AdaBoost And Its Variants: Boosting Methods For Classification With Small Sample Size And Brain Activity In Schizophrenia

Brittany Perry
Brock University

May 3, 2023

Abstract

AdaBoost is an ensemble method that can be used to boost the performance of machine learning algorithms by combining several weak learners to create a single strong learner. The most popular weak learner is a decision stump (low depth decision tree). One limitation of AdaBoost is its effectiveness when working with small sample sizes. This work explores variants to the AdaBoost algorithm such as Real AdaBoost, Logit Boost, and Gentle AdaBoost. These variants all follow a gradient boosting procedure like AdaBoost, with modifications to the weak learners and weights used. We are specifically interested in the accuracy of these boosting algorithms when used with small sample sizes. As an application, we study the link between functional network connectivity (as measured by EEG recordings) and Schizophrenia by testing whether the proposed methods can classify a participant as Schizophrenic or healthy control based on quantities measured from their EEG recording.

Keywords: AdaBoost , decision trees, small sample size, gradient boosting, Schizophrenia

Contents

1 Introduction **1**
- 1.1 Decision Trees . . . 1
- 1.2 Boosting . . . 4
- 1.3 AdaBoost . . . 5

2 Literature Review **6**
- 2.1 Forward Stagewise Additive Modelling . . . 6
- 2.2 Relation between AdaBoost and Forward Stagewise Additive Modelling 8
- 2.3 Gradient Boosting . . . 10
- 2.4 Variants of AdaBoost . . . 11
  - 2.4.1 Real AdaBoost . . . 11
  - 2.4.2 Gentle AdaBoost . . . 12
  - 2.4.3 Logit Boost . . . 13

3 Simulation Study **14**
- 3.1 Data Generation Process . . . 14
  - 3.1.1 Number of Observations . . . 14
  - 3.1.2 Number of Classes . . . 15
  - 3.1.3 Nonlinear Data . . . 15
- 3.2 Results . . . 16
  - 3.2.1 Binary Linear Model . . . 16
  - 3.2.2 Multi Class Linear Model . . . 16
  - 3.2.3 Binary Nonlinear Model . . . 17
  - 3.2.4 Multi Class Nonlinear Model . . . 17
- 3.3 Logistic Regression vs. Boosting Models . . . 18

4 Application **19**
- 4.1 Data . . . 20
- 4.2 Results . . . 20

5 Conclusions **21**

6 References **22**

7 Appendix **23**

1 Introduction

1.1 Decision Trees

Decision trees are a type of machine learning methods that are often used today, given the fact that they provide an intuitive visual output and are effective in their predictions. They are a type of supervised machine learning algorithm that predict the target outcome for a given data set, based on a recursive splitting procedure of the data. Decision tree algorithms can be used for both classification and regression problems. For the purpose of this project we will be focusing on decision trees used for classification problems, as well as adjusting this to regression decision trees when necessary. Decision trees are constructed by splitting the input space into parts (referred to as *leaves* or *nodes*) using a set of conditions until a stopping criteria is met. This stopping criteria could be a specific number of splits, or when the prediction accuracy is no longer changing. In general, finding the optimal partition is a difficult problem so decision trees simplify this by using a recursive splitting procedure, where each split is based on a single input. The input space is split into parts and a constant function $y = \hat{f}(x)$ is estimated within each part. This function is estimated as the following,

$$\hat{f}(x) = \sum_{m=1}^{M} \hat{c}_m \cdot I_{R_m}(x)$$

where given a partition $\{R_m\}$, we can estimate $\hat{c}_m$ as the average of the observations in the mth part,

$$\hat{c}_m = \frac{\sum_{i=1}^{n} I_{R_m}(x_i) \cdot y_i}{\sum_{j=1}^{n} I_{R_m}(x_j)}.$$

In this case, we have the set of parts R_m for $m = 1,...,M$ where M is equal to the total number of parts the input space is split into. The function $I_{R_m}(x)$ represents an indicator function equal to 1 when an observation falls in the mth part and 0 otherwise, n is equal to the total number of observations in the given part, and x_i and y_i are the values of the x and y variables corresponding to the ith observation. Thus, for a given part R_m, $\hat{c}_m$ is estimated as the average y value for the mth part and $\hat{f}(x)$ estimates the sum of these average values for each new x observation that falls in the mth part.

When working with decision trees, the *root node* represents the full data set before any splits occur and *branches* split and create new nodes until a stopping criteria is met. The nodes in which no more splits occur after are known as *leaf nodes* (or leaves) and the nodes followed by another split are called *interior nodes.* This is where decision trees get their name from, as when visualizing this algorithm it resembles a tree-like structure growing downward. Another important term to mention is a *decision stump*, which is a decision tree consisting of only one split with a root node and two leaves.

To determine where each split should occur, decision trees use a measure called purity. The goal when measuring the impurity of a node is to maximize classification accuracy. A natural choice for determining each split is to use the mis-classification

rate, however this measure is only based on the majority class in the node and does not take into account the distribution of the other classes. In terms of purity, a pure node is one where all observations belong to the same class, so classification accuracy is 100%. A node is considered 100% impure if there is an even 50/50 split and any impurity reduces classification accuracy. Thus we want to minimize impurity and it is based on the distribution of the classes within the nodes resulting from a split. In practice, to determine where the optimal split occurs, one would need to compute the purity of nodes for all possible splits. To compute the purity of a node, two of the most popular measures of impurity used are *entropy* and *Gini impurity index*. Gini impurity index and entropy both contain values between the interval range [0,1] and the optimum split occurs where the measure of impurity is minimized in both cases. The following formulas are used to compute the impurity of a given node, where $Q(p)$ represents the measure of impurity using entropy and Gini impurity index respectively,

$$Q(p) = -\sum_{i=1}^{k} p_i \cdot \log_2 p_i$$

$$Q(p) = 1 - \sum_{i=1}^{k} p_i^2$$

where p_i is the probability estimated from the data that an observation is of class i in the given node, and k is the number of classes (for our case k = 2). We will be focusing specifically on using entropy as our measure of impurity for the purpose of this project. We will use the entropy formula on the following sample data to determine the best possible split,

Table 1: Sample Dataset

Class	X	Class	X
1	1	-1	5
1	2	-1	6
-1	3	-1	7
1	4	1	8

For the purpose of this example we will look at two possible splits, however as mentioned, in practice one should consider all possible splits.The first split we will calculate the entropy for is the split with the first four observations (X = 1,2,3,4) in one node and the last four (X = 5,6,7,8) in the other node. For this split, there are 3 observations from one class and 1 observation from the second class in each of

the nodes. Using the entropy formula we have,

$$\begin{aligned} Q_1(p) &= -\sum_{i=1}^{k} p_i \cdot \log_2 p_i \\ &= -\left(\left(\frac{3}{4}\right) \cdot \log_2\left(\frac{3}{4}\right) + \left(\frac{1}{4}\right) \cdot \log_2\left(\frac{1}{4}\right)\right) \\ &= -\left(-0.31125 - \frac{1}{2}\right) \\ &= 0.8113 \end{aligned}$$

Similarly, we can obtain $Q_2(p) = 0.8113$ since the distribution of the classes is the same in the two nodes. Taking the average across the nodes, we have an entropy of 0.8113 for our first potential split.

Next, we will look at splitting the first five observations in one node (X = 1,2,3,4,5) and the remaining three observations (X = 6,7,8) in the other node. For this split, there are 3 observations from one class and 2 observation from the second class in the first node. The second node contains the remaining observations - 1 belonging to the first class and 2 from the second class. Using the same formula as above,

$$\begin{aligned} Q_1(p) &= -\left(\left(\frac{3}{5}\right) \cdot \log_2\left(\frac{3}{5}\right) + \left(\frac{2}{5}\right) \cdot \log_2\left(\frac{2}{5}\right)\right) \\ &= -(-0.4422 - 0.5288) \\ &= 0.971 \end{aligned}$$

In this case, the distribution of the classes is not the same in the two nodes so we will re-calculate the entropy for the second node the same way,

$$\begin{aligned} Q_2(p) &= -\left(\left(\frac{1}{3}\right) \cdot \log_2\left(\frac{1}{3}\right) + \left(\frac{2}{3}\right) \cdot \log_2\left(\frac{2}{3}\right)\right) \\ &= -(-0.5283 - 0.39) \\ &= 0.9183 \end{aligned}$$

Taking the average across the nodes, we have an entropy of 0.9447 for our second potential split. When looking at these two splits only, the first split is optimal since its entropy is the minimum of the two.

In the above example we proceeded with each observation having an equal weight of $\frac{1}{8}$, but we can also find an optimal split when observations are weighted unequally. This will be a critical component later on in this project when we begin to implement boosting algorithms. We will use our previous example and the first potential split to demonstrate how adjusting the distribution of weights can affect the entropy and purity of a node. Previously, each of our two classes accounted for 50% of the total weight. We will update the weights such that the four observations from the first class account for 80% of the total weight and the four observations from the second class account for 20% of the total weight. Table 2 shows the distribution of weights for a single observation in each class - both the original and adjusted weight.

