

Multi-class Boosting

Mario Rodriguez
Department of Computer Science
University of California, Santa Cruz
mrod@soe.ucsc.edu

December 13, 2009

Abstract

This paper briefly surveys existing methods for boosting multi-class classification algorithms, as well as compares the performance of one such implementation, Stagewise Additive Modeling using a Multi-class Exponential loss function (SAMME), against that of Softmax Regression, Classification and Regression Trees, and Neural Networks.

1 Introduction and Motivation

The goal in classification is to take an input vector \mathbf{x} and to assign it to one of K discrete classes C_k where $k = 1, \dots, K$. In the most common scenario, the classes are taken to be disjoint, so that each input is assigned to one and only one class. This is the nature of the classification problem we consider in this paper, where we discuss learning a classifier which predicts wine quality based on physicochemical data. The data used was the Wine Quality Dataset available for download at the UCI Machine Learning Repository [4] and described in detail by Cortez et al. in [2]: “Regarding the preferences, each sample was evaluated by a minimum of three sensory assessors (using blind tastes), which graded the wine in a scale that ranges from 0 (very bad) to 10 (excellent). The final sensory score is given by the median of these evaluations.”

None of the wines present in the white wine dataset received a score of 0, so we only considered 10 discrete classes in which to classify any given white wine. It is worth noting that given the ordered nature of the target variable (values from 1 to 10), treating the problem as a regression problem rather than a classification problem would have been a sound strategy (indeed such is the strategy pursued by Cortez et al.). In addition, researchers have argued that modeling preferences is neither a pure regression (the target is, after all, discrete) nor a pure classification problem (the inherent structure present in the ordering of the data is not exploited), and consider methods designed explicitly for ranking (Rajaram et al. [6] provide an interesting approach as well as a discussion of several such methods).

2 Multi-class Boosting

Boosting is a technique to improve the performance of any given learning algorithm, generally consisting of sequentially learning classifiers¹ with respect to a distribution and adding them to an ensemble. When classifiers are added to the ensemble, they are typically weighted in some way that is related to their accuracy. After adding a classifier, the data is also reweighted: examples that are misclassified gain weight and examples that are classified correctly lose weight, thus forcing the next classifier to focus on previously hard to classify data points. This basic idea has been surprisingly successful, with performances comparable to more complex methods such as Support Vector Machines. In fact, a particular boosting algorithm, AdaBoost, when used with trees as the component classifiers, has been referred to as the “best off-the-shelf classifier in the world [3].”

```
for  $m = 1 : M$ 
  // Inputs:
  //  $\mathbf{X}$  is the covariate matrix
  //  $\mathbf{y}$  is the target vector
  //  $\mathbf{w}$  is the vector of observation weights, initially set to  $w_i = 1/N, i = 1, 2, \dots, N$ 
  // Outputs:
  //  $h_m$  is the classifier
  //  $\mathbf{d}$  is a vector of length  $N$  with  $d_i = \mathbf{I}[y_i \neq h_m(x_i)]$  for  $i = 1, 2, \dots, N$ 
  // error =  $\mathbf{d}^T \mathbf{w}$ 
   $h_m, \mathbf{d}, \text{error} = \text{learnClassifier}(\mathbf{X}, \mathbf{y}, \mathbf{w})$ 

  if error  $\leq 0$  or error  $\geq 1/2$ 
    break

   $\alpha_m = \log((1 - \text{error}) / \text{error})$ 

  //  $\mathbf{E}$  is the ensemble
   $E_{m,1} = \alpha_m$ 
   $E_{m,2} = h_m$ 

   $w_i = w_i \cdot \exp(\alpha_m \cdot (d_i \cdot 2 - 1))$  for  $i = 1, 2, \dots, N$ 
   $\mathbf{w} = \text{normalize}(\mathbf{w})$ 
```

Algorithm 1: AdaBoost

¹Boosting methods have been developed to deal with regression problems as well [5].

AdaBoost, originally intended for boosting binary classifiers, has only 2 requirements of its component classifiers: (1) that they have an accuracy greater than 50%, and (2) that they be able to handle weighted data points. Failing the first requirement would cause the observation weights to not be updated (if accuracy was exactly 50%) or to be updated in the opposite direction (if accuracy was worse than 50%). The second requirement can easily be bypassed if we sample from the training dataset with replacement (according to the weight distribution) to generate a new dataset, unweighted, that is then passed to the component classifier.

The number of boosting iterations depicted in Algorithm 1, M , is a meta-parameter than can be estimated using cross-validation. When AdaBoost is asked to classify an unseen data point, each classifier in the ensemble contributes its own α_m to the class it predicts, and in the end, the class with the highest value is chosen as the ensemble's prediction.

