooses a different attribute to base the classification on. Decision stumps are particularly nice for AdaBoost because it is simple to incorporate data of varying weight and they are incredibly fast to train. In addition, all that must be stored is a decision boundary as a number, so it is incredibly storage efficient when the number of classifiers grows large. The following figure illustrates a single decision stump for the classification between two of our flower species:

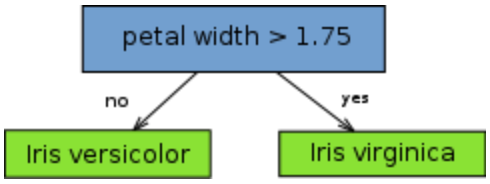

Figure 1. An example of a single decision stump

An entire AdaBoost model consists of several of these stumps, each with a different weight based on their accuracy compared to each other, so, for example, a very accurate stump is given a higher weight in the final classifier than a stump which is only slightly better than random. These weights will be discussed more precisely in the "Algorithm" section.

2.2. Decision Trees

Decision trees, next to decision stumps, are the easiest to understand of the weak classifiers. A decision tree can be seen as a series of if-then statements based on attributes of the data, with classification based on following these statements until we reach a leaf of the tree that classifies the test data point. To give a concrete example, consider Figure (2) which presents a tree considering the option of playing tennis on a given day.

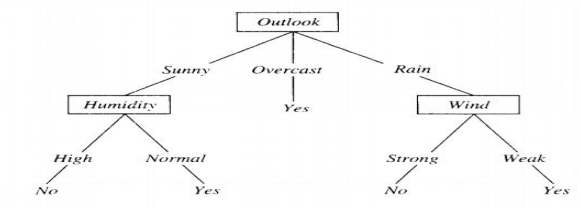

Figure 2. An example of a decision tree

Consider a day of which (Outlook = Sunny, Humidity = High, Wind = Strong), then this tree would sort the data point along the left-most branches leading to the final classification to not play tennis on that day.

To create the decision tree, we implemented the ID3 algorithm with some modifications adapted from Mitchell's ID3 paper *Decision Tree Learning*. The general idea of the algorithm is at each node in the tree, we evaluate a function for a given attribute of the data, which then returns a scalar value representing how well that attribute will split the data. We then split the data based off that attribute into a left and right branch, performing the same process with those two sets minus the attributes tested previously until we have a fully fleshed out tree. In this way, we split the data by testing the most general parameter to give the largest split, testing more specific attributes as we get further to the tree until we reach a leaf that classifies the data. To begin our formal discussion of decision trees requires the definition of a few core concepts.

First, we define the entropy of a given set as the impurity of a collection of samples respective to some target concept. In our case, the target concept was the binary classification of the training data. The entropy of a set of S observations is given by (2), in which P_+ and P_- are the proportion of positive and negative classifications in the set.

$$E(S) = -P_+ log_2(P_+) - P_- log_2(P_-) \quad (2)$$

Second, we evaluate the gain function at each node given by (3). This gain function measures the expected reduction in entropy caused by partitioning set S according to attribute A_i. The set Values(A_i) correspond to the possible values attribute A_i can take, in our case it was a binary value corresponding to our 1-of-K model classification of the training point. |S| is simply the number of elements in set S. To separate our data, we used the already-implemented decision stump to find the boundary and separate a given set for classification with respect to attribute A_i.

$$G(S, A_i) = E(S) - \sum_{v \in Values(A_i)} \frac{|S_v|}{|S|} E(S_v) \quad (3)$$

Now with the core concepts defined the ID3 algorithm can be discussed at each node, we have a set of training data: S, and a set of attributes: A = {A_1, A_2,..A_m).. At this node, we evaluate the gain function (3) for each A_i in A to find the attribute which maximizes the reduction in entropy. We then split the data according to the decision stump corresponding to this attribute, with positive classifications going to the left branch and negative classifications going to the right branch. This process is repeated recursively at each branch with the given split sets, and the list of attributes updated to A = A \ A_i in which A_i is the attribute tested at the previous level. The process is repeated until either the set of a given branch contains all positive or negative classifications, in which a leaf is created with corresponding classification, or until there are no more attributes to test, in which case classification is given as the classification corresponding to: max{P_+, P_-}.