This time we will use the class weights and the sum of the weights to calculate the entropy for each node as follows,

Table 2: Updated Weights

Class	Original Weight	Updated Weight
1	12.5%	20%
-1	12.5%	5%

$$\begin{aligned}
Q_1(p) &= -\sum_{k=1}^{K}\left(\frac{\sum_{i=1}^{n} w_{ik}}{\sum_{i=1}^{N} w_i}\cdot\log_2\frac{\sum_{i=1}^{n} w_{ik}}{\sum_{i=1}^{N} w_i}\right)\\
&= -\left(\left(\frac{0.20+0.20+0.20}{0.65}\right)\cdot\log_2\left(\frac{0.20+0.20+0.20}{0.65}\right)+\left(\frac{0.05}{0.65}\right)\cdot\log_2\left(\frac{0.05}{0.65}\right)\right)\\
&= -(-0.1066-0.2846)\\
&= 0.3912
\end{aligned}$$

Since we have unequal distributions of weights within the nodes for this case, we need to re-calculate the entropy for node 2 as well,

$$\begin{aligned}
Q_2(p) &= -\left(\left(\frac{0.20}{0.35}\right)\cdot\log_2\left(\frac{0.20}{0.35}\right)+\left(\frac{0.05+0.05+0.05}{0.35}\right)\cdot\log_2\left(\frac{0.05+0.05+0.05}{0.35}\right)\right)\\
&= -(-0.4614-0.5239)\\
&= 0.9853
\end{aligned}$$

Taking the weighted average across the nodes, we obtain a new entropy value of 0.6883 with our adjusted weights. When comparing this to our equally weighted example, we can see the entropy has decreased from 0.8113 to 0.6883. This shows why it is important to consider adjusting the weights of the classes as the purity of the node could be improved on. This was a simple example using only decision stumps with one split, and later in this project we will explore how combining such simple models can be used to produce a relatively effective final model.

1.2 Boosting

Boosting is a machine learning methods that works by combining a set of weak learners into a single strong learner. This is an example of a class of machine learning methods known as *ensemble methods*. Ensemble methods are a family of machine learning methods that use a number of base learners to create one final model. In the case of boosting, the base learners are what are referred to as *weak learners*. A weak learner is defined as a model with an accuracy slightly better than could be achieved by flipping a coin.

A model using M weak learners can be thought of as follows,

$$\hat{f}(x) = g\left(\hat{f_1}(x), \hat{f_2}(x), ..., \hat{f_M}(x)\right)$$

where the final model is a function of all the weak learners. This is how one would combine several weak learners to form a single strong learner and improve the effectiveness of the model. It is difficult to simultaneously fit M weak learners, so we use

boosting algorithms to do so sequentially. The meta model (final model consisting of a combination of the weak learners) of a boosting algorithm is the weighted sum of its weak learners,

$$\hat{f}(x) = \sum_{m=1}^{M} \hat{\alpha}_m \cdot \hat{f_m}(x)$$

When working with decision trees, we can use trees consisting of a single decision node and two prediction leaves, also know as a stump, since they are weak learners. In theory, boosting can be used to significantly reduce the error of any weak learning algorithm that can generate classifiers only slightly better than random guessing (Schapire, 1990; Freund, 1995; Freund & Schapire, 1996). By combining such weak learning algorithms to form a single strong learner, one can significantly improve the effectiveness of the model. Boosting can been applied to problems such as object detection, computer vision, or behaviour analysis (Appel et al., 2013).

1.3 AdaBoost

AdaBoost, short for Adaptive Boosting, is an example of such a boosting algorithm built off of decision trees and known as the first practical boosting algorithm, proposed by Fruend and Schapire in 1996 (Freund & Schapire, 1997). The general idea for the AdaBoost algorithm is to start with a single weak learner consisting of equal weights for each observation, and iteratively repeating this process until a certain stopping criteria is met. This stopping criteria could be a specified number of weak learners M or a threshold of the change in prediction error between iterations. For each iteration, the weights for observations that were incorrectly predicted are increased and the weights that were accurately predicted are decreased or "pushed down". Once the stopping criteria is met, the weak learners are combined all with their own weights (depending on the accuracy of each specific model) to form one final strong learner. In other words, the final classifier is a weighted average of all the weak classifiers. AdaBoost is the first algorithm to adjust adaptively to the errors of the hypotheses returned by the weak learning algorithm, hence where its name comes from. It is referred to as the "best out-of-the-box classifier", because it has the ability to create a strong final model with a relatively high accuracy using weak learners, with little to no intervention by the user.

The AdaBoost algorithm can be run as follows (Friedman et al., 2017):

1. Initialize the observation weights $w_i = \frac{1}{n}$, i = 1,2,...,n. Here w_i is equal to the weight of the ith observation and n is equal to the total number of observations in the training data set.

   For $m = 1$ to M, repeat steps 2-5:

2. Fit a classifier $f_m(x)$ to the training data using corresponding weights w_i.

3. Compute the weighted error of this new classifier as

$$err_m = \frac{\sum_{i=1}^{N} w_i \cdot I\left(y_i \neq f_m\left(x_i\right)\right)}{\sum_{i=1}^{N} w_i}$$

   where m denotes the mth weak learner, y_i represents the class of the ith observation, $f_m\left(x_i\right)$ represents the mth classifier on the ith observation and $I\left(y_i \neq f_m\left(x_i\right)\right)$ is an indicator function taking the value of 0 when a prediction is correct and 1 when a prediction is incorrect.

4. Compute

$$\alpha_m = log\left(\frac{1 - err_m}{err_m}\right)$$

   which corresponds to the weight given to $f_m\left(x\right)$ in producing the final classifier $f\left(x\right)$.

5. Update the individual weights of each of the observations for the next iteration by setting

$$w_i^{new} = w_i^{old} \cdot exp\left(\alpha_m \cdot I\left(y_i \neq f_m\left(x_i\right)\right)\right), i = 1, 2, ..., n.$$

6. Repeat the above Steps 2 - 5 for $m = 1$ to M or until a specified stopping criteria is reached. The stopping criteria can be either a specified number of weak learners or a threshold of the change in prediction error between iterations. Lastly, output

$$f\left(x\right) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m \cdot f_m\left(x\right)\right).$$

   The above outlines the AdaBoost algorithm when the base classifier $f_m\left(x\right)$ returns a discrete class label. If the base classifier instead returns a real-valued prediction (such as a probability mapped to the interval $[-1, 1]$), AdaBoost can be modified accordingly.

Throughout this paper we will explore modifications to this base AdaBoost algorithm to compare and determine strategies that might improve the accuracy of the model when working with a small sample size of data. We will then apply these modified algorithms to a small data set to analyze the effectiveness of them compared to the original AdaBoost model. First, gradient boosting will be introduced and discussed in the next chapter, as well as its relation to AdaBoost.

2 Literature Review

2.1 Forward Stagewise Additive Modelling

As discussed in the previous chapter, AdaBoost has the ability to increase the performance of even a very weak classifier - weak meaning the output is only slightly better than random guessing. The weak learners used in AdaBoost are decision stumps and the algorithm works by weighting the observations, giving more weight

to the cases incorrectly classified in the previous iteration and less to the cases that were classified correctly. New weak learners are added sequentially, increasing the weight each time on the cases which are more difficult to classify, until a specified stopping criteria is met. The final classifier is a weighted average of all the weak classifiers, where the weights depend on the accuracy of each model.

Boosting, in general, is an ensemble method that combines many weak learners to produce a strong final model. The final model of a boosting algorithm is the weighted sum of its weak learners. In other words, boosting is a way of fitting an additive expansion in a set of elementary 'basis' functions, where the basis functions are the individual weak classifiers $f_m(x) \in \{-1, 1\}$ (Friedman et al., 2017). The general form for an additive expansion is,

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m)$$

where in the case of boosting, $b(x; \gamma_m)$ is the m-th weak learner $f_m(x)$ outputting -1 or 1. The coefficients β_m are the corresponding weights associated to each weak learner $f_m(x)$ and the final output $f(x)$ is a weighted summation of all the learners. Ideally, additive models are fit by minimizing a loss function averaged over the training data,

$$\min_{\{\beta_m, \gamma_m\}_1^M} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b(x_i; \gamma_m)\right).$$

However, this method is not feasible due to the computational power required to solve this numerical optimization problem for most loss functions and/or basis functions. Often a simple alternative referred to as forward stagewise additive modelling (or forward stagewise boosting) can be used where instead we solve the subproblem of fitting just a single basis function,

$$\min_{\beta, \gamma} \sum_{i=1}^{N} L(y_i, \beta b(x_i; \gamma)).$$

This simplifies the above problem as now we only need to fit a single model at a time.

Forward stagewise modelling works by consecutively adding new basis functions to the expansion without adjusting the functions that have already been added. By using this approach, the model is able to approximately minimize the overall loss function averaged over the training data - without changing the parameters or coefficients of previously added basis functions for each new iteration.

The algorithm for forward stagewise boosting is outlined as follows (Friedman et al., 2017):

1. Initialize $f_0(x) = 0$
2. For $m = 1$ to M:
   (a) Compute

$$(\beta_m, \gamma_m) = \arg\min_{\beta, \gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$

(b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.

For each iteration m, the parameters β_m and γ_m are computed such that the updated model $f_m(x)$ achieves minimum loss on the training dataset. The basis function $b(x; \gamma_m)$ and corresponding coefficient β_m are added to the current expansion $f_{m-1}(x)$ to produce $f_m(x)$. This process is repeated M times and previously added terms are not modified.

Multiple different loss functions can be implemented when using the boosting method of forward stagewise additive modelling. Next, we will demonstrate how the AdaBoost algorithm previously presented is equivalent to the forward stagewise boosting algorithm when using an exponential loss function.

