Exploration of Bagging and Boosting Ensemble Methods through Random Forests and Adaboosting

Yousuf Baker
B.Sc Electrical Engineering, Boston University
ybaker@bu.edu

Andreas Oliveira
B.Sc Computer Engineering, Boston University
andoliv@bu.edu

Grayson Wiggins
B.Sc Electrical Engineering, Boston University
gdw15@bu.edu

December 13, 2022

1 Introduction

This paper will outline the exploration of ensemble bagging and boosting methods through the close study of Random Forests and Adaboosting. The goals of this exploration are to establish a theoretical motivation and intuition for the algorithms, highlight key aspects of their practical implementation, and highlight some of these key theoretical properties on both synthetic and real world datasets with interesting geometries.

1.1 Connection to Rest of Machine Learning

Bagging and Boosting methods generally fall into a class of algorithms called ensemble methods. At a high level, ensemble methods are a class of methods that come to a single, combined decision rule by aggregating a variety of decision rules in some way. The specific methods we are looking into are AdaBoosting (boosting) and Random Forests (bagging): AdaBoost sequentially combines its high bias weak learners into a strong learner; whereas Random forests trains and combines its individual high variance decision rules in parallel.

1.2 Challenges

The anticipated challenges of this exploration include the following:

1. Understanding the theoretical motivation of the methods well enough to motivate our intuition and enable their implementation

2. Implementing not only the Random Forests and AdaBoost algorithms themselves, but also the decision tree algorithm that both methods use as their learner

2 Literature Review

2.1 General Review and Decision Trees

The literature review for decision trees involved reading the decision trees section on "Understanding Machine Learning : From Theory to Algorithms" by Shai Shalev-Shwartz and Shai Ben-David [6], the CART procedure invented by Leo Breiman [1] and a set of Cornell Lectures by Kilian Weinberger that can be found the Youtube video platform and on his personal website. Shwartz's book and Breiman's paper served to provide a strong theoretical foundation of the methods, while the lectures were important an accessible introduction to the topic.

2.2 Boosting and AdaBoosting

For boosting, the literature reviewed includes two papers primarily: the first is Yoav Freund and Robert Schapire's "A Decision-Theoretic Generalization of On-line Learning and an Application to Boosting" [2], which gives a rigorous general overview of boosting, its motivations, and a few of the boosting methods. This acted as a reference text since it contained all the necessary mathematical formulations, proofs, and derivations for boosting in general, as well as for the AdaBoost algorithm. The second, and the main text used for this paper, was Freund and Schapire's much condensed "A short Intro to Boosting" [3], which highlights some of the key conclusions and mathematical formulations. Combining both these papers gives a much more complete picture of boosting and AdaBoosting than what would be possible by simply studying one or the other.

2.3 Bagging and Random Forests

The literature review for random forests included "Random Forests" by Breiman [1] and an additional in-depth review by Gilles Louppe entitled "Understanding Random Forests: From Theory to Practice" [4]. These two papers served to understand the deep theoretical properties of the Random Forests Algorithm. More specifically, Brieman shows two important properties in the Random Forests procedure: first the convergence of the ensemble generalization error and second the dependence of the generalization error on the variance of the ensemble. These were key findings that were explored and detailed in the theoretical section as well as the experimental section to demonstrate some empirical validation of the theory. The Louppe paper further characterizes the generalization error discussed by Breiman and highlights how it can be reduced through decreasing the correlation of decision trees. Therefore, the Louppe paper was also directly useful for the theory section of our paper, where we relate the correlation of decision trees to ensemble variance.

3 Problem Formulation and Solution Approaches

In this project we seek to explore Bagging and Boosting. Specifically, we will look into Random Forests and AdaBoosting, which try to minimize high variance and high bias problems respectively, on binary classification tasks. The goal is to try to explore how each algorithm deals with different dataset topologies. We also plan to explore tuning methods for each algorithm and outline the impact of adjusting each hyperparameter of the algorithm. For example, Random Forests specify that the decision trees should partition their dataset into a small set of features compared to the features in the original space (approximately $k = \sqrt{d}$). Thus one exploration is loosening that condition (meaning increasingly select larger number of features to partition the space of each decision tree) and observing what happens. Boosting methods are posed to be solving the problem of high bias error, and so a potential exploration is driving the depth of each decision tree to introduce higher variance into the constituent weak learners. The main metric for comparison will be CCR because we are experimenting these algorithms with binary classification tasks. After this exploration using synthetic datasets, we will apply the algorithms explored to the following dataset:

https://www.kaggle.com/sulianova/cardiovascular-

disease-dataset

which is a dataset with sufficiently large features as to allow us to demonstrate the effect some of the key tuning parameters in random forests.

To motivate our discussion of Random Forests and AdaBoosting, we will begin with a brief analysis of decision trees, and its associated issues.