AdaBoost.M1 is a trivial extension of AdaBoost to the multi-class classification problem, in which the only modification is that the component classifiers must be capable of multi-class classification. However, the component classifiers are still required to have an accuracy greater than 50% (for the same reason mentioned earlier). This requirement may place an undue constraint on the type of classifiers that can be boosted, and so several approaches have been designed to lift this restriction. One viable option is to transform the multi-class classification problem into several binary classifications problems. This can be accomplished by using one of 2 popular approaches: (1) one-against-all and (2) one-against-one.

The **one-against-all** strategy consists of constructing one model per class, where each model is trained to distinguish the samples of one class from the samples of all remaining classes. Usually, classification of an unknown pattern is done according to the maximum output among all models. In other words, if we have a probabilistic binary classifier for each class, we query each classifier with an unknown pattern and receive back a class prediction and a probability for that class. The class prediction of the classifier that returns the highest probability (intuitively, the most confident classifier) is chosen for the unknown pattern.

On the other hand, the **one-against-one** strategy consists of constructing one model for each pair of classes. Thus, for a problem with K classes, $K(K-1)/2$ models are trained to distinguish the samples of one class from the samples of another class. Usually, classification of an unknown pattern is done according to the maximum voting, where each model votes for one class. Although this sounds unnecessarily computationally intensive, it is not. In fact, if the classes are evenly populated, pairwise classification is at least as fast as any other multi-class method. The reason is that each pairwise learning problem only involves instances pertaining to the two classes under consideration. If N instances are divided evenly among K classes, this amounts to $2N/K$ instances per problem. Suppose the learning algorithm for a binary classification problem with N instances takes time proportional to N seconds to execute. Then the run time for pairwise classification is proportional to $K(K-1)/2 \cdot 2N/K$

class	code word
<i>a</i>	1 0 0 0
<i>b</i>	0 1 0 0
<i>c</i>	0 0 1 0
<i>d</i>	0 0 0 1

Table 1: One-against-all class encoding

seconds, which is $(K - 1)N$. In other words, the method scales linearly with the number of classes. If the learning algorithm takes more time, say proportional to N^2 , the advantage of the pairwise approach becomes even more pronounced.

These 2 approaches are special cases of a more general approach based on the concept of **Error-Correcting Output Coding (ECOC)**, as presented by Allwein et al. in [1]. ECOC is a method for making the most of the transformation of the multi-class problem into several binary problems. A simple ECOC method is known as **Hamming encoding**, and it can be easily illustrated with an example drawn from Witten and Frank [8]. Consider a multi-class problem with the four classes *a*, *b*, *c*, and *d*, which will be transformed into 4 binary classification problems using the one-against-all approach described earlier. The transformation can be visualized as shown in Table 1. Each of the original class values is converted into a 4-bit code word, 1 bit per class, and the 4 classifiers predict the bits independently. Interpreting the classification process in terms of these code words, errors occur when the wrong binary bit receives the highest confidence.

However, we do not have to use the particular code words in Table 1. Indeed, there is no particular reason why each class must be represented by 4 bits. Look instead at the code of Table 2, where classes are represented by 7 bits. When applied to a dataset, 7 classifiers must be built instead of 4. To see why this might be useful, consider the classification of a particular instance. Suppose it belongs to class *a*, and that the predictions of the individual classifiers are 1 0 1 1 1 1 1 (respectively). Obviously, comparing this code word with those in Table 2, the second classifier has made a mistake: it predicted 0 instead of 1. However, comparing the predicted bits with the code word associated with each class, the instance is clearly closer to *a* than to any other class. This can be quantified by the number of bits that must be changed to convert the predicted code word into those of Table 2: the **Hamming distance**, or the discrepancy between the bit strings, is 1, 3, 3, and 5 for the classes *a*, *b*, *c*, and *d*, respectively. We conclude, then, that the second classifier made a mistake and correctly identify *a* as the instance’s true class.

That kind of error correction is not possible with the code words of Table 1, because any predicted string of 4 bits other than these four 4-bit words has the same distance to at least 2 of them. Thus, the output codes in Table 1 are not “error correcting.”

class	code word
<i>a</i>	1 1 1 1 1 1 1
<i>b</i>	0 0 0 0 1 1 1
<i>c</i>	0 0 1 1 0 0 1
<i>d</i>	0 1 0 1 0 1 0

Table 2: Error-correcting class encoding

Transforming the multi-class classification problem into several binary classification problems is particularly useful if the component classifiers used are inherently incapable of producing multi-class predictions. How can we take advantage of component classifiers that do not suffer from this limitation?