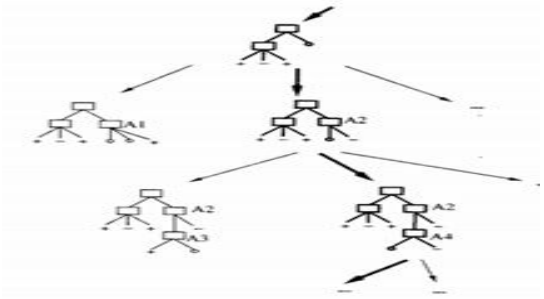

Figure 3. Visual of incomplete search of complete hypothesis space

The application of ID3 to our data set involved a decision tree of small depth. As our data set had only four attributes to test, the maximum depth of a tree was four. It's worth noting that, as two of the classes were linearly inseparable in two dimensions, trees of maximum depth less than four were extremely error prone and not implemented. Implementing less than four possible attributes in the ID3 algorithm couldn't allow for the depth of classification required for the data set. As a result of testing all four attributes, the creation of the decision tree is described as an incomplete search of a complete hypothesis space. In

this manner, the decision tree tested all possible attributes of our data (the complete hypothesis space), however it did not consider backtracking or parallel trees (the incomplete search). While there exists a definite probability of parallel trees existing which test attributes in a different order and lead to better classification results, as the ID3 algorithm considers only the max gain at each node with no insight into backtracking on itself to test parallel trees, the ID3 algorithm can only lead to a local minima of the error function (1).

2.3. Logistic Regression

The goal of logistic regression is to estimate a nonlinear relationship between features of the data and its classification. While this weak learner is a regression, its final output is a binary classification value corresponding to our 1-of-K model. Logistic regression in particular works well with linearly inseparable data as it transforms the data into a linearly separable form as seen in Figure (4).

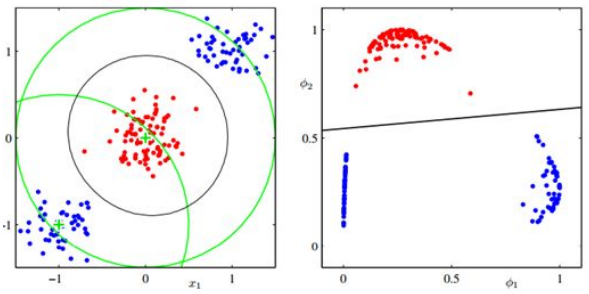

Figure 4. Visualization of basis function transformation

Before the implementation of logistic regression is discussed, background information on general logistic regression is required. To begin, we have N observations of training points, with X_n in R^D. We have t_n taking values in {0,1} corresponding to the classification of our training data. We define y_n as in (4), in which represents the sigmoid function, W represents the regression weights learned in the training data, and is , the data point x_n passed through our basis functions.

$$y_n = \sigma(W^T \phi_n) = \frac{1}{1+e^{-W^T \phi_n}} \qquad (4)$$

By applying the negative log likelihood to equation (5), we get the gradient of our error function (6):

$$p(t|W) = \prod_{n=1}^{N} y_n^{t_n} (1-y_n)^{(1-t_n)} \qquad (5)$$

$$\nabla E(W) = \sum_{n=1}^{N} (y_n - t_n) \phi_n \qquad (6)$$

While our gradient error function has no closed form solution, it is strictly convex meaning the application of iterative methods will lead us to a local minimum. To find the minimum of (6) we applied a standard gradient descent with fixed step size.