3.1 Decision Trees Algorithm

The decision tree algorithm is a data structure that builds a classifier $h : \mathbb{X} \to Y$ by recursively constructing a tree that partitions a dataset based on a information gain criteria. Suppose that the original training set is comprised of

$$\mathbb{X} \supseteq \{(x_{11}, x_{12}, ..., x_{1d}), (x_{21}, x_{22}, ..., x_{2d}), (x_{31}, x_{32}, ..., x_{3d}), ..., (x_{n1}, ..., x_{nd})\}$$

(where each $x \in X$ is just a d-dimensional point inside $\mathbb{R}^d$)

$$\mathbb{P} \supseteq \{p_{x_1}, p_{x_2}, p_{x_3}, ..., p_{x_n}\}$$

(which is just the discrete probability distribution associated with the points in $\mathbb{X}$) and the labels

$$Y \supseteq \{y_1, ..., y_n\}$$

At the the first stage, the decision tree looks at the whole training set and determines how it can best "partition" the dataset so that the partitioned data contains a more "pure" group of people. Once found, it stores an object representing this partition into one of its attributes, calls the function that is making the first step on each of the partitioned data, and halts when the user specifies it doesn't want to partition more (for pseudocode refer to the appendix). "Partition" here means taking an attribute or single coordinate n from a point $p \in \mathbb{R}^d$ and determining whether a point $x \in \mathbb{R}^d$ falls in the left partition $\mathbb{X}_1$ if $p[n] \leq x[n]$ (where indexing is used to fetch the nth coordinate) and in the right partition $\mathbb{X}_2$ otherwise. Note that since our training set will always be discrete, all possible partitions of a set are satisfied by greedily partitioning on every dimension of every point in $\mathbb{X}$. Intuitively, "pure" just means that we have more of a single class representing the partition when compared to the previous unpartitioned data. Mathematically, the purity of a partition is given by the Gini Impurity score which is the following:

$$g(p_1, ..., p_k) = 1 - \sum_{i=1}^{k} p_i^2 \tag{1}$$

where p_i is the probability of class i inside the partition, or precisely:

$$p_i = \sum_{j=1}^{n} I(y_j = i) p_{x_j} \tag{2}$$

To get some intuition of what this function looks like, please refer to the Appendix. Formula (1) represents only the impurity score of a single partition stemming from a larger dataset. To get the overall impurity score of the whole dataset that went through the parition process, let:

$$G : \mathbb{R} \times \mathbb{Z}_d \to \mathbb{R} \tag{3}$$

be the function that takes a point $t \in \mathbb{R}^d$ and a integer $z \in \mathbb{Z}_d$ where $\mathbb{Z}_d = \{0, 1, ..., d\}$ (where d is the number of dimensions for a single point x) and outputs the overall impurity score given by the combination $[t, z]$. To refer to the precise mathematical definition of G, refer to the Appendix. To select the best partition we get the lowest overall impurity score or in precise mathematical notation:

$$\{t_{optimal}, z_{optimal}\} = argmin_{t \in \mathbb{X}, z \in \mathbb{Z}_d} G(t, z) \tag{4}$$

Once we have $\{t_{optimal}, z_{optimal}\}$, we divide $\mathbb{X}$, $\mathbb{P}$, $\mathbb{Y}$ onto $\mathbb{X}_1$, $\mathbb{P}_1$, $\mathbb{Y}_1$ and $\mathbb{X}_2$, $\mathbb{P}_2$, $\mathbb{Y}_2$ according to the way we partition points described earlier, and most importantly store $\{t_{optimal}, z_{optimal}\}$ as an attribute of the node as described earlier. It then proceeds to recursively partition $\mathbb{X}_1$, $\mathbb{P}_1$, $\mathbb{Y}_1$, $\mathbb{X}_2$, $\mathbb{P}_2$, $\mathbb{Y}_2$ storing the best $\{t_{optimal}, z_{optimal}\}$ for their respective datasets until a specified depth is reached (the depth is decremented by 1 after every recursive call, for more details on the code refer to the appendix). At the node where you are no longer partitioning the dataset (depth is 0), you attribute a new element to that node which is its value. The value can be defined in different ways: for regression tasks you want to associate the

node with the average values inside the Y, and for multi-class classification tasks you want to associate the value of that node with the $mode(Y)$. Now that we have built this decision tree we can classify any point $r \in \mathbb{R}^d$ by traversing through the tree and reading the value associated with the node at the end of the traversal. The way you traverse through the tree is by comparing r to the $\{t_{optimal}, z_{optimal}\}$ associated with each node and if $r[z_{optimal}] > t[z_{optimal}]$ then go to the partition of all training points that are greater then $t[z_{optimal}]$ and do the opposite if $r[z_{optimal}] \leq t[z_{optimal}]$. Stop when you cannot go to any other partition and predict $h(r)$ as the value of the partition where you stopped.

3.1.1 Problem with Decision Trees

The problem with decision trees is that they can be very poor learners with a great dependency on the depth of a tree resulting in either a high variance error predictor (very deep trees) or a high bias error predictor (very shallow trees). Therefore, in the next section we will introduce algorithms that take this into account and combine decision trees into ensemble methods to compensate for either this high bias error or high variance error.

3.2 Boosting and Adaboosting

The boosting family of algorithms for classification, described generally and at a high level, is any method that is based on aggregating rougher "moderately inaccurate" decision rules (weak hypotheses learned by weak learners) into a combined decision rule [3]. For AdaBoost, this amounts to combining individual high bias, weak learners into an ensemble learner, where the weak learner being trained is a decision stump (decision tree of depth one). AdaBoosting is thus directly improving on the high bias error asymptotic case of decision trees.