One particular algorithm, SAMME, is strikingly similar to the original AdaBoost algorithm and is capable of handling multi-class component classifiers (differences are in red):

```

for  $m = 1 : M$ 
  // Inputs:
  //  $\mathbf{X}$  is the covariate matrix
  //  $\mathbf{y}$  is the target vector
  //  $\mathbf{w}$  is the vector of observation weights, initially set to  $w_i = 1/N, i = 1, 2, \dots, N$ 
  // Outputs:
  //  $h_m$  is the classifier
  //  $\mathbf{d}$  is a vector of length  $N$  with  $d_i = \mathbf{I}[y_i \neq h_m(x_i)]$  for  $i = 1, 2, \dots, N$ 
  // error =  $\mathbf{d}^T \mathbf{w}$ 
   $h_m, \mathbf{d}, \text{error} = \text{learnClassifier}(\mathbf{X}, \mathbf{y}, \mathbf{w})$ 

  if error  $\leq 0$  or error  $\geq (1 - (1/K))$ 
    break

   $\alpha_m = \log((1 - \text{error}) / \text{error}) + \log(K - 1)$ 

  //  $\mathbf{E}$  is the ensemble
   $E_{m,1} = \alpha_m$ 
   $E_{m,2} = h_m$ 

   $w_i = w_i \cdot \exp(\alpha_m \cdot (d_i \cdot 2 - 1))$  for  $i = 1, 2, \dots, N$ 
   $\mathbf{w} = \text{normalize}(\mathbf{w})$ 

```

Algorithm 2: SAMME

Stagewise Additive Modeling using a Multi-class Exponential loss function, **SAMME**, is a natural extension of AdaBoost to the multi-class case: if $K = 2$, we have AdaBoost. A key detail is that component classifiers are no longer required to have an accuracy greater than 50%, but instead need only be better than random guessing.

SAMME was developed by Zhu et al. in [9], where they include empirical tests which show SAMME’s performance to be comparable, if not slightly better, than that of **AdaBoost.MH** (essentially, a one-against-all approach developed by Schapire and Singer [7]).

The modifications that turned AdaBoost into SAMME are not arbitrary, they come from a statistical perspective of AdaBoost in which it fits an additive expansion in a set of elementary “basis” functions which seeks to optimize an exponential loss function:

$$L(y, f(x)) = \exp(-yf(x)) \tag{1}$$

The basis functions are the component classifiers, h_m , and the expansion coefficients are each of the component classifiers’ own α_m , $m = 1, 2, \dots, M$. The basis function expansion takes the form:

$$f(x) = \sum_{m=1}^M \alpha_m h_m \tag{2}$$

Forward Stagewise Additive Modeling approaches the optimization problem by sequentially adding new basis functions to the expansion without adjusting the parameters and coefficients of those that have already been added; same as boosting.

For the K -class classification problem, SAMME encodes the target variable as $\mathbf{y}_i = (y_1, \dots, y_K)^T$, $i = 1, 2, \dots, N$, with

$$Y_k = \begin{cases} 1, & \text{if } y_i = C_k \\ -\frac{1}{K-1}, & \text{otherwise} \end{cases} \tag{3}$$

Then, if $f = (f_1, \dots, f_K)^T$ and $\sum_{k=1}^K f_k = 0$, the multi-class loss optimized by SAMME is

$$L(y, f) = \exp\left(-\frac{1}{K} Y^T f\right) \tag{4}$$

AdaBoost and SAMME are the iterative algorithms to optimize said exponential loss functions (derivation details can be found in [9] and [3]).

3 Comparators

Three algorithms were chosen as comparators to be run against the wine dataset. They were chosen for the distinct kinds of decision boundaries each algorithm is capable of:

1. SoftMax Regression: linear decision boundaries
2. Classification and Regression Trees (CART): jagged decision boundaries (between linear and non-linear)
3. Neural Networks: non-linear decision boundaries

For **SoftMax Regression**, also known as Multi-class Logistic Regression, we model the target variable as distributed according to a multinomial distribution conditioned on the covariates and parameterized by $K - 1$ parameter vectors of length p , where p is the number of covariates. In other words, our hypothesis outputs the estimated probability that $p(y = i|x; \theta)$, for every value of $i = 1, \dots, K$, and even though $h_\theta(x)$ is only $K - 1$ dimensional, $p(y = K|x; \theta)$ can be obtained as $1 - \sum_{i=1}^{K-1} p(y = i|x; \theta)$. Finally, we use **Batch Gradient Ascent** to find the parameter matrix θ that maximizes the log-likelihood of our data:

$$\ell(\theta) = \sum_{i=1}^N \log p(y^{(i)}|x^{(i)}; \theta) \tag{5}$$

$$\ell(\theta) = \sum_{i=1}^N \log \prod_{l=1}^K \left(\frac{e^{\theta_l^T x^{(i)}}}{\sum_{j=1}^K e^{\theta_j^T x^{(i)}}} \right)^{1_{\{y^{(i)}=l\}}} \tag{6}$$