Finally, to discuss our implementation of logistic regression, we used a simple polynomial basis as described in (7) in which:

$$\phi(x) = [1, x_1^1, x_2^2, x_2^2, x_3^3, x_4^4] \qquad (7)$$

The algorithm is rather simple for logistic regression. After choosing our basis functions, we run a gradient descent on a modified form of (6) in which we add a weight element to the product as shown in (8). . This added weight point W_n comes from the Adaboost algorithm weight for misclassified point X_n, in this manner we take into account the misclassified points from the previous logistic regression, and can adjust our regression weights W accordingly. Finally, a new point x is classified by evaluating the sigmoid function at $W^T\Phi(x)$

$$\nabla E(W) = \sum_{n=1}^{N} (y_n - t_n) \phi_n * W_n \qquad (8)$$

There were a few variables in the logistic regression that lead to a higher error than the other two weak learners. The first of the problems came with the optimization of the gradient function (8). The initial weights, step size, and number of iterations of the gradient descent method, and additionally that we were only guaranteed a local minimum, caused the regression weights W to be inaccurate in classifying all points. Second, the choice of basis functions was a major factor in determining the regression weights, and thus the error. While the polynomial basis as shown in figure (5) did well to separate the data from the pre-basis transformation compared to the post-basis transformation, there still exists a clear area in which the boundary between classes two and three is not separable. While logistic regression can separate linearly inseparable data, the fact that our data took inseparable values for two classes in two dimensions causes the basis function choice to not be able to separate them entirely.

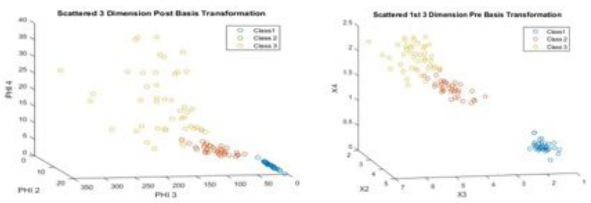

Figure 5. Pre basis transform (right) vs post basis (left)

3. The AdaBoost Algorithm

The official AdaBoost algorithm, as partially presented in Bishop's *Pattern Recognition and Machine Learning* is as follows:

1. Initialize the data weight coefficients $\{w_n\}$ by setting $w_n^{(1)} = 1/N$ for each $n = 1, \ldots, N$.
2. For $m = 1, \ldots, M$:
   a. Fit a classifier $y_m(x)$ to the training data by minimizing (1)
   b. Evaluate the quantities ε_m given by:

$$\varepsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}} \quad (9)$$

which are then used to compute the weights of the model α_m:

$$\alpha_m = ln(\frac{1 - \varepsilon_m + \varepsilon_0}{\varepsilon_m + \varepsilon_0}) \quad (10)$$

Note that the ε_0 term is not a part of the actual algorithm, but included in our implementation for numerical stability when ε_m is very small.

   c. Update the data weights:

$$w_n^{(m+1)} = w_n^{(m)} exp\{\alpha_m I(y_m(x_n) \neq t_n)\} \quad (11)$$

3. Adjust the α_m terms to sum to one
4. Make predictions using the final model:

$$Y_m = \sum_{m=1}^{M} \alpha_m y_m(x) \quad (12)$$

We omitted the sign() function because in our implementation, we were concerned about how close (5) was to 1 because we were comparing several AdaBoost 1 vs. K schemes to make the final classification. For example, in the Iris dataset we used for experiments, because there were 3 distinct target classes, we ran this algorithm 3 different times and compared the different 1 vs. K results. Figure (2) from Bishop gives a good visual representation of a single AdaBoost model.