3.2.1 AdaBoosting decision rule

**Note: Notation for AdaBoosting given in the appendices*

The ensemble decision rule outputted by AdaBoost is as follows:

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

From this expression, it is clear that AdaBoosting creates an ensemble decision rule, since the prediction for any given point is arrived at by taking a linear combination of the predictions of all the constituent learners. Intuitively, this can be thought of as taking a weighted majority vote of the ensemble of weak decision rules. The key to understanding AdaBoost is thus understanding how alpha is calculated, its interpretation, and how the weak learners are trained.

3.2.2 AdaBoosting Algorithm

The AdaBoost algorithm is best highlighted in the pseudo-code (adapted [3]) in the appendices. In the notation and pseudo-code, there is reference to a distribution: a vector of weights that sum to one, with a weight allocated to each example. Intuitively, the distribution weights tell the weak learner currently being trained which examples to most focus on. The first weak learner is trained according to a uniform distribution where all points are allocated weight $1/n$ where n is the number of examples. After training this weak learner, the error is calculated, α_t is calculated using the error (a lower error = higher alpha), and finally the new distribution weights are computed based on α_t. When recalculating the distribution weights, the points that are misclassified have a higher weight associated with them, thus causing the next trained weak learner to focus on them further, and the correctly classified points are assigned a lower weight. In essence what the algorithm is doing is *adapting* the distribution weights to the errors of the individual weak learners, and *boosting* the performance of the ensemble decision rule by training weak learners sequentially that "cover" the mispredictions of the previously trained weak learners. (thus *AdaBoosting*). Thus, the AdaBoost outputted decision rule can be interpreted as the following: the sign of the individual weak learner $h_t(x)$ gives its predicted label and the magnitude of this sum element (and

thus the magnitude of α_t) gives the confidence in that individual weak learners prediction. This is directly in-line with the weighted majority vote interpretation given earlier.

3.2.3 Training Error and Guarantees on the AdaBoost Algorithm

AdaBoost's most fundamental property, and why it is guaranteed to provide a strong learner that classifies better than its constituent weak learners, is its ability to reduce training error exponentially fast. To understand this property, one must recall that on a binary classification task, a purely random guess on a given point has a probability of 1/2 of correctly predicting the point's label. Thus, we can rewrite the error of any given weak hypothesis h_t as: $\epsilon_t = \frac{1}{2} - \gamma_t$, where γ_t can be interpreted as "how much better than random are h_t's predictions" [3]. Freund and Schapire show that the upper bound on the training error of the strong hypothesis H_t is:

$$\prod_t \left[2\sqrt{\epsilon_t(1-\epsilon_t)}\right] = \prod_t \sqrt{1-4\gamma_t^4} \leq \exp\left(\sum_t -2\gamma_t^2\right)$$

Thus, "if each weak hypothesis is slightly better than random so that $\gamma_t \geq \gamma$ for some $\gamma > 0$, the training error drops exponentially fast" [3] over iterations t. Thus, the AdaBoost algorithm can be thought of as a sort of steepest descent algorithm operating over this training error function. The strength of the AdaBoost algorithm is that this lower bound γ need not be known "a priori before boosting begins", since the AdaBoost algorithm "adapts" to the "error rates of the individual weak hypotheses" [3]. Though this is based on the assumption that each individual learner is a weak learner, which does not always hold for every data set, and does not always hold in the case where the number of iterations T is not asymptotic.

Further, the performance of the AdaBoost algorithm in terms of the max iterations, T, is understood through Freund and Schapire's expression for the generalization error:

$$\hat{P}[H(x) \neq y] + \tilde{O}\left(\sqrt{\frac{Td}{m}}\right)$$

This implies that the generalization error increases with higher with max iteration T. Traditionally, this has been interpreted as overfitting, but experimental results show that the AdaBoost algorithm does not neccesarily overfit [3]. In response, Schapire derives an upper bound on the generalization error of the AdaBoost algorithm using margins (Schapire's margin definition is given in the appendices):

$$\hat{P}[\text{margin}(x,y) \leq \theta] + \tilde{O}\left(\sqrt{\frac{d}{m\theta^2}}\right)$$

This upper bound on generalization error, defined for $\theta > 0$, is independent of the max iterations T, and thus the generalization error does not increase directly with the maximum number of iterations. An interesting property of the AdaBoosting algorithm is that it is particularly aggressive in reducing these margins, and Schapire shows that the left term monotonically decreases over iterations such that even if the training error were to converge, it is possible for the algorithm to keep decreasing the testing error (and thus the generalization error) by continuing to decrease these margins. The net result of the AdaBoost algorithms in any case where T is sufficiently large is thus a strong learner with a bias error significantly lower than any of its constituent weak learners.

Complementing this is the Random Forest algorithm, which addresses the opposite problem in a complementary way: driving high variance error down by aggregating high variance trees in parallel. The Random Forest also parallels the AdaBoost algorithm in some of its theoretical properties regarding overfitting.

3.3 Theory behind Random Forests

3.3.1 Introduction

The idea behind Random Forests is to aggregate a set of decision tree learners with high variance in order to produce a low error, low variance predictor. This works by growing all the decision trees in parallel (with a slightly different method than the previously explained procedure for the decision tree section) and then predicting any point by selecting the majority

vote for multi-class classification tasks or computing the mean vote of all decision trees for regression tasks. The following sections will give a precise definition of Random Forests, why they are effective, and what theoretical guarantees they have for performance.