For **Classification and Regression Trees**, we recursively and greedily perform binary splits on the data at each tree node (starting from root), on the attribute that enables us to minimize the entropy impurity, I , of the immediate descendent nodes:

$$I(J) = - \sum_{i=1}^K p(y_i) \log_2 p(y_i) \tag{7}$$

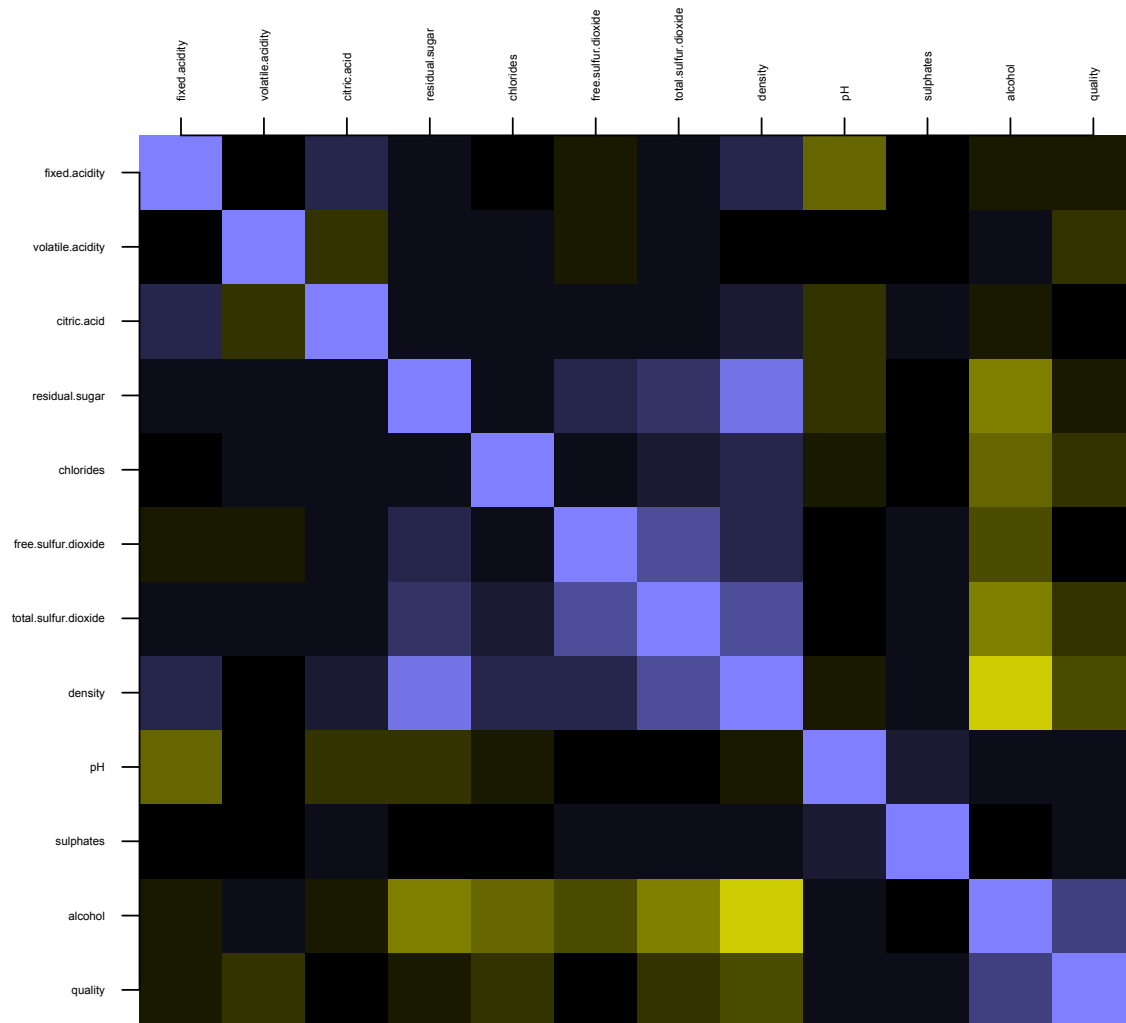
Where $p(y_i)$ is the fraction of patterns at tree node J that belong to class y_i . The tree is grown from each node until support at that node is broken, support being defined as the minimum number of data points that a node requires in order to be eligible for splitting (a value that can be determined with cross-validation). In addition, no pruning of a grown tree is performed.