2.2 Relation between AdaBoost and Forward Stagewise Additive Modelling

We will prove that AdaBoost is equivalent to forward stagewise additive modelling when using the following exponential loss function,

$$L(y, f(x)) = \exp(-yf(x)).$$

From $2(a)$ of the forward stagewise boosting algorithm, we have the following parameters to calculate:

$$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$

In the case of AdaBoost, the basis functions are the individual classifiers $G_m(x)$ returning output -1 or 1. Using the exponential loss function and the formula above, we have:

$$\begin{aligned}(\beta_m, G_m) &= \arg\min_{\beta,G} \sum_{i=1}^{N} exp\left[-y_i\left(f_{m-1}(x_i) + \beta G(x_i)\right)\right] \\ &= \arg\min_{\beta,G} \sum_{i=1}^{N} w_i^{(m)} exp\left(-\beta y_i G(x_i)\right)\end{aligned}$$

with $w_i^{(m)} = \exp(-y_i f_{m-1}(x_i))$. At each iteration, we are solving for the classifier G_m and corresponding coefficient β_m. For any $\beta > 0$, G_m should minimize the weighted error rate. Otherwise, the loss could always be lower for the same value of β. The weight $w_i^{(m)}$ is dependent of $f_{m-1}(x_i)$, but independent of β and $G(x)$, and thus the values of the individual weights change with each new iteration. Since G_m should minimize the weighted error rate in order to achieve minimum loss, we have the following solution:

$$G_m = \arg\min_{G} \sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i)).$$

Now that we have solved for G_m, we can use this to solve for β_m:

$$\beta_m = \arg\min_{\beta} \sum_{i=1}^{N} w_i^{(m)} exp\left(-\beta y_i G_m\left(x_i\right)\right).$$

Note that when $y_i = G_m(x_i)$, the exponent will simplify to $-\beta$. Similarly, when $y_i \neq G_m(x_i)$, the exponent will simplify to β. Thus, we can express the formula presented above as:

$$\begin{aligned}\beta_m &= \arg\min_{\beta} \left(e^{-\beta} \cdot \sum_{y_i = G_m(x_i)} w_i^{(m)} + e^{\beta} \cdot \sum_{y_i \neq G_m(x_i)} w_i^{(m)} \right) \\ &= \arg\min_{\beta} \left[\left(e^{\beta} - e^{-\beta}\right) \cdot \sum_{i=1}^{N} w_i^{(m)} I\left(y_i \neq G\left(x_i\right)\right) + e^{-\beta} \cdot \sum_{i=1}^{N} w_i^{(m)} \right].\end{aligned}$$

The second equation is simply a different way to write the formula — when simplified this is equivalent to the first expression as the $e^{-\beta}$ terms will cancel out for the observations where $y_i \neq G_m(x_i)$. In order to find the β that will minimize the above formula, we will take the derivative with respect to β and set it equal to 0:

$$\begin{aligned}&\left(e^{\beta} + e^{-\beta}\right) \cdot \sum_{i=1}^{N} w_i^{(m)} I\left(y_i \neq G\left(x_i\right)\right) - e^{-\beta} \cdot \sum_{i=1}^{N} w_i^{(m)} = 0 \\ &\Rightarrow \left(e^{\beta} + e^{-\beta}\right) \cdot \sum_{i=1}^{N} w_i^{(m)} I\left(y_i \neq G\left(x_i\right)\right) = e^{-\beta} \cdot \sum_{i=1}^{N} w_i^{(m)} \\ &\Rightarrow \left(e^{2\beta} + 1\right) \cdot \sum_{i=1}^{N} w_i^{(m)} I\left(y_i \neq G\left(x_i\right)\right) = \sum_{i=1}^{N} w_i^{(m)} \\ &\Rightarrow e^{2\beta} = \frac{\sum_{i=1}^{N} w_i^{(m)}}{\sum_{i=1}^{N} w_i^{(m)} I\left(y_i \neq G\left(x_i\right)\right)} - 1.\end{aligned}$$

Letting $err_m = \frac{\sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i))}{\sum_{i=1}^{N} w_i^{(m)}}$, which is the same rated error rate as in AdaBoost, we have:

$$\begin{aligned}e^{2\beta} &= \frac{1}{err_m} - 1 \\ &= \frac{1 - err_m}{err_m}.\end{aligned}$$

Now, taking the log of both sides and solving for β to get the solution for β_m:

$$\begin{aligned}2\beta &= \log\left(\frac{1 - err_m}{err_m}\right) \\ \Rightarrow \beta &= \frac{1}{2} \log\left(\frac{1 - err_m}{err_m}\right).\end{aligned}$$

Thus, we now have the solution for β_m, the coefficient for learner G_m. Note that in the AdaBoost algorithm, α_m is of the same form - specifically $\alpha_m = 2\beta_m$. The

difference between the two solutions is a constant of 2. Updating the current model, we have:

$$f_m(x) = f_{m-1}(x) + \beta_m G_m(x)$$

with the weights for the next iteration as follows:

$$w_i^{(m+1)} = w_i^{(m)} \cdot \exp\left[-\beta_m y_i G_m(x_i)\right].$$

Now, we can use the fact that $\alpha_m = 2\beta_m$ and $-y_i G_m(x_i) = 2 \cdot I(y_i \neq G(x_i)) - 1$ and sub this into the above formula:

$$w_i^{(m+1)} = w_i^{(m)} \cdot \exp\left[\alpha_m I(y_i \neq G(x_i))\right] \cdot \exp\left[-\beta_m\right].$$

When comparing to the AdaBoost algorithm, we see that there is an extra term $\exp[-\beta_m]$. However, this term is applied to all samples and multiplies all weights by the same value, meaning it has no effect. Thus, this weight update is equal to the weight update method used in AdaBoost. We can conclude that the AdaBoost algorithm can be derived via a forward stagewise additive modelling approach when using an exponential loss function. Details of this proof can be found in Friedman et al. (2017).

As mentioned, forward stagewise additive modelling is a general framework used for boosting. We have demonstrated that when using an exponential loss function, we can derive the AdaBoost algorithm with this framework. Next, we will introduce gradient boosting, which is one way to solve forward stagewise boosting models using various different loss functions and considering the loss as a numerical optimization problem.

2.3 Gradient Boosting

The framework for gradient boosting originated from Breiman (1999) when he first referred to the idea as 'Arcing' algorithms, which is an acronym for Adaptive Reweighting and Combining. He observed that AdaBoost and related algorithms can be interpreted as an optimization algorithm which minimizes some loss function. Each step in an arcing algorithm consists of a weighted minimization followed by a recomputation of [the classifiers] and [weighted input] (Breiman, 1999). This framework was further developed by Friedman (1999) into gradient boosting machines, later referred to as gradient boosting, which are described as numerical optimization problems where the goal is to minimize the loss of a model by using a gradient descent like procedure to add weak learners at each step. For the purpose of this project, we will be focusing specifically on gradient tree boosting algorithms.

The gradient tree boosting algorithm can be viewed as a forward stagewise additive model as one new weak learner is added at a time in a given model and the existing weak learners are left unchanged. Gradient boosting is one specific way to solve forward stagewise boosting models, where the loss is viewed as a numerical optimization problem rather than considering the loss itself as a whole piece. In this case, each new added learner follows the gradient of the previous learner.

Below the generic gradient tree-boosting algorithm is shown, however different loss criteria $L(y, f(x))$ must be entered in order to obtain a specific algorithm. The

gradient tree-boosting algorithm, using regression trees as the base learners and as shown in Friedman et al. (2017), is run as follows:

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$
2. For $m = 1$ to M:
   (a) For $i = 1, 2, .., n$ compute
   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions $R_{jm}, j = 1, 2..., J_m$.
   (c) For $j = 1, 2, ..., J_m$ compute
   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma).$$
   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

For each iteration m, at line 2(a) the *pseudo* residuals r are computed, referring to the negative gradient based on the specific loss function used in the algorithm. At line 2(b), R_{jm} represents the jth region or node in the decision tree for the mth iteration and is computed using the pseudo residuals r in order to minimize these residuals and therefore improve model accuracy. Next, the coefficient γ_{jm} is computed using y for each region R_j at the mth iteration such that the updated model $f_m(x)$ achieves minimum loss on the training dataset. The current model is updated and this process is repeated M times where previously added terms are not modified. Lastly, the estimator $\hat{f}(x)$ is outputted as the final model computed in the Mth iteration.

2.4 Variants of AdaBoost

Now that we have generalized AdaBoost and introduced gradient boosting in order to handle a variety of loss functions, we will look at multiple different variants to the original AdaBoost algorithm. Note that the standard AdaBoost algorithm presented in the previous chapter can also be referred to as *Discrete* AdaBoost, as the base classifier $G_m(x)$ returns a discrete class label. As noted earlier, the algorithm shown is used for binary data in the two-class classification setting, and the following variants are algorithms that can be used on binary data as well.

2.4.1 Real AdaBoost

The Real AdaBoost algorithm was first introduced by Schapire and Singer (1999) and it can be viewed as a generalization of (Discrete) AdaBoost. The algorithm for Real AdaBoost uses real-valued “confidence-rated” classifiers where the weak learners return class probability estimates rather than the $[-1, 1]$ of AdaBoost. This real

value can be seen as the probability that a given input belongs to a specific class, while also accounting for the current weight distribution of the dataset.

The Real AdaBoost algorithm is presented below (Friedman et al., 2000):

1. Initialize the observation weights $w_i = \frac{1}{n}$, $i = 1,2,...,n$. Here w_i is equal to the weight of the ith observation and n is equal to the total number of observations in the training data set.