Definition

The definition of Random Forests for Multi-Class classification given by (Brieman, 2001) is the following:

Definition 1. *A Random Forest is a classifier consisting of a collection of tree structured classifiers $h(x, \Theta_k), k = 1, ...$ where the Θ_k are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input x.*

Since this definition of decision trees is different from the one presented earlier, it warrants further explanation. Here Θ_k can represent two important parameters of Random Forests: the first representation of Θ_k is recognized as bootstrapped data, where Θ_k represents a random vector that generates a subsample of the original training set by randomly selecting points with replacement. In the second representation, Θ_k is recognized as a set of dimensions from which each node will try to partition on (instead of all dimensions as previously defined in the Decision Tree Procedure). We will see later on that both will be beneficial to the procedure of Random Forests. Additionally, note that in Random Forests all points have equal probability, so we can disregard the earlier $\mathbb{P}$ notation from Section 3.1. If we make the small addition of our earlier protocol to take in the dimensions to split at each node, then we can use the earlier defined procedure. We will continue with Brieman's notation, but we wanted to highlight that any difference can be adapted to what we explained earlier. Now, what are the guarantees Random Forests can make given this definition of Θ_k?

Analysis

The most important aspect of the analysis of Random Forests involve the overall generalization error of the ensemble of decision learners. That error can be very simply defined for a set of classifiers $h_1, ..., h_k$ as:

$$P_{GE} = P_{\mathbf{X},Y}(mg(\mathbf{X}, Y) < 0) \tag{5}$$

where,

$$mg(\mathbf{X}, Y) = av_k I(h_k(\mathbf{X}) = Y) - max_{j \neq Y} av_k I(h_k(\mathbf{X}) = j) \tag{6}$$

and I is the identity function. This margin $mg(\mathbf{X}, Y)$ is just a function that takes in a single point in the entire labeled dataset and computes the difference between the average number of correct predictions of each decision tree h_i and the maximum average number of incorrect predictions made on the same point. If the margin function is smaller than 0, then it means you made more incorrect predictions of a single point then correct and your ensemble or random forest would end up incorrectly classifying such point. Therefore, assuming the $P_{X,Y}()$ denotes the probability of an event over the entire dataset, then the probability of error is the sum of all probabilities of points in the dataset that have $mg(\mathbf{X}, Y)$ smaller than 0. Now Brieman goes on to show that as the number of trees increases, for almost surely all sequences $\Theta_1, ..., \Theta_k$ the P_{GE} converges to:

$$mr(\mathbf{X}, Y) = P_{X,Y}(P_\Theta(h(X, \Theta) = Y) - \max_{j \neq Y} P_\Theta(h(X, \Theta) = j) < 0) \tag{7}$$

This is a very important result that highlights no matter how many trees you add to your ensemble, the overall error will converge to the error of a single decision learner over a distribution Θ which generates the i.i.d Θ_i inside the ensemble. It is also important to mention that this does not provide an upper bound on $mr(\mathbf{X}, Y)$ (which really is the error), but rather it gives a mathematical statement. If we have a finite number l of decision trees in an ensemble learner, then the error of the ensemble is guaranteed to be ϵ away from the $mr(\mathbf{X}, Y)$ defined above. Adding more trees can only make the error of the ensemble closer to $mr(\mathbf{X}, Y)$. Therefore, we can conclude that adding a significant number of trees will not drive the generalization error to higher values and will prevent overfitting. From this definition:

$$P_{GE} = P_{\mathbf{X},Y}(mr(\mathbf{x}, y) < 0) \tag{8}$$

Brieman also shows that the overall generalization error can be upper bounded by:

$$P_{GE} \leq \frac{var(mr(\mathbf{X}, Y))}{\mu^2} \tag{9}$$

(for a proof of the following refer to the appendix) Therefore, we know that the upper bound of generalization error decreases with variance. Since decision trees are high variance predictors, we must examine how Random Forests can be implemented to reduce variance and thus generalization error. First, we let:

$$var(\mathbf{x}) = \mathbb{V}_{\theta_1,\ldots,\theta_M}[\frac{1}{M}\sum_{m=1}^{M}\psi_{\theta_m}(\mathbf{x})] \tag{10}$$

where $\psi_{\theta_m}|m = 1, \ldots, M$ are a set of M randomized models built on the same data, but each built from an independent random seed θ_m. By incorporating the Pearson's correlation coefficent $\rho(\mathbf{x})$ and performing some simplification described in the appendix, we get the variance of the randomized ensemble to be:

$$var(\mathbf{x}) = \rho(\mathbf{x})\sigma_\theta^2(\mathbf{x}) + \frac{1-\rho(\mathbf{x})}{M}\sigma_\theta^2(\mathbf{x}) \tag{11}$$

where $\sigma_\theta^2(\mathbf{x})$ is the variance of a single learner. Right away, we see that (1) the correlation of the models and (2) the number of models contribute to the ensemble's variance and generalized error. If we decorrelate trees in an ensemble, $\rho(\mathbf{x}) \to 0$ and variance reduces to $\frac{\sigma_\theta^2(\mathbf{x})}{M}$, which can be further reduced as the size of the ensemble grows (M$\to \infty$). On the other hand, if the trees are correlated ($\rho(\mathbf{x}) \to 1$), then the variance of the ensemble becomes $\sigma_\theta^2(\mathbf{x})$, which gives the ensemble no benefit. Overall, random perturbations in conjunction with a large number of models can drive down variance and generalization error. The general tuning parameters for a Random Forest classifier characterize the conclusions found in this section. They are: Bootstrap Sample Ratio, Feature Subsample size, Depth of Trees, and Number of Trees. The effects of these tuning parameters on variance and CCR will be explored in the Experimental Results section.