Finally, for **Neural Networks**, a standard 3-layer feedforward backpropagation neural net was used, where the hidden layer and the output layer consisted of non-linear activation functions (logistic units), and where the optimal number of nodes in the hidden layer can be determined using cross-validation. The total number of weights in the neural net is given by:

$$Weight\ Count = \begin{cases} 11 \cdot Nh + & // \text{ contributed by edges from input layer to hidden layer} \\ 1 \cdot Nh + & // \text{ contributed by edge from bias to hidden layer} \\ 10 \cdot Nh + & // \text{ contributed by edges from hidden layer to output layer} \\ 1 \cdot 10 + & // \text{ contributed by edge from bias to hidden layer} \end{cases} \quad (8)$$

4 Experiments

Below we have a correlation matrix of the wine data:

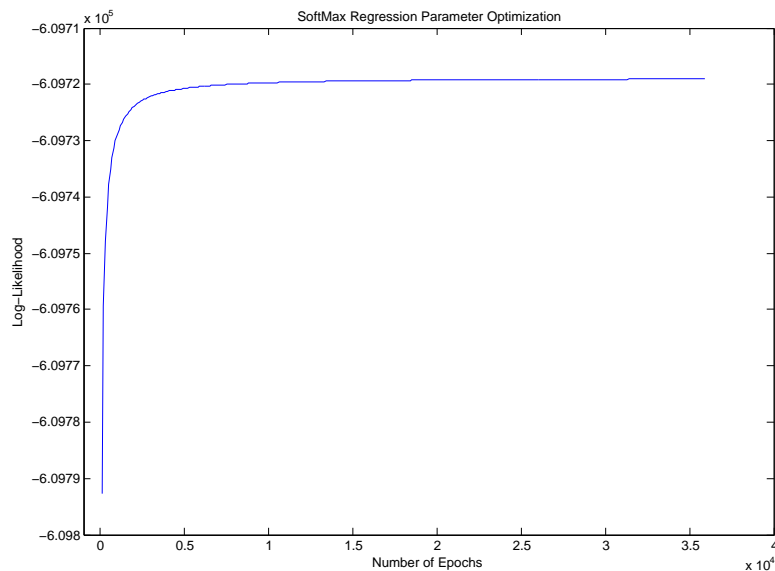


Blue values indicate high positive correlation, dark values indicate no correlation, and yellow values indicate high negative correlation. **It is interesting to note that the level of alcohol content is what is most positively correlated with perceived wine quality.**

Regarding the dataset, it is composed of approximately 5000 training points, with 11 real-valued covariates and one discrete target variable (a 1-10 ranking of wine quality).

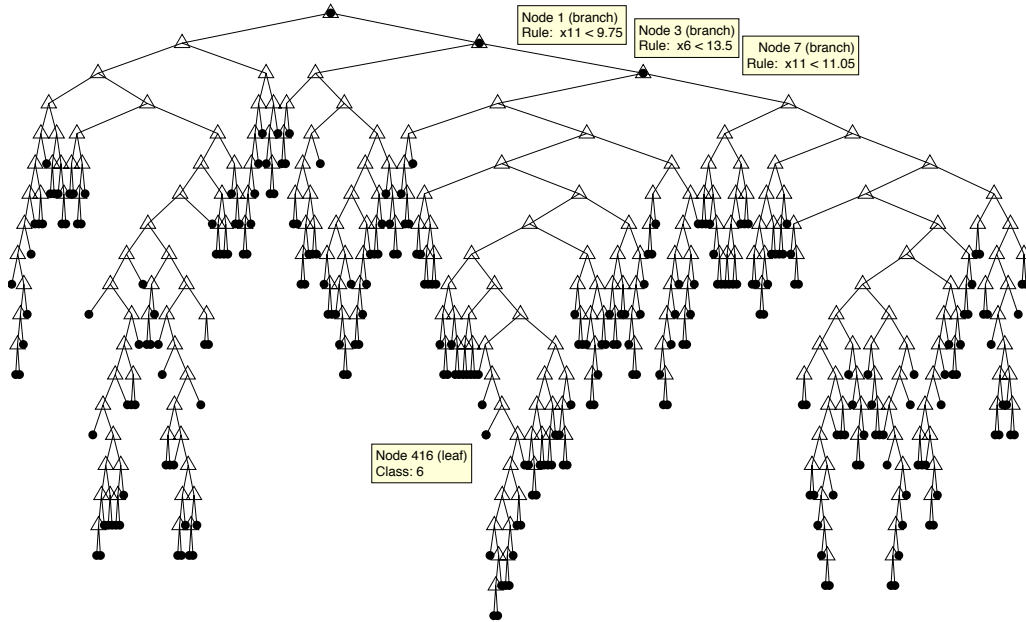
The data was only transformed prior to running SoftMax Regression and Neural Networks, where the covariates were normalized to have zero mean and unit standard deviation. The convention of adding an additional, artificial, unit feature vector (bias) was followed for SoftMax Regression and Neural Networks as well.