2. For $m = 1$ to M:

   (a) Fit the classifier to obtain a class probability estimate $p_m(x) = \hat{P}_w(y = 1 \mid x) \in [0, 1]$, using weights w_i on the training data.

   (b) Set $f_m(x) = \frac{1}{2} \log \frac{p_m(x)}{(1-p_m(x))} \in \mathbb{R}$.

   (c) Update the weights: $w_i = w_i \exp[-y_i f_m(x_i)]$ for $i = 1, 2, ..., N$, and renormalize so that $\sum_i w_i = 1$.

3. Output the classifier,

$$f(x) = \text{sign}\left(\sum_{m=1}^{M} f_m(x)\right).$$

When comparing the two, we can see the main differences between Real AdaBoost and AdaBoost are in lines 2(a) and 2(b). In the Real AdaBoost algorithm, these steps consist of computing and assessing the probability, or degree of confidence, that each data sample belongs to the predicted class. The standard AdaBoost algorithm classifies the data samples and computes the weighted error rate (Ferreira & Figueiredo, 2012).
Another difference is in Freund and Schapire's (1997) standard AdaBoost algorithm, weak models are restricted to the range $[-1, +1]$ whereas in Real AdaBoost, the weak models have range over $\mathbb{R}$. This is demonstrated in step 2(b) where the original range $[-1, +1]$ is converted to $\mathbb{R}$ by taking half of the the log of the odds ratio of the probability found in 2(a). If the probability is greater than 0.5, the output will be positive, and if less than 0.5, the output will be negative. The final classifier is based on the sum of these positive or negative values — if the total sum is positive, this corresponds to the $+1$ class, and if the total sum is negative, this corresponds to the -1 class. This confidence rating method could be useful as the harder to classify cases will have less of an effect on the final classifier. For example, a class probability estimate of 0.51 will provide a just slightly positive $f_m(x)$ value, thus having little effect on the final sum. A highly probable estimate of 0.99 will have a relatively extreme positive value, emphasizing the degree of confidence in the data belonging to that class.

2.4.2 Gentle AdaBoost

The Gentle AdaBoost algorithm can be viewed as a modified and improved version of Real AdaBoost by using adaptive Newton steps, rather than exact optimization, providing a more reliable ensemble by putting less emphasis on outliers. The second derivative of the expected log-likelihood is used to compute the update, which could

provide a faster convergence than the optimization method used for the updates in the previous two algorithms. The algorithm is considered *gentle* because it is viewed as more stable than the Real AdaBoost algorithm (Ferreira & Figueiredo, 2012). The main difference between Gentle AdaBoost and the Real AdaBoost is how the estimates of the weighted class probabilities are used to perform each update. Note that a regression decision tree stump is used for this variant, as a regression model is required for the weak learners.

The Gentle AdaBoost algorithm is shown below (Friedman et al., 2000):

1. Initialize the observation weights $w_i = \frac{1}{N}$, $i = 1,2,...,N$. Here w_i is equal to the weight of the ith observation and N is equal to the total number of observations in the training data set. Start with $f(x) = 0$.
2. For $m = 1$ to M:
   (a) Fit the regression function $f_m(x)$ by weighted least-squares of y_i to x_i using weights w_i.
   (b) Update $f(x) = f(x) + f_m(x)$.
   (c) Update the weights: $w_i = w_i \exp[-y_i f_m(x_i)]$ for $i = 1, 2, ..., N$, and renormalize so that $\sum_i w_i = 1$.
3. Output the classifier

$$f(x) = \text{sign}\left(\sum_{m=1}^{M} f_m(x)\right).$$

The updates in line 2(b) of the algorithm above have a strong similarity to those in the Logit Boost algorithm, since both algorithms use adaptive Newton steps.

2.4.3 Logit Boost

The Logit Boost variant uses adaptive Newton steps to fit an additive logistic regression model by directly optimizing the Bernoulli log-likelihood in a stagewise manner. Instead of minimizing the exponential loss as in AdaBoost, Logit Boost minimizes the logistic loss. Again, a regression model is required for the learners and a regression tree stump is used.

Focusing on the binary case, y^* represents the response and the probability that $y^* = 1$ is represented by $p(x)$. Here,

$$p(x) = \frac{e^{f(x)}}{e^{f(x)} + e^{-f(x)}}.$$

Using this formula, the Logit Boost algorithm runs as follows (Friedman et al., 2000):

1. Initialize the observation weights $w_i = \frac{1}{N}$, $i = 1,2,...,N$. Here w_i is equal to the weight of the ith observation and N is equal to the total number of observations in the training data set. Start with $f(x) = 0$ and probability estimates $p(x_i) = \frac{1}{2}$.

2. For $m = 1$ to M:

   (a) For $i = 1, 2, .., N$ compute the working response and weights

$$z_i = \frac{y_i^* - p(x_i)}{p(x_i)(1 - p(x_i))},$$

$$w_i = p(x_i)(1 - p(x_i)).$$

   (b) Fit the function $f_m(x)$ to the regression tree stump by a weighted least squares regression of z_i to x_i using weights w_i.

   (c) Update $f(x) = f(x) + \frac{1}{2} f_m(x)$ and $p(x) = \frac{e^{f(x)}}{e^{f(x)} + e^{-f(x)}}$.

3. Output the classifier,

$$f(x) = \text{sign}\left(\sum_{m=1}^{M} f_m(x)\right).$$

3 Simulation Study

A simulation study was conducted in order to test the algorithms present in the context of low sample sizes.

3.1 Data Generation Process

The five input variables of the model, the $X_i's$, are simulated independently and identically from the standard normal distribution, as many real world scenarios follow this distribution. Each initial data set is simulated with 1000 observations. The target variable is created using a multiple logistic regression formula, as follows,

$$\pi(x) = \frac{e^{X \cdot \beta}}{1 + e^{X \cdot \beta}}.$$

$\pi(x)$ represents the probability that the outcome is 1 out of the two possibilities 1 or -1. A uniform random number is compared to the probability - if it is less than or equal to the probability, the target variable $y = 1$ and if it is greater than the probability then $y = -1$. For simplicity, β_0 was set to zero but this parameter can be adjusted. Similarly, the remaining β values will be set to 1 to start but may be adjusted.

This paper explores many different variations to the base simulated data set in order to analyze how the algorithms perform in each case.

3.1.1 Number of Observations

The first variation will investigate how the number of observations or sample size of the data set affects the performance of the models. Data sets are simulated with 50, 100, 250, 500, and 750 observations, and compared with the base data set consisting of 1000 observations, in order to explore the impact that a smaller sample size has on the results.

3.1.2 Number of Classes

Another variation that will be explored is the number of classes and how extending our algorithms past binary classification will affect the outcomes and performance of the models. By looking at data sets simulated with 3 possible classes, it will be interesting to see if we have a different model outperforming the rest than the optimal model found for binary classification.

For the multi class linear data, logistic regression method will be used again to calculate the class probabilities. For non-binary data, these probabilities are modelled as

$$\pi\left(y=k \mid x\right)=\frac{e^{X\cdot\beta_k}}{1+\sum_{k=1}^{K-1}e^{X\cdot\beta_k}},$$

for $k=1...K-1$, and

$$\pi\left(y=K \mid x\right)=\frac{1}{1+\sum_{k=1}^{K-1}e^{X\cdot\beta_k}}.$$

Again, a uniform random number is compared to the probabilities to determine if the target variable y is equal to 1, 2, or 3. Here, $K=3$ classes.

3.1.3 Nonlinear Data

The last variation to look at is simulated data that follows a nonlinear link function, rather than a linear model - logistic regression in this case. Our input vector X is still simulated such that it is independently and identically distributed from the normal distribution. The class outcome variable y is simulated by first using a decision tree split to separate the data into 8 different nodes. For each of the nodes, a different link function is used with different probabilities of the outcome being equal to 1.

In order to simulate data with a nonlinear link, the data will be split into 4 different parts following a decision tree structure. The first split is observations where $X_1 <= 0.99$, the second split is based on if $X_2+X_3>0.6$, the third split is observations where $X_0<X_1$, and the last split is the remaining observations. Once the data is split into these four sections, the probabilities that $y=1$ are simulated using a different variation to the logistic regression formula (used for the linear method) for each section. These variations include adjusting the β parameters, squaring or cubing some of the X variables, and/or adding in correlation between the independent variables. The last step is comparing a uniform random number to $\pi\left(x\right)$ to determine if $y=1$ or $y=-1$.

To simulate multi class nonlinear data, the method used to simulate binary nonlinear data is combined with the method used to simulate multi class linear data. The data will follow the same decision tree structure as the binary nonlinear in order to be split into four different sections or nodes. Once split, the probabilities are modelled similar to the non-binary logistic regression method described in section 3.1.2. Again variations are added that range from adjusting the β parameters, adding exponents to some of the X variables, and/or adding in correlation between

the independent variables. Lastly, a uniform random number is compared to these probabilities to determine if the target variable y is equal to 1, 2, or 3 where $K = 3$.

3.2 Results

A total of 50 data sets were simulated for each combination of sample size and algorithm. The optimal M, or number of weak learners, was selected for each model based on the highest 5-fold cross validated accuracy when testing values from 5 to 60. The full data sets were then re-simulated 50 times using this optimal M value. The results shown in the figures throughout this section are the average or expected accuracy of the 50 simulated data sets of full sample size.