4 Implementation

All code for the implementation of Decision Trees, AdaBoost and Random Forests algorithms was implemented from scratch in Python. The pseudocode for each method is provided in the Appendix section and the actual code is linked in the Github repository and Google Drive folder. At a high level view, the AdaBoost algorithm is a one-to-one implementation of the pseudo code, where the weak learner (decision stump) is trained using our own decision tree class/function with depth = 1. Similarly, the Random Forest algorithm uses the decision tree class, but specifies that the decision tree will subsample at each node. Both the Adaboost and Random Forest algorithms are tested on linear, Gaussian, spiral, and real world datasets. The only code from an outside source that is present in our Implementation are SciKitLearn and Matplot libraries. SciKitLearn is used for advanced tuning of Random Forests on the real world dataset because it is computationally more efficient and we could not feasibly run our code on the real world dataset on our current hardware [5]. Matplotlib is used to create visualizations of our results.

5 Experimental Results

5.1 Experimental Overview

The purpose of our experiments is to compare how Adaboosting and Random Forest perform with respect to training and testing CCR (CCR since these are classification set), and to explore how both algorithms decrease error. Both algorithms were tuned and tested on four datasets: a linearly separable dataset, a Guassian distributed dataset, a spiral dataset, and a real world dataset. The Linearly Separable dataset is useful for comparing ensemble performance to basic SVM performance. Further, we expect all three algorithms to achieve high CCRs on this dataset. The Gaussian dataset will be useful for distinguishing the ensemble methods from SVM because basic SVM can only perform well on linearly separable datasets. Only the Adaboost and Random Forest will be tested on the Spiral dataset - the pur-

pose of this experiment is to visualize how each algorithm aggregates the individual learners to arrive at a decision boundary. Finally the real world dataset, which has high dimensionality, will be used specifically for testing and tuning the hyperparamters of the Random Forest model.

5.2 Linearly Separable Dataset

The following datasets contain 40 points, 20 of class 1 and 20 of class -1. The 1D dataset is separable by by a horizontal line and the 2D dataset is separable by a diagonal line. They were generated by manually selecting points inside a grid. Figure 1 displays the AdaBoost decision boundary when it was trained on 20 stumps. Figure 2 displays the Random Forest decision boundary that was trained on 8 trees, each of depth 5 and bootstrap ratio of 0.3. Figure 3 displays a basic SVM decision boundary. The training CCR for all three algorithms is 1.0, as expected. We didn't have any test set for this dataset because this experiment mainly serves to show that AdaBoost and Random Forests can learn linear decision boundaries like the algorithms we were taught in class.

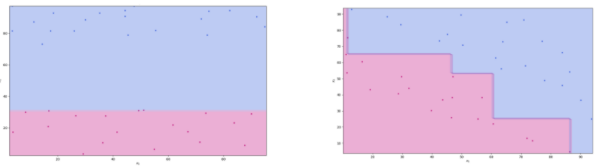


Figure 1: AdaBoost on 1D and 2D Linearly Separable Dataset

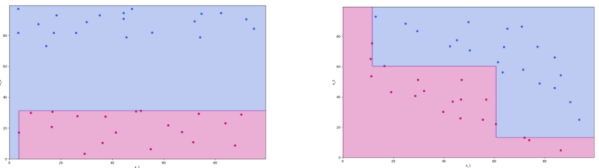


Figure 2: RF on 1D and 2D Linearly Separable Dataset

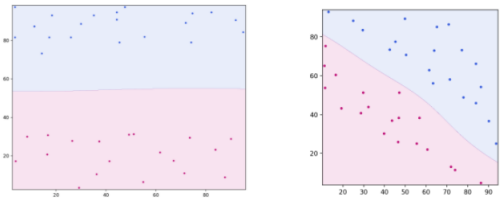


Figure 3: SVM on 1D and 2D Linearly Separable Dataset

5.3 Gaussian Dataset

The following dataset contains 400 points, 200 of class 1 and 200 of class -1. It was generated with a Gaussian distribution of mean 0 and variance 1. The left plot in Figure 4 is the Adaboost model, trained on 40 stumps. It achieved a CCR of 0.975. The right plot is the Random Forest model, trained on 15 trees, each of depth 7 and bootstrap ratio of 0.3. It achieves a CCR of 0.995 on the whole dataset. We did not split the dataset into training and testing set because we simply wanted to have a proof of concept that AdaBoost and Random Forests can learn complex decision boundaries. They clearly did because they achieved a CCR of 0.975 and 0.995, respectively.