Below we have a plot for the evolution of the log-likelihood of the data given the SoftMax Regression model as it is being optimized:



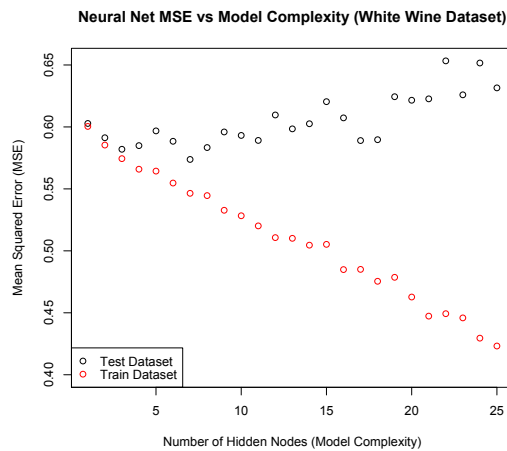
Student Version of MATLAB

For illustration purposes, here is what the trees looked like after training (true answers to each binary split go left, whereas false answers go right):

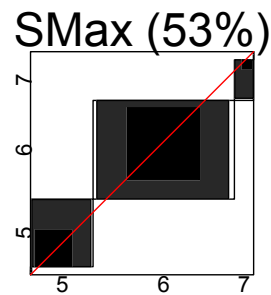
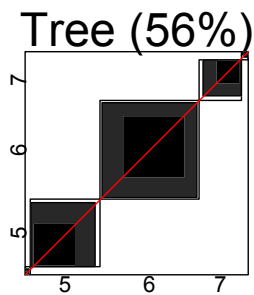
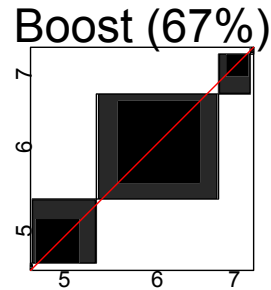
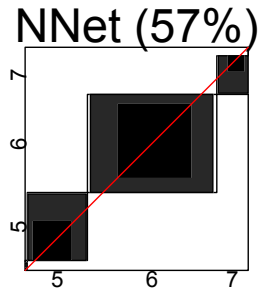


Student Version of MATLAB

For choosing the optimal number of hidden neurons in the Neural Network, a total of 20 neural networks were examined, with the results that 7 hidden nodes appear to be the threshold after which the network starts to overfit (yielding a total of 164 weights to be fitted vs 108 for SoftMax Regression):



Those 3 models were compared with SAMME using trees as the component classifiers, and how they fared can be determined by analyzing each algorithm's confusion matrix below (most of the ratings given were 5, 6, and 7):