3.2.1 Binary Linear Model

We can see that both the Real AdaBoost and Logit Boost algorithms outperform the AdaBoost algorithm for all sample sizes. However, the Gentle AdaBoost algorithm stands out as having a much higher accuracy than the other three algorithms throughout. The standard deviation of the accuracy is shown by the dotted lines, and decreases as the sample size increases for both algorithms.

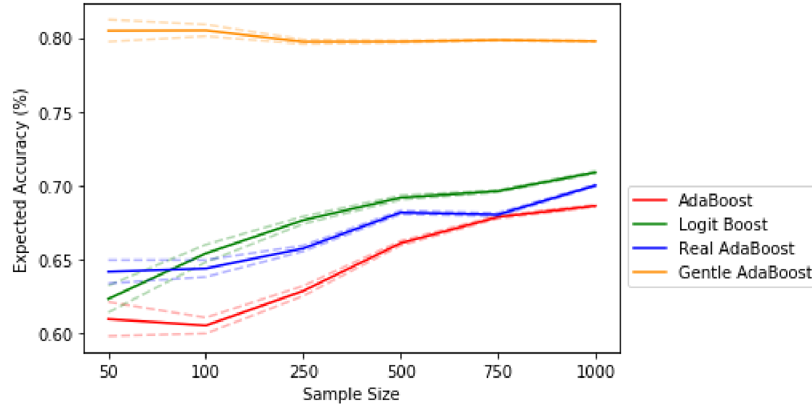

Figure 1: Comparison of the expected accuracy for different sample sizes across 50 iterations, using the four algorithms and binary, linear data.

As the sample size increases, the difference between the accuracy of the algorithms shrinks slightly.

3.2.2 Multi Class Linear Model

From Figure 2 it is clear that Gentle AdaBoost still outperforms the remaining algorithms, however the difference is less.

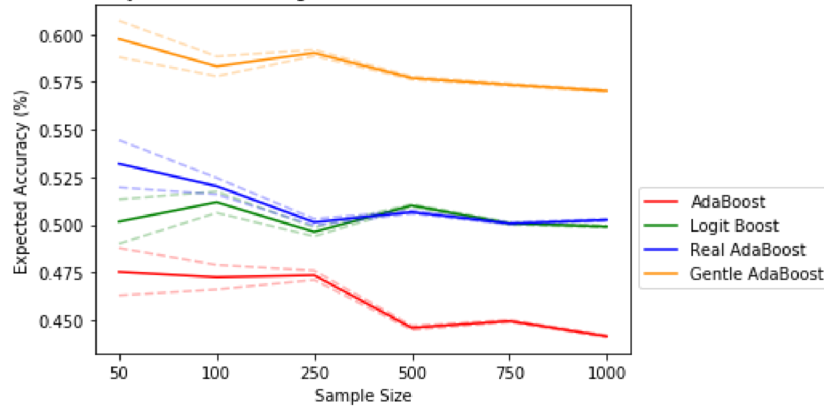

Figure 2: Comparison of the expected accuracy for different sample sizes across 50 iterations, using the four algorithms and multi class, linear data.

Using 3 possible classes, the above figure shows that for AdaBoost, Real AdaBoost, and Gentle AdaBoost, higher expected accuracies occurs at the smaller sample sizes. For Logit Boost, the highest average accuracy occurs at a sample size of 500.

3.2.3 Binary Nonlinear Model

Comparing Figure 3 with Figure 1, it is evident that Gentle AdaBoost is much closer in accuracy to the other three algorithms when using a nonlinear data simulation.

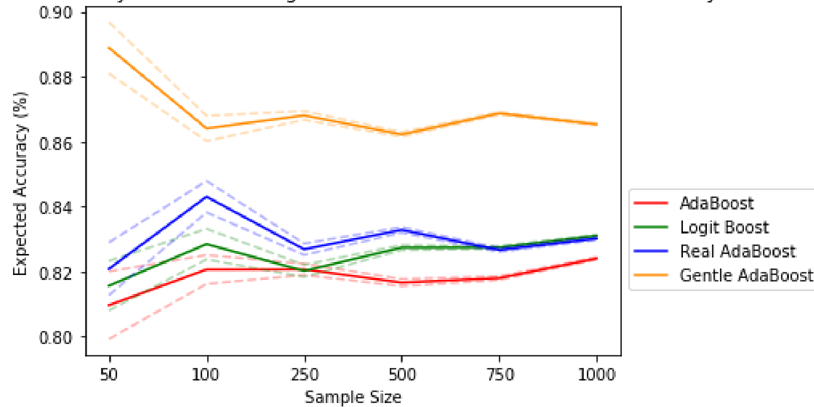

Figure 3: Comparison of the expected accuracy for different sample sizes across 50 iterations, using the four algorithms and binary, nonlinear data.

We can see that Real AdaBoost, and Logit Boost perform much better when using nonlinear data vs linear data. The expected accuracies for the models are above 80% among all sample sizes, whereas the accuracies for the linear model were much lower, ranging from 65% to 72%. Alternatively, Gentle AdaBoost only increases its accuracy slightly when using nonlinear vs linear data.

3.2.4 Multi Class Nonlinear Model

Extending the nonlinear model to multi class does not provide different results or model performance, much like when we extended the linear model to multi class.

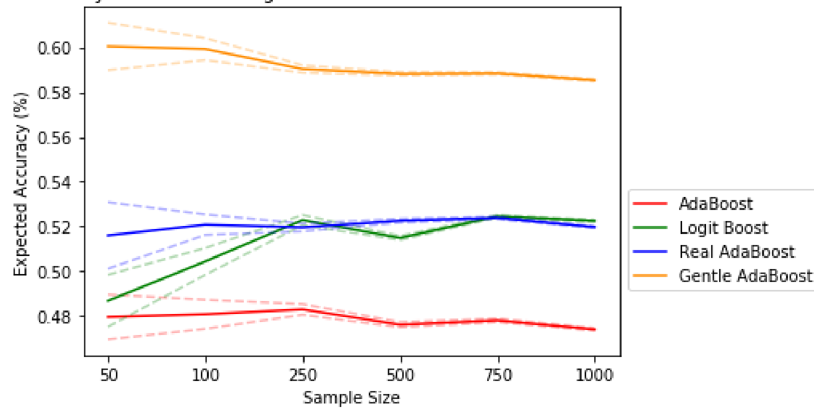

Figure 4: Comparison of the expected accuracy for different sample sizes across 50 iterations, using the four algorithms and multi class, nonlinear data.

Figure 4 shows that Gentle AdaBoost still has the highest prediction accuracy compared to the other three models. When using the nonlinear models, the other three algorithms are closer in accuracy to the Gentle AdaBoost algorithm, suggesting that if we were to further increase the nonlinearity of the data, Real AdaBoost or Logit Boost could potentially surpass Gentle AdaBoost in accuracy.

3.3 Logistic Regression vs. Boosting Models

Next we will compare the four boosting algorithms to a logistic regression model. For the linear data generation methods, this is the true model of the generated data, so it can be expected that the logistic regression model will outperform the boosting models. We will also investigate how the models compare when using the nonlinear data generation methods.

Comparing the logistic regression model below to the four boosting models, the only boosting model that outperforms it using binary linear data is the Gentle AdaBoost model. Specifically at the smaller sample sizes of 50 and 100, the Gentle AdaBoost algorithm performs much better.

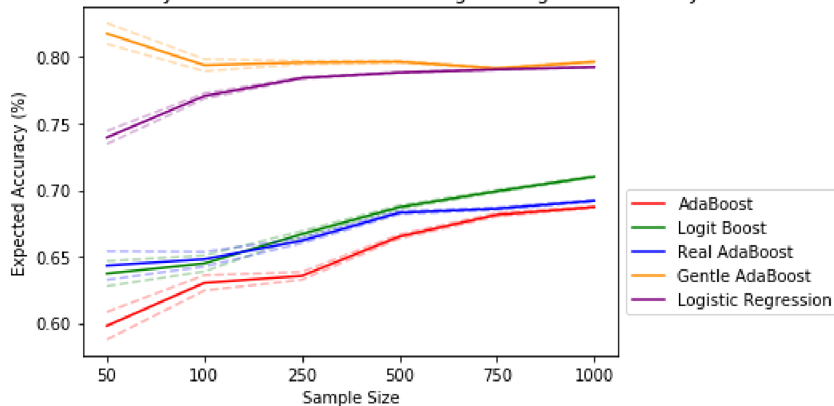

Figure 5: Plot of the expected accuracy for different sample sizes across 50 iterations, using a logistic regression model and the four AdaBoost algorithms with binary, linear data.

Looking at the smaller sample sizes of the logistic regression model below, using binary, nonlinear data, there is an accuracy of 81.13% with a sample size of 50 and 82.37% with a sample size of 100.