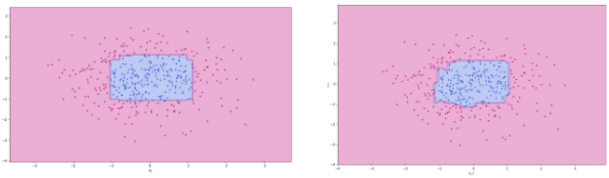


Figure 4: AdaBoost/RF on Gaussian Dataset

5.4 Spiral Dataset

The following dataset contains 800 points, 400 of class 1 and 400 of class -1. The purpose of experimenting with this dataset is to get a better understanding of how the decision boundaries generated from AdaBoosting and Random Forests evolve by adding more decision trees. Figure 5 displays the Adaboost and Random Forest evolution over 1, 5, 10, and 50 trees. For the AdaBoosting method (displayed on the

top row of Figure 5), the decision boundary starts with high bias - the decision boundary is 1 dimensional and does not differentiate the classes. As more stumps are added, however, the decision boundary becomes more fitted to the dataset and the bias error is driven down. This trend is aligned with our expectations because AdaBoost is guaranteed to decrease bias error as T increases. On the other hand, the Random Forest method starts with a decision boundary that overfits to the spiral training data. As more trees are added, the decision boundary becomes more uniform while not increasing bias error. This trend also aligns with our expectations because Random Forests are intended to decrease the high variance aspect of Decision Trees. This coincides with Equation 11 from the Problem Formulation because as the number of trees increase, the variance decreases. Further, from examining the Testing CCR plots in the Appendix, we can see that AdaBoost converges relatively quickly when increasing the number of trees, though minor fluctuations persist and minor gains are possible when increasing the number of trees further. Contrastingly, the Random Forest CCR plot does show an initial spike in performance when increasing the number of trees being aggregated, however the long term trend is not as predictable. Both algorithms perform similarly well on this dataset, with AdaBoost (stumps=250) showing a training/testing CCR of .9819/.9625 and Random Forests (trees=50, depth=10, features=1, bootstrap=1.0) performing slightly better with training/testing CCR of 1.0/.9875.

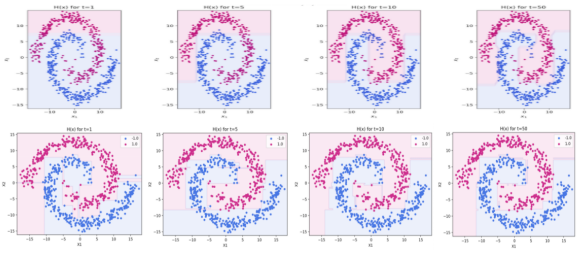


Figure 5: Evolution AdaBoost/RF on Spiral Dataset

5.5 Real World Data Set

This data set is a 12 feature binary classification data set where the labels are "have cardiovascular disease" (1) or "don't have cardiovascular disease" (-1), and the features including metrics such as age, height, weight, BMI, so on so forth. This data set was meant to highlight the behaviours of the different tuning parameters of the algorithms with respect to their performance metric CCR. The AdaBoost algorithm shows a Training CCR of .7256 and a Testing CCR 0.7321 after training with T=50 stumps. From the CCR plot, it is clear that the AdaBoost algorithm converges relatively quickly, though there is not much tuning to do past this other than increasing T which will provide incremental improvements to the CCR.

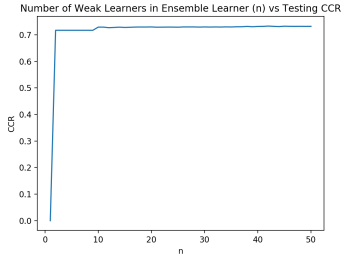


Figure 6: Testing CCR vs Number of Weak Learners in Ensemble Decision

Random Forests, on the other hand, require more tuning in order to achieve a comparable CCR. There are four hyperparameters that can be adjusted: Bootstrap Sample Ratio, Features Sample Size, Depth, and Number of Trees. It can be expected that increasing the Bootstrap Sample Ratio and the Feature Sample Size causes trees to be less diverse and possibly induce overfitting. Increasing depth, as mentioned in the Section 3.1, also increases variance. We anticipate that increasing number of trees in a Random Forest decreases variance and generalized error. That said, increasing number of trees also increases computational complexity and should be tuned carefully. To better understand how the tuning parameters affect CCR for cardiovascular disease classification, the following experiment was done: adjust one hyperparameter while holding the other three

constant. We understand this experiment is flawed because it does not account for hyperparameters interacting with each other, but it is still useful to test whether our intuitions are true about the effects of the individual hyperparamters. Figure 7 displays that the Bootstrap Sample Ratio has no clear trend when for tuning - this may be because the dataset is too small. The trend of the Depth and the Feature Sample Size indicates that our intuitions are true; increasing these hyperparameters too much decreases testing CCR. Increasing number of trees also increased Testing CCR. When these hyperparameters are tuned to the account for experiment results (trees=60, depth=5, features=5, bootstrap=0.8), the Random Forest Training CCR is 0.7285 and the Testing CCR is 0.7350.