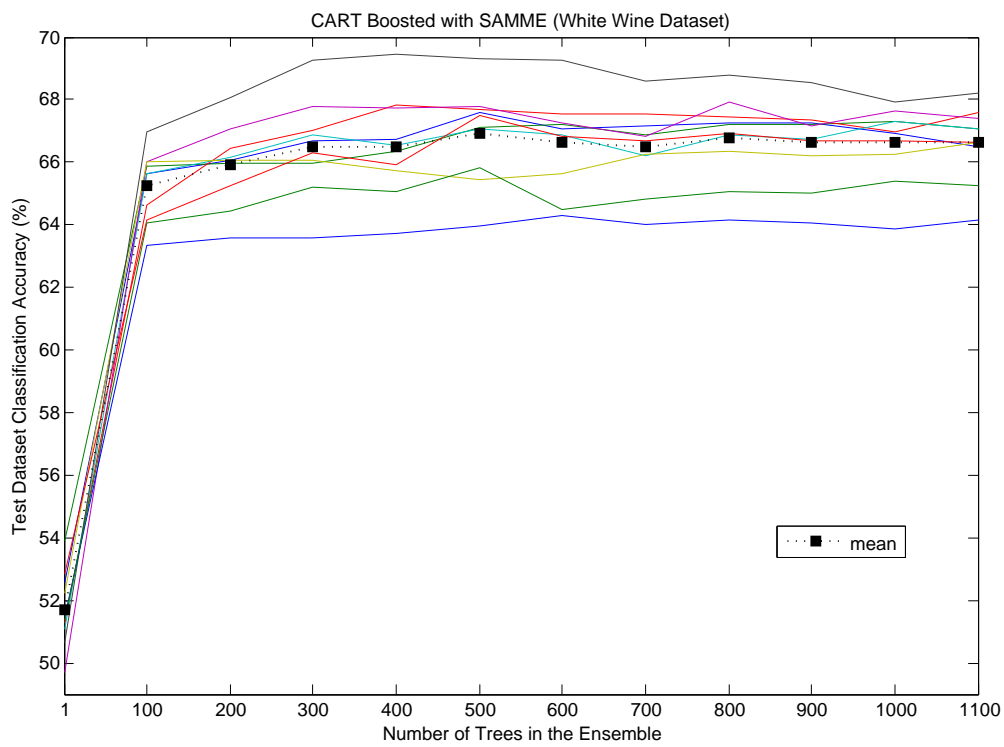


Boosting clearly outperformed the other models, including the ones in the original paper [2], where they followed the regression approach and obtained the results in table 3 using Multiple Regression (MR), Neural Networks (NN), and Support Vector Machine (SVM). In Table 3, T refers to Tolerance, a measure of how much wiggle room a prediction has in order to be considered accurate.

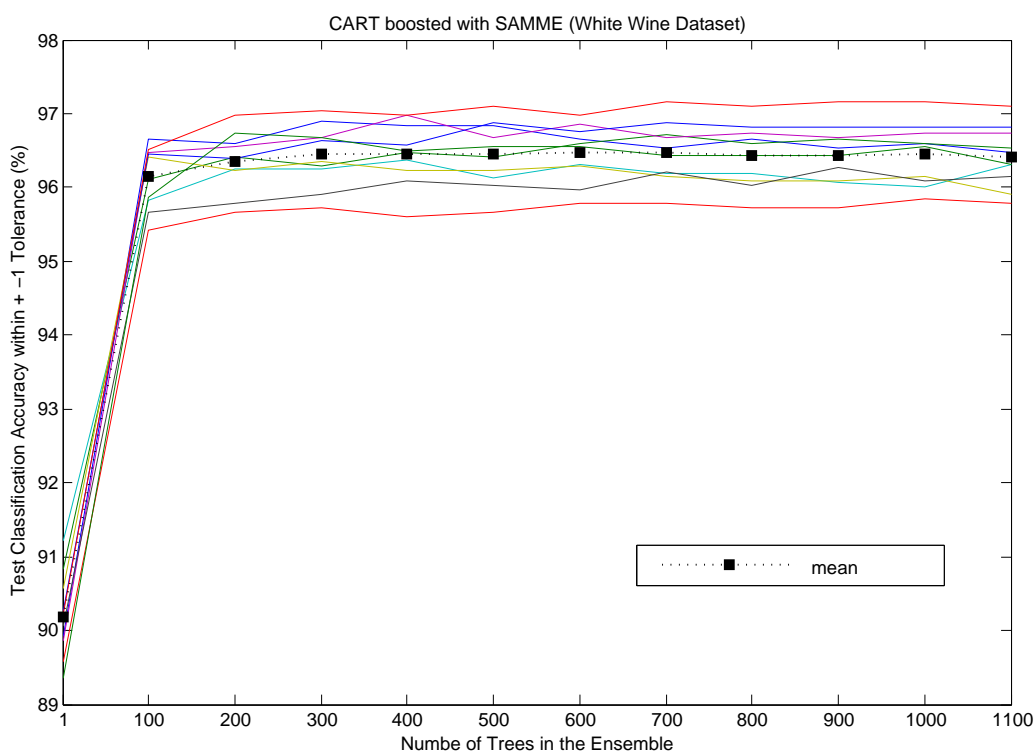
Next, we have a plot of 10 runs of boosting, where we attempt to determine whether there is any amount of overfitting as we continue to add component classifiers to our ensemble (there appears to be no overfitting):

White Wine			
	MR	NN	SVM
Accuracy ($T = \pm 0.25$) (%)	25.6	26.5	50.3
Accuracy ($T = \pm 0.50$) (%)	51.7	52.6	64.6
Accuracy ($T = \pm 1.00$) (%)	84.3	84.7	86.8