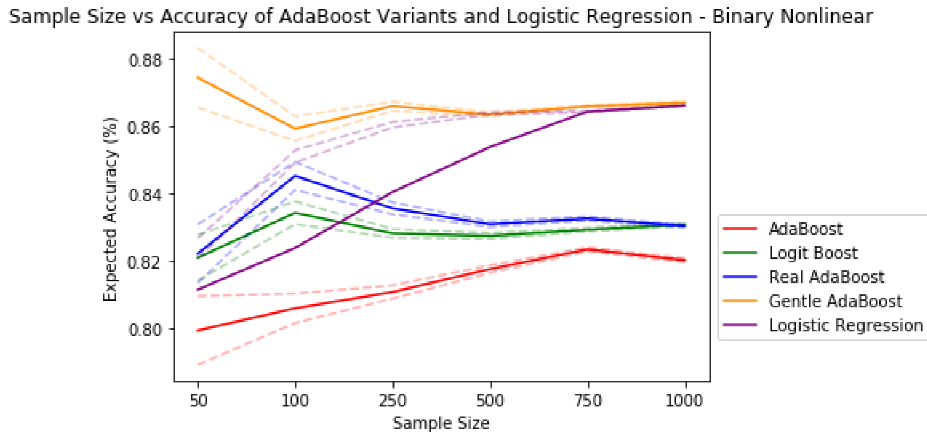

Figure 6: Plot of the expected accuracy for different sample sizes across 50 iterations, using a logistic regression model and the four AdaBoost algorithms with binary, nonlinear data.

Comparing this to the Real AdaBoost, Logit Boost, and Gentle AdaBoost models, which have an accuracy of 82.20%, 82.08%, 87.44% respectively at a sample size of 50, the logistic regression model performance is slightly worse than the Real AdaBoost and Logit Boost models with a more significant difference to the Gentle AdaBoost model. At a sample size of 100, Real AdaBoost has an accuracy of 84.02% and Logit Boost has an accuracy of 83.42%, which both have a better performance than the logistic regression model again — something we did not see with the linear data.

4 Application

The boosting methods explored in this paper will now be applied to real-world data in order to determine if they could be useful in predicting a categorical outcome. The context of this problem is classifying the subject condition in a sample of individuals based on which parts of the brain are activated while performing different tasks. We are interested in predicting y, where y is a binary categorical variable classified into the control vs patient group in this case. The control group is individuals with a brain classified as healthy or normal, and the patient group is individuals who have been diagnosed with schizophrenia. There are 16 inputs in total we will be considering, which are based on 4 different frequency bands and 3 tasks for each, plus the variance across tasks. The values given in the data are a measure called the Small World Propensity (SWP) from the estimated function networks. We want to see if the data shows a link between patterns of neural interactions in the brain and classifying whether a brain is healthy or not. We will be using the group variable as our output (control or patient) to see if we can accurately predict it using the boosting algorithms discussed in this paper and the other variables in the data set as the inputs.

4.1 Data

The data set comes from an EEG test conducted for 81 individuals, each in either the control or patient group. An EEG test is a way of investigating the neurons working together in the brain by measuring the electric potentials at the scalp. The neurons working together will show up as oscillations in signals where the speed of the oscillations depends on the type of neurons. The oscillations and how in sync they are will determine when two brain regions are communicating and can be observed from the electric potentials at the scalp. Once the test is conducted, the oscillations are used to estimate a graph of connections among electrodes. SWP measures if the resulting graph has something called the 'small world' property, and if it does, this may be referred to as a small-world graph. This type of graph has two important characteristics — high clustering and short path lengths between nodes. Thus, the measure given in the data quantifies the small world-ness of the resulting graph for each specific frequency band and activity. These quantities are measured within four different frequency bands for resting, listening to music, and watching a screen to determine which parts of the brain an individual was using for each task. The variance between the tasks is also calculated for each frequency band.

As mentioned, we will use subject group as the output and we will use the 16 variables as the inputs. The inputs are taken from four frequency bands in the brain represented by δ, θ, α, and β. Roughly, the δ brainwaves correspond to a sleep state, the θ brainwaves correspond to a creative, meditative state (i.e. between α and δ waves), the α brainwaves correspond to a daydreaming, relaxed state, and the β brainwaves correspond to an active state. As well as this, each frequency band was measured while the individual was partaking in different activities to determine which area of the brain was activated for each activity.

There are 39 observations for the control group and 42 observations for the patient group - stratified k-fold cross validation will be used to keep this proportion of observations. As mentioned, the resting/auditory/visual measures for each frequency band represent a quantity called the small world propensity (SWP) of a graph, and the variance is the variance of the SWP across the three tasks.

4.2 Results

The accuracy shown is computed based on the average testing accuracy using 5-fold cross validation, where the optimal M is chosen based on the training data. Unlike our simulation study, the data is limited here and our average accuracy is based only on one data set.

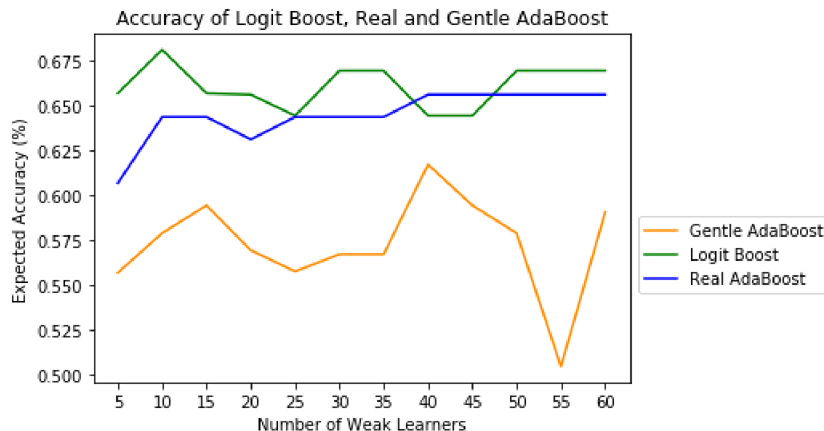

Figure 7: Comparison of the expected accuracy for different algorithms against the number of weak learners used. Accuracy is computed based on the average testing accuracy across 5 folds.

Based on Figure 7, when using the EGG data, Logit Boost and Real AdaBoost both outperform Gentle AdaBoost — a different result than what the results of the simulation study showed. The accuracies are also much lower overall, with the highest accuracy of Logit Boost being 67.5% with 10 weak learners.

The differences in our results here vs the simulation study could be explained by the differences in the data. This data set contains 16 input variables while our simulated data worked with only 5 inputs. Our data was also simulated using identically and independently normally distributed inputs, while it is unlikely our application data follows this same distribution. We can assume the inputs corresponding to the same frequency band are correlated with each other, specifically the variances. Lastly, although the nonlinearity of the data was increased from the original logistic regression model, it is unknown to what extent the nonlinearity was actually increased. Using a different method to simulate the data and provide more nonlinear links between the inputs and target variable could potentially affect the results, and may explain why our application does not follow the same conclusions.

5 Conclusions

This research explored three variants to the standard AdaBoost algorithm — Real AdaBoost, Gentle AdaBoost, and Logit Boost — and provided evidence to suggest that all three variations outperform the original model. Based on our simulation study, Gentle AdaBoost is the optimal algorithm and provides the highest prediction accuracy for normally distributed data, while our application suggests that Real AdaBoost and Logit Boost perform better in other circumstances. Further exploration into how these variants perform when increasing nonlinearity of the data or using a different distribution could be beneficial in determining more clearly which model works best for each situation.

6 References

Appel, R., Fuchs, T., Dollár, P., & Perona, P. (2013, May). Quickly boosting decision trees–pruning underachieving features early. In *conference on machine learning* (pp. 594-602). PMLR.

Breiman, L. (1999). Prediction games and arcing algorithms. *Neural computation*, *11*(7), 1493-1517.

Ferreira, A. J., & Figueiredo, M. A. (2012). Boosting algorithms: A review of methods, theory, and applications. *Ensemble machine learning*, 35-85.

Freund, Y. (1995). Boosting a weak learning algorithm by majority. *Information and computation*, *121*(2), 256-285.

Freund, Y., & Schapire, R. E. (1996, July). Experiments with a new boosting algorithm. In *icml* (Vol. 96, pp. 148-156).

Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, *55*(1), 119-139.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 1189-1232.

Friedman, J., Hastie, T., & Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, *28*(2), 337-407.

Friedman, J.H., Hastie, T., & Tisbshirani, R. (2017). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer.

Schapire, R. E. (1990). The strength of weak learnability. *Machine learning*, *5*(2), 197-227.

Shahraki, A., Abbasi, M., & Haugen, Ø. (2020). Boosting algorithms for network intrusion detection: A comparative evaluation of Real AdaBoost, Gentle AdaBoost and Modest AdaBoost. *Engineering Applications of Artificial Intelligence*, *94*, 103770.