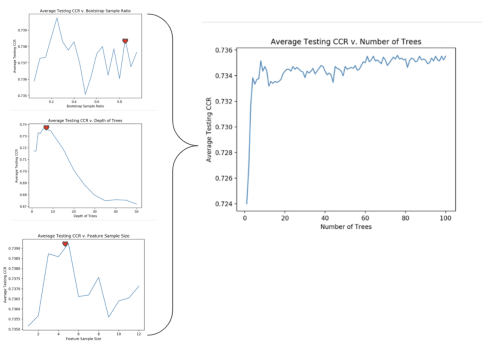


Figure 7: CCR vs Different Hyperparameters as well as final hyperparameters for Random Forests

6 Conclusion

In our theoretical work, we have shown that both AdaBoost and Random Forests come from a similar family of methods denoted as ensemble methods and perform very efficiently in combining a set of faulty decision rules into a strong one. More importantly, in the case where we have the capacity to run the algorithms for infinitely long and where the datasets satisfy certain theoretical constraints, the expectation is that the training error and generalization error will converge to zero for both algorithms. In fact for AdaBoost, we expect the upper bound on training error to exponentially decay with iteration t. The error convergence property is shown to great degree for both Random Forests and Ada Boost in the set of synthetic datasets that were tested (linear, gaussian, spiral). Furthermore, while AdaBoost shows a better "out of the box" performance (in regards to the relatively little tuning required), Random Forests performs as well if not better with proper tuning. When considered in the larger scheme of data science approaches, both algorithms are relatively minimal to tune when weighed against the performance benefit, and especially so when considering the tuning required for modern approaches such as neural networks. With that said, while the theoretical assumptions provide strong foundational understanding of the algorithms and their behavior, they do not always translate cleanly into real world scenarios. More specifically, we have shown with our experimental section that both algorithms though performing decently well (CCR of 0.7) across a real word dataset, do not drive the training error to zero, but rather to a constant value after a significant number of iterations. Overall, the group is thrilled to have explored the deep theoretical analysis behind these algorithms as well as their strengths and weaknesses in practice.

7 Description of Individual Effort

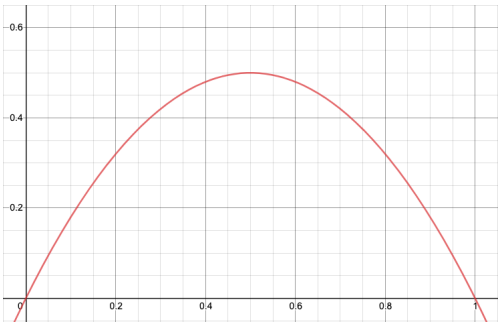
The group worked together in all aspects of the projects, with everyone getting their share of work done at the right time. Additionally, everyone discussed all algorithmic parts of the project to make sure we all came out understanding the theory we had explored as a whole. Each individual member was responsible for a specific aspect of the project: Andreas was responsible for the implementation and exploration of decision trees as well as random forests, Grayson was responsible for the implementation and exploration of bagged trees and random forests, and Yousuf was responsible for the exploration and implementation of boosting and AdaBoosting. As a whole, the group collectively agrees that the project was split fairly, and each person put in their 33 percent.

8 Appendices

8.1 Decision Trees

8.1.1 Gini Plot

To get some intuition of the gini impurity score we just defined take a simple binary decision problem.Since we only have two classes, if $p_1 = x$ then $p_2 = (1-x)$. Now taking a look at what $g(p_1, 1-p_1)$ looks like as a function of p_1



. We can see that if p_1 goes to 0 or to 1 meaning either the partition has increasingly less points belonging to class 1 or has increasingly more points belonging to class 1 then the impurity score $g(p_1, 1-p_1)$ goes to 0 which means we have created a pure partition.

8.1.2 Definition of G described in the decision trees section

Before we give the precise mathematical formula of G, let:

$$w_1(t,z) = \sum_{i=1}^{n} I(t[z] \leq x_i[z]) \tag{12}$$

$$w_2(t,z) = \sum_{i=1}^{n} I(t[z] > x_i[z]) \tag{13}$$

which can be interpreted as the weight of each partition, and

$$j_{1j}(t,z) = \frac{\sum_{i=1}^{n} I(t[z] \leq x_i[z]) I(y_i == j)}{w_1(t,z)} \tag{14}$$

$$j_{2j}(t,z) = \frac{\sum_{i=1}^{n} I(t[z] > x_i[z]) (y_i == j)}{w_1(t,z)} \tag{15}$$

which is just the weight of each class inside a partition. Then:

$$G(t,z) = w_1(t,z)(1 - \sum_{j=1}^{k} j_{1j}^2) + w_2(t,z)(1 - \sum_{j=1}^{k} j_{2j}^2) \tag{16}$$

which is just a weighted average of the impurity of each partition.

8.1.3 Decision Trees Complexity

Ans: By applying this greedy search method all we need to do in order to grow the full decision tree is find the indexes that sort $\mathbb{X}$ according to a single dimension, for every single dimension. Once you have that computing the best split for each node can be done in $O(ndk)$ where n is the number of points inside of $\mathbb{X}$, d is the number of dimensions and k is the number of classes inside Y. The number of times you do $O(ndk)$ is optimistically $log(n)$ if you want to grow the full decision tree and the decision tree is a balanced binary tree but in the worst case you may do $O(n * ndk) = O(n^2 dk)$. Therefore the overall computation complexity of the full grown decision tree is $O(nlog(n)dk)$ in the best case and in the worst case $O(n^2 dk)$ (notice we haven't included the cost of sorting since its complexity is fixed in the beginning of the algorithm and is only computed once).