Table 3: Cortez et al. [2] results

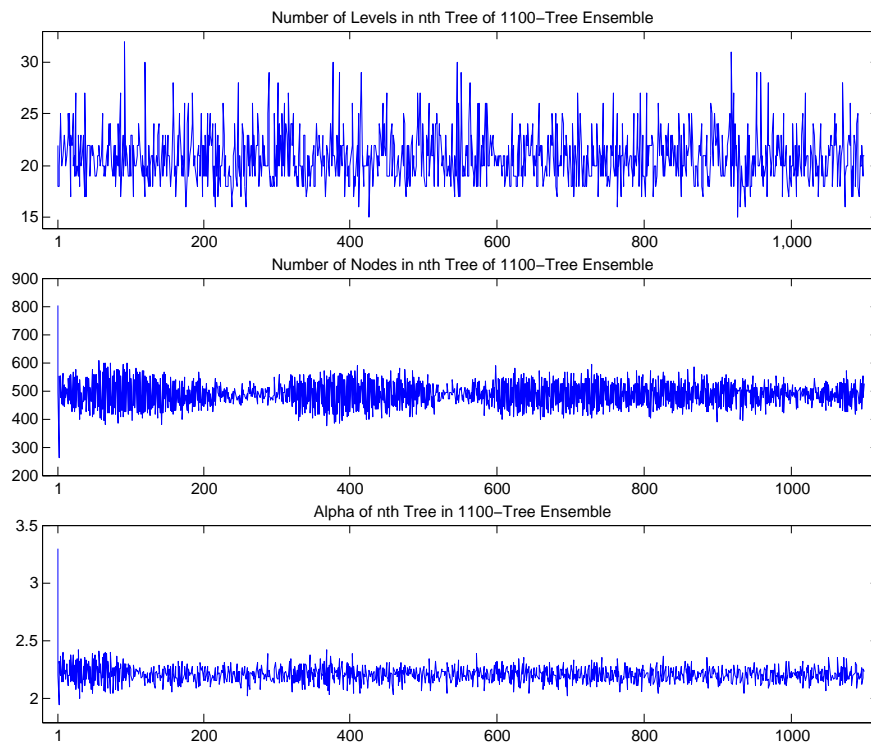


The results we have achieved can be naturally thought of as having a Tolerance of $T = \pm 0.50$. If we allow for a Tolerance of $T = \pm 1.00$, the results are even more impressive:



Student Version of MATLAB

Finally, below we have some interesting statistics on the trees that make up the ensemble. The trees themselves are fairly unbalanced in general. In addition, there is a sharp contrast between the first couple of trees and the remaining trees: the first couple of trees have a higher number of nodes (but are also much more balanced) and have a higher α , which indicates that their error on the data was lower. Also, from the previous plots, we see that adding about 100 trees to the ensemble provides us with the bulk of the improvement in accuracy, and this phenomena can also be observed on the α plot below, where for the first few 110-120 or so trees the α is higher than average, and then significantly drops off momentarily before settling into a fairly stable pattern.



5 Conclusions

A quick recap of the notable observations:

1. The level of alcohol content is the only covariate significantly correlated (positively or negatively) with the perceived wine quality (Pearson correlation of +0.44)
2. The model ranking, by increasing level of accuracy was: (1) SoftMax Regression (53%), (2) Classification and Regression Trees (56%), (3) 3-Layer Neural Network (57%), and (4) Classification and Regression Trees Boosted with SAMME (67%)
3. No overfitting was observed while boosting trees
4. The bulk of the increase in accuracy thanks to boosting came from the first 120 trees in the ensemble (it took an average of 1.6 seconds to learn each tree, with no significant overhead added by the SAMME code)
5. Boosting trees outperformed the models built by the authors in [2]

Boosting trees has received significant attention lately (especially for supervised learning approaches to data mining) and understandably so. Boosting minimizes the disadvantages of trees (predictive power, ability to extract linear combinations of features), while preserving and/or maximizing their strengths (natural handling of mixed data types, handling of missing values, robustness to outliers in input space, insensitivity to monotone transformations of inputs, computational scalability, ability to deal with irrelevant inputs).

References

- [1] E. Allwein, R. Schapire, and Y. Singer. Reducing multiclass to binary: A unifying approach for margin classifiers, 2000.
- [2] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. Modeling wine preferences by data mining from physicochemical properties, 2009.
- [3] T. Hastie, R. Tibshirani, and J. Friedman. Elements of statistical learning: data mining, inference, and prediction, 2009.
- [4] U. C. Irvine. Machine learning repository. <http://archive.ics.uci.edu/ml/datasets/Wine+Quality>.
- [5] R. Nock and F. Nielsen. A real generalization of discrete adaboost, 2006.
- [6] S. Rajaram, A. Garg, X. S. Zhou, and T. S. Huang. Classification approach towards ranking and sorting problems, 2003.

- [7] R. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions, 1999.
- [8] I. H. Witten and E. Frank. Data mining: Practical machine learning tools and techniques, 2005.
- [9] J. Zhu, S. Rosset, H. Zou, and T. Hastie. Multi-class adaboost, 2006.