7 Appendix

Import necessary packages

```
import pandas as pd
from numpy.random import seed
from numpy.random import normal
from numpy import mean
from numpy import std
import matplotlib.pyplot as plt
import numpy as np
import math

from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold

from sklearn import tree
```

Code to simulate data

```
#binary linear data simulation

def simulate_data(n, beta0):
    #Inputs:
    #n: number of observations generated
    #beta0: logistic regression beta0 parameter
    #Ouputs:
    #X: nx5 matrix of X values or inputs
    #y: nx1 output y vector
    X = np.zeros((n, 5))
    for i in range (0, n):
        mu,sigma = 0, 1
        for j in range(0, 5):
            X[i, j] = np.random.normal(mu, sigma, 1)[0]
    y = [None] * n
    p = [None] * n
    for k in range(0,n):
        p[k] = math.exp(beta0 + X[k, 0] + X[k, 1] + X[k, 2] + X[k, 3] + X[k, 4]
                        )/(1 + math.exp(beta0
                        + X[k, 0] + X[k, 1] + X[k, 2] + X[k, 3] + X[k, 4]
                        ))
        num = np.random.uniform(0, 1, 1)
        if num <= p[k]:
            y[k] = 1
        elif num > p[k]:
```

```
            y[k] = -1
    return X, y

########################################################################
#binary non-linear data simulation
def simulate_data_nl(n, beta0):
    #Inputs:
    #n: number of observations generated
    #beta0: logistic regression beta0 parameter
    #Ouputs:
    #X: nx5 matrix of X values or inputs
    #y: nx1 output y vector
    X = np.zeros((n, 5))
    for i in range (0, n):
        mu,sigma = 0, 1
        for j in range(0, 5):
            X[i, j] = np.random.normal(mu, sigma, 1)[0]
    y = [None] * n
    p = [None] * n
    for k in range(0,n):
        num = np.random.uniform(0, 1, 1)

        if X[k, 1] <= 0.99:

            p[k] = math.exp(beta0 + 0.2*X[k, 0] + X[k, 1] + 6*X[k, 2] + X[k, 3] +
            (-0.7)*X[k, 4])/(1 + math.exp(beta0
            + 0.2*X[k, 0] + X[k, 1] + 6*X[k, 2] + X[k, 3] + (-0.7)*X[k, 4]))

            if num <= p[k]:
                y[k] = 1
            elif num > p[k]:
                y[k] = -1

        elif X[k, 2] + X[k, 3] > 0.6:

            p[k] = math.exp(beta0 + X[k, 0] + np.square(X[k, 1]) + X[k, 2] +
            0.8*np.square(X[k, 3] + X[k, 4]))/(1 + math.exp(beta0
            + X[k, 0] + np.square(X[k, 1]) + X[k, 2] + 0.8*np.square(X[k, 3]
            + X[k, 4])))

            if num <= p[k]:
                y[k] = 1
            elif num > p[k]:
                y[k] = -1

        elif X[k, 0]  < X[k, 1]:

            p[k] = math.exp(beta0 + np.power(X[k, 0] + X[k, 1], 3) + X[k, 2]
```

```
            + np.square(X[k, 3]) + X[k, 4])/(1 + math.exp(beta0
            + np.power(X[k, 0] + X[k, 1], 3) + X[k, 2] + np.square(X[k, 3])
            + X[k, 4]))

            if num <= p[k]:
                y[k] = 1
            elif num > p[k]:
                y[k] = -1

        else:

            p[k] = np.random.uniform(0, 1, 1)

            if num <= p[k]:
                y[k] = 1
            elif num > p[k]:
                y[k] = -1

    return X, y

########################################################################
#multiclass linear data simulation
#using for multi class data
def simulate_data_multi(n, beta0):
    #Inputs:
    #n: number of observations generated
    #beta0: logistic regression beta0 parameter
    #Ouputs:
    #X: nx5 matrix of X values or inputs
    #y: nx1 output y vector
    X = np.zeros((n, 5))
    for i in range (0, n):
        mu,sigma = 0, 1
        for j in range(0, 5):
            X[i, j] = np.random.normal(mu, sigma, 1)[0]
            #X[i, j] = np.random.poisson(3, 1)[0]
            #X[i, j] = np.random.binomial(1, 0.6, 1)[0]
    y = [None] * n
    p1 = [None] * n
    p2 = [None] * n
    p3 = [None] * n

    for k in range(0,n):
        p1[k] = math.exp(beta0 #+ X[k, 0] + X[k, 1] + X[k, 2] + X[k, 3] + X[k, 4]
                        )/(1 + 2*math.exp(beta0
                        + X[k, 0] + X[k, 1] + X[k, 2] + X[k, 3] + X[k, 4])
                        )
```

```
        p2[k] = math.exp(beta0 + X[k, 0] + X[k, 1] + X[k, 2] + X[k, 3] + X[k, 4]
                    )/(1 + 2*math.exp(beta0
                    + X[k, 0] + X[k, 1] + X[k, 2] + X[k, 3] + X[k, 4]
                    ))
        p3[k] = 1 - p1[k] - p2[k]

        #generate a uniform random number
        num = np.random.uniform(0, 1, 1)

        if num <= p1[k]:
            y[k] = 2
        elif num > p1[k] and num <= (p1[k] + p2[k]):
            y[k] = 1
        elif num > (p1[k] + p2[k]):
            y[k] = 0
    return X, y

################################################################################
#multiclass non linear data simulation
#using for multi class data
def simulate_multi_nl(n, beta0):
    #Inputs:
    #n: number of observations generated
    #beta0: logistic regression beta0 parameter
    #Ouputs:
    #X: nx5 matrix of X values or inputs
    #y: nx1 output y vector
    X = np.zeros((n, 5))
    for i in range (0, n):
        mu,sigma = 0, 1
        for j in range(0, 5):
            X[i, j] = np.random.normal(mu, sigma, 1)[0]
            #X[i, j] = np.random.poisson(3, 1)[0]
            #X[i, j] = np.random.binomial(1, 0.6, 1)[0]
    y = [None] * n
    p1 = [None] * n
    p2 = [None] * n
    p3 = [None] * n

    for k in range(0,n):
        num = np.random.uniform(0, 1, 1)

        if X[k, 1] <= 0.99:

            p1[k] = math.exp(beta0
                    )/(1 + 2*math.exp(beta0
                    + X[k, 0] + X[k, 1] + X[k, 2] + X[k, 3] + X[k, 4])
                    )
```

```
    p2[k] = math.exp(beta0 + X[k, 0] + X[k, 1] + X[k, 2] + X[k, 3] +
    X[k, 4])/(1 + 2*math.exp(beta0
    + X[k, 0] + X[k, 1] + X[k, 2] + X[k, 3] + X[k, 4]
                ))
    p3[k] = 1 - p1[k] - p2[k]

    if num <= p1[k]:
        y[k] = 2
    elif num > p1[k] and num <= (p1[k] + p2[k]):
        y[k] = 1
    elif num > (p1[k] + p2[k]):
        y[k] = 0

elif X[k, 2] + X[k, 3] > 0.6:

    p1[k] = math.exp(beta0
                )/(1 + math.exp(beta0
                + X[k, 0] + X[k, 1] + X[k, 2] + X[k, 3] + X[k, 4])
                + math.exp(beta0 + X[k, 0] + np.square(X[k, 1]) + X[k, 2]
                + 0.8*np.square(X[k, 3] + X[k, 4])
                ))

    p2[k] = math.exp(beta0 + X[k, 0] + np.square(X[k, 1]) + X[k, 2] +
        0.8*np.square(X[k, 3] + X[k, 4]))/(1 + math.exp(beta0
        + X[k, 0] + X[k, 1] + X[k, 2] + X[k, 3] + X[k, 4])
        + math.exp(beta0 + X[k, 0] + np.square(X[k, 1])
        + X[k, 2] + 0.8*np.square(X[k, 3] + X[k, 4])
                ))

    p3[k] = 1 - p1[k] - p2[k]

    if num <= p1[k]:
        y[k] = 2
    elif num > p1[k] and num <= (p1[k] + p2[k]):
        y[k] = 1
    elif num > (p1[k] + p2[k]):
        y[k] = 0

elif X[k, 0]  < X[k, 1]:

    p1[k] = math.exp(beta0 + np.power(X[k, 0] + X[k, 1], 3) + X[k, 2] +
    np.square(X[k, 3]) + X[k, 4]
    )/(1 + math.exp(beta0
    + np.power(X[k, 0] + X[k, 1], 3) + X[k, 2] + np.square(X[k, 3]) +
    X[k, 4]) + math.exp(beta0 + 0.2*X[k, 0] + X[k, 1] +
    (-1.3)*np.square(X[k, 2]) + X[k, 3] + X[k, 4])
                )
```

```
            p2[k] = math.exp(beta0 + 0.2*X[k, 0] + X[k, 1] +
                (-1.3)*np.square(X[k, 2]) + X[k, 3] + X[k, 4]
                )/(1 + math.exp(beta0
                + np.power(X[k, 0] + X[k, 1], 3) + X[k, 2] + np.square(X[k, 3])
                + X[k, 4]) + math.exp(beta0 + 0.2*X[k, 0] + X[k, 1] +
                (-1.3)*np.square(X[k, 2]) + X[k, 3] + X[k, 4])
                )

            p3[k] = 1 - p1[k] - p2[k]

            if num <= p1[k]:
                y[k] = 0
            elif num > p1[k] and num <= (p1[k] + p2[k]):
                y[k] = 1
            elif num > (p1[k] + p2[k]):
                y[k] = 2

        else:
            p1[k] = math.exp(beta0)/
            (1 + math.exp(beta0 + (-2)*X[k, 0] + X[k, 1] + 0.8*X[k, 2] + X[k, 3] +
            np.square(X[k, 4])) + math.exp(beta0 + X[k, 0] + np.power(X[k, 1], 3)
            + X[k, 2] + (-0.1)*np.square(X[k, 3] + X[k, 4])
                 ))

            p2[k] = math.exp(beta0 + (-2)*X[k, 0] + X[k, 1] + 0.8*X[k, 2] + X[k, 3]
            + np.square(X[k, 4]))/(1 + math.exp(beta0
            + (-2)*X[k, 0] + X[k, 1] + 0.8*X[k, 2] + X[k, 3] + np.square(X[k, 4]))
            + math.exp(beta0 + X[k, 0] + np.power(X[k, 1], 3) + X[k, 2] +
            (-0.1)*np.square(X[k, 3] + X[k, 4])
                        ))

            p3[k] = 1 - p1[k] - p2[k]

            if num <= p1[k]:
                y[k] = 2
            elif num > p1[k] and num <= (p1[k] + p2[k]):
                y[k] = 1
            elif num > (p1[k] + p2[k]):
                y[k] = 0

    return X, y
```