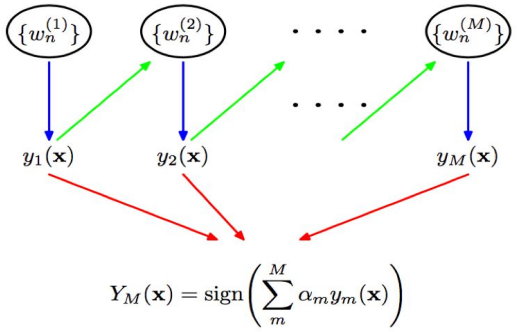

Figure 2. An illustration of boosting with base classifiers y_m

4. Experimental Results

To test our implementation, we measured the error of a model for varying numbers of base classifiers. Additionally, because training speed differed greatly between each model, we report the average time over 3 runs to train a model of 50 weak classifiers.

4.1. Accuracy and Convergence

The following scatter plots show how our implementation converges using each of the different weak classifiers:

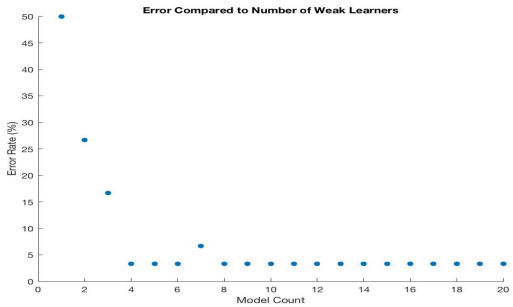

Figure 6. Error using differing numbers of decision stumps

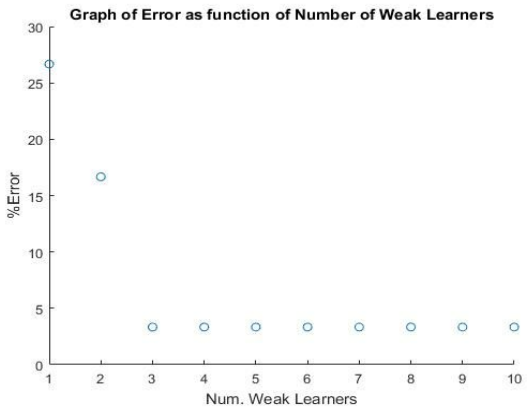

Figure 7. Error using differing numbers of decision trees

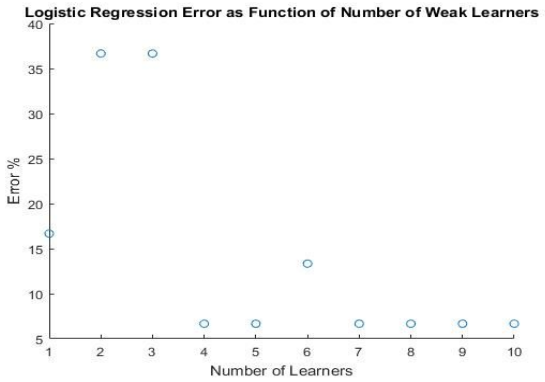

Figure 8. Error using differing number of logistic regression models

These accuracy tests were performed using our first training-testing split because the models did best on this set. From the graphs, it is clear decision stumps and decision trees performed relatively similar in terms of accuracy, whereas logistic regression did not perform as well.

4.2. Execution Time

Because both the logistic regression and decision stump implementations involve an iterative search, in order to make the following test fair, we set the max iterations for each to be 10,000. The following are the results of the tests (averaged over 3 runs):

Decision Stump: 8.2 secs

Decision Tree: 76.9 secs

Logistic Regression: 285.9 secs

We should also note that only logistic regressions accuracy improved by training 30 more models. It went from 6.6% to 0% on test set 1, which these timing tests were performed on.

5. Conclusion and Recommendations

From the results of our experiments, we conclude that decision stumps perform best in terms of accuracy and execution time with Adaboost. While a small number of decision stumps perform poorly, training several is relatively fast and clearly gives good results. Adaboost is clearly a powerful tool, which turned a model (decision stumps) from about 50% error down to 3.3% error. It's power also lies in the fact that it can be used with almost any classification method, so there is much more experimentation that can be done to learn more.