Functions to use for simulation

```
################################################################################

clf = DecisionTreeClassifier(criterion = "entropy", max_depth=1)
r1, r2 = 5, 60
```

```
def createList(r1, r2):
    return np.arange(r1, r2+5, 5)

# get a list of models to evaluate
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = (createList(r1,r2))
    #using one vs rest classification and adaboost
    for n1 in n_trees:
        models[str(n1)] = OneVsRestClassifier(AdaBoostClassifier(clf, n_estimators
        = n1, learning_rate = 1/n1, algorithm = 'SAMME'))
    return models

#same thing for Real AdaBoost
def real_models():
    models = dict()
    # define number of trees to consider
    n_trees = (createList(r1,r2))
    #using one vs rest classifcation and real adaboost
    for n1 in n_trees:
        models[str(n1)] = OneVsRestClassifier(AdaBoostClassifier(clf, n_estimators
        = n1, learning_rate = 1/n1, algorithm = 'SAMME.R'))
    return models

#same thing for Logit Boost
def logit_models():
    models = dict()
    # define number of trees to consider
    n_trees = (createList(r1,r2))
    #using one vs rest classification and logit boost
    for n1 in n_trees:
        models[str(n1)] = OneVsRestClassifier(LogitBoost(DecisionTreeRegressor
        (max_depth=1), n_estimators = n1, learning_rate = 1/n1))
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    #10 fold stratified k fold cross validation
    cv = StratifiedKFold(n_splits=10)
    # evaluate the model and testing accuracy using cross validation
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs = -1)
    return scores
```

Functions for Gentle AdaBoost

```
#Accuracy function
```

```
# Accuracy
def Accuracy(y, preds):
    # Inputs:
    # y: vector of datasets y-values
    # preds: model of interests predictions
    #
    # Outputs:
    # accuracy: accuracy of model
    # Initialize count of correct predictions and length of dataset
    count = 0
    n = len(y)
    # Determine number of correct predictions
    for obs in range(0, n):
        if y[obs] == preds[obs]:
            count = count + 1
        else:
            count = count
    # Divide correct predictions by number of observations
    accuracy = count/n
    return(accuracy)

########################################################
#Gentle AdaBoost algorithm
def GentleAdaBoost(X_train, y_train, X_test, y_test, n_trees = 50):
    # Inputs:
    # X-train: matrix of x-values in training set
    # y-train: vector of y-values in training set(must be binary classification)
    # X-train: matrix of x-values in testing set
    # y-train: vector of y-values in testing set(must be binary classification)
    # # n_trees - define number of trees or weak learners to consider
    #
    # Outputs:
    # final_f: final prediction of observation classes

    # Initialize dataset length and data point weights among other measures needed
    #for Gentle AdaBoost
    n_test = len(y_test)
    n_train = len(y_train)

    w = [1/n_train] * n_train

    preds_train = [None] * n_trees
    preds_test = [None] * n_trees
    new_w= [None] * n_train

    f = [0] * n_test

    # Run the AdaBoost algorithm 'n_trees' times (default = 50)
```

```
    for i in range(0, n_trees):
        #add an intercept point since default does not include
        X_train = sm.add_constant(X_train)
        X_test = sm.add_constant(X_test)
        # Fit weak learner (weighted least squares) and make predictions
        model = sm.WLS(y_train, X_train, w, hascosnt = True)

        model_fit = model.fit()

        #get predictions for training to update weights and testing for predicting
        #accuracy
        preds_train[i] = model_fit.predict(X_train)
        preds_test[i] = model_fit.predict(X_test)

        for k in range(0, n_train):
            new_w[k] = w[k] * math.exp(-y_train[k] * preds_train[i][k])


        for l in range(0, n_train):
            w[l] = new_w[l] / sum(new_w)

        for j in range(0, n_test):
            f[j] = f[j] + preds_test[i][j]

    ## comment out for multi class
    # Determine final class predictions based on sign of predictions
    final_f = [0] * n_test
    for b in range(0, n_test):
        if f[b] >= 0:
            final_f[b] = 1
        else:
            final_f[b] = -1

    return final_f

##########################################################
# For binary classification running 50 times

#Create StratifiedKFold split with 10 folds
skf = StratifiedKFold(n_splits=10, shuffle=True)

exp_accuracy_gentle = []
std_accuracy_gentle = []

for size in (50, 100, 250, 500, 750, 1000):#, 10000):
    total_accuracy = []
    n = size
    for it in range(0,50):
```

```
        X, y = simulate_data(n,beta0)
        results = list()
        #testing M values using k fold cross validation
        for n1 in n_trees:
            scores = list()
            for train_index, test_index in skf.split(X, y):

                x_train_fold, x_test_fold = X[train_index], X[test_index]
                y_train_fold, y_test_fold = np.array(y)[train_index],
                np.array(y)[test_index]

                # Fit weak learner and make predictions
                model = GentleAdaBoost(x_train_fold, y_train_fold, x_test_fold,
                y_test_fold, n1)

                accuracy = Accuracy(y_test_fold, model)
                scores.append(accuracy)

            results.append(mean(scores))
        total_accuracy.append(max(results))

    #take expected value and std of 50 iterations
    exp_accuracy_gentle.append(mean(total_accuracy))
    std_accuracy_gentle.append(std(total_accuracy)/math.sqrt(n))

########################################################
#For multi class classifiation running 50 iterations

exp_accuracy_gentle = []
std_accuracy_gentle = []

for size in (50, 100, 250, 500, 750, 1000):#, 10000):
    total_accuracy = []
    n = size
    for it in range(0,50):
        #X, y = simulate_data(n,beta0)
        X, y = simulate_multi_nl(n, beta0)

        results = list()
        #testing M values using k fold cross validation
        for n1 in n_trees:
            scores = list()
            for train_index, test_index in skf.split(X, y):

                x_train_fold, x_test_fold = X[train_index], X[test_index]
                y_train_fold, y_test_fold = np.array(y)[train_index],
                np.array(y)[test_index]
```

```
y_train_0 = [0] * y_train_fold
y_train_1 = [0] * y_train_fold
y_train_2 = [0] * y_train_fold
#y_train_3 = [0] * y_train_fold

for i in range(0, len(y_train_fold)):
    if y_train_fold[i] == 0:
        y_train_0[i] = 1
    elif y_train_fold[i] == 1:
        y_train_1[i] = 1
    elif y_train_fold[i] == 2:
        y_train_2[i] = 1
    # elif y_train_fold[i] == 3:
    #     y_train_3[i] = 1

y_test_0 = [0] * y_test_fold
y_test_1 = [0] * y_test_fold
y_test_2 = [0] * y_test_fold
#y_test_3 = [0] * y_test_fold

for i in range(0, len(y_test_fold)):
    if y_test_fold[i] == 0:
        y_test_0[i] = 1
    elif y_test_fold[i] == 1:
        y_test_1[i] = 1
    elif y_test_fold[i] == 2:
        y_test_2[i] = 1
    # elif y_test_fold[i] == 3:
    #     y_test_3[i] = 1

# Fit weak learner and make predictions
model0 = GentleAdaBoost(x_train_fold, y_train_0, x_test_fold,
y_test_0, n1)
model1 = GentleAdaBoost(x_train_fold, y_train_1, x_test_fold,
y_test_1, n1)
model2 = GentleAdaBoost(x_train_fold, y_train_2, x_test_fold,
y_test_2, n1)
#model3 = GentleAdaBoost(x_train_fold, y_train_3, x_test_fold,
y_test_3, n1)

model = [0] * y_test_fold

for num in range(0, len(y_test_fold)):
    if max(model0[num], model1[num], model2[num]) == model0[num]:
        model[num] = 0
    elif max(model0[num], model1[num], model2[num]) == model1[num]:
        model[num] = 1
    elif max(model0[num], model1[num], model2[num]) == model2[num]:
```

```
                model[num] = 2
            #     else:
            #         model[num] = 3
            accuracy = Accuracy(y_test_fold, model)
            scores.append(accuracy)

        results.append(mean(scores))
    total_accuracy.append(max(results))

#take expected value and std of 50 iterations
exp_accuracy_gentle.append(mean(total_accuracy))
std_accuracy_gentle.append(std(total_accuracy)/math.sqrt(n))
```

Run the Simulation 50 times

```
#change depending on algorithm (AdaBoost, Real AdaBoost, and Logit Boost)
exp_accuracy = []
std_accuracy = []

beta0 = 0

for size in (50, 100, 250, 500, 750, 1000):#, 10000):
    accuracy = []
    n = size #testing 50 sample size for LogitBoost
    for i in range(0, 50):

        #change depending if linear/ non linear and binary/ multiclass
        X, y = simulate_multi_nl(n,beta0)

        #using get_models for discrete AdaBoost, real_models for Real_AdaBoost,
        logit_models for Logit Boost
        models = get_models()

        # evaluate the models and store results
        results, names, means = list(), list(), list()

        for name, model in models.items():
            #accuracy
            scores = evaluate_model(model, X, y)
            results.append(scores)
            #number of weak learners
            names.append(name)
            #mean accuracy
            means.append(mean(scores))
       # print('>%s %.3f (%.3f)' % (name, max(means)))
        #taking optimal m
        accuracy.append(max(means))
```

```
#change depending on algorithm
 exp_accuracy.append(mean(accuracy))
 std_accuracy.append(std(accuracy)/math.sqrt(n))
```