8.1.4 Decision Tree Pseudo-Code

We followed the procedure below to make a decision tree:

- init tree = Tree(depth : integer, right tree: is empty, left tree: is empty, value: is empty, boundary is empty)
- initialize dataset X = some data, Y = labels for some data (only 1,-1), weights = associated probability function to dataset X
- call make tree(tree,X,Y,weights)
  - if: tree.depth = 0

        * then assign value of tree with tree.val = sign(Y*weights), and return tree
    - else
        * call optimal split(tree,X,Y,weights)
        * get left and right partitions of data from optimal split
        * init tree left with depth - 1, init tree right with depth - 1
        * call make tree(tree left,X left,Y left,weights left)
        * call make tree(tree right,X right,Y right,weights right)
        * assign tree.left= tree left, tree.right = tree right
    - return tree

The following is the procedure we coded to evaluate a point in a decision tree:

- check if tree left is None and if tree Right is None
    - return tree.val
- if not
    - check if tree.boundary greater than test point
        * x = call evaluate point(tree right, test point)
        * return x
    - else
        * x = call evaluate point(tree right, test point)
        * return x

8.2 AdaBoost Psuedo-Code

8.2.1 Notation

In the case of AdaBoosting, each of the components involved in the algorithm is defined rigorously as follows: Training Set:

$$\mathcal{D} : \{ (x_1, y_1), \ldots, (x_n, y_n) \}$$
$$x_j \in \mathbb{R}^d, y_j \in \{ +1, -1 \}, \forall j$$

Where all features and labels are drawn in an independent and identically distributed manner from a joint probability distribution on $\mathcal{X}$ (features) and $\mathcal{Y}$ (labels) $p(x, y)$

Error: The error ϵ_t is defined as:

$$\epsilon_t = P_{i \sim D_t}[y_i \neq h_t(x_i)] = \sum_{i : h_t(x_i) \neq y_i} D_t(i)$$

Weak Learner: A weak learner is any algorithm that satisfies the following:

When trained on set $\mathcal{D}_{train}$ generated by IID samples drawn from $p(x, y)$ and on distribution D_t, will produce a decision rule $h_t(x)$ such that:

$$\epsilon_t < 0.5$$

In the limit in where the number of iterations goes to infinity, and where (x_i, y_i) are part of the testing set, and $(X_{test}, Y_{test}) \sim p(x, y)$.

8.2.2 Pseudo-Code

Input:

$$\mathcal{D} : \{ (x_1, y_1), \ldots, (x_n, y_n) \}$$
$$x_j \in \mathbb{R}^d, y_j \in \{ +1, -1 \}, \forall j$$

Initialize: $D_1(i) = 1/n, \forall n$

For $t = 1, \ldots, T$:

1. Train weak learner according to distribution D_t, get weak hypothesis $h_t : X \to \{ -1, +1 \}$
2. Calculate Error ϵ_t
3. Calculate weight of weak learner (derivation omitted training loss omitted):

$$\alpha_t = \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$$

4. Update Distribution Weights:

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if} \quad h_t(x_i) = y_i \\ e^{\alpha_t} & \text{if} \quad h_t(x_i) \neq y_i \end{cases}$$

$$D_{t+1}(i) = \frac{D_t(i) \cdot e^{-\alpha_t y_i h_t(x_i)}}{Z_t}$$

Where Z_t is a normalization factor

Output:

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

8.2.3 AdaBoosting Margin Definition

Schapire defines the margin of a point (x, y) as:

$$\text{margin}(x, y) = \frac{y_i \sum_t \alpha_t h_t(x)}{\sum_t \alpha_t}$$

The margin evaluates to number between -1 and +1, which is only positive if and only if H correctly predicts the label y, and the magnitude of which can be interpreted as a confidence interval (Freund Schapire).

8.3 Random Forest Pseudo-Code

We followed the procedure below to make a random forest:
Input:

$$\mathcal{D} : \{ (x_1, y_1), \ldots, (x_n, y_n) \}$$
$$x_j \in \mathbb{R}^d, y_j \in \{ +1, -1 \}, \forall j$$

Parameters:

- Bootstrap Sample Ratio (*boot*): Ratio of $\mathcal{D}$ that is randomly sampled with replacement for each learner
- Feature Subsample Number (*subf*): Number of features from y_j that are randomly considered at each split of decision tree node
- Depth of Decision Trees (*dep*)
- Number of Decision Trees (M)

Code:
For $i = 1, \ldots, M$:

1. Randomly sample ($boot * \mathcal{D}$) samples from $\mathcal{D}$
2. Train a decision tree on sample, randomly considering *subf* samples at each node
3. End construction of decision tree after reaches a depth = *dep*
4. Save decision tree as an object

Output: Set of Decision Tree objects

We followed the procedure below to evaluate points:
Input: Set of Decision Tree objects
Code:

- **For** $i = 1, \ldots, M$:
  1. Load the ith decision tree object
  2. Evaluate input dataset $\mathcal{D}$ with specified decision tree
  3. Store vector with evaluated points in the ith column of a results matrix
- Compute the mode of each row in results matrix

Output: Row vector for evaluated points from Random Forests

Proof of RF upper bound property

Note that Chebyshev's inequality is given by:

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2} \tag{17}$$

So let $X = mr(\mathbf{X}, Y)$ from the previous expression and the strength or μ of the classifer is:

$$\mu = E_{\mathbf{X},Y} mr(\mathbf{X}, Y) \tag{18}$$

Now also let $k = \frac{\mu}{\sigma(mr(\mathbf{X},Y))}$, then

$$P(|mg(\mathbf{X}, Y) - \mu| \geq \mu) \leq \frac{var(mr(\mathbf{X}, Y))}{\mu^2} \tag{19}$$

, and we know that if $mr(\mathbf{X}, Y) < 0$ then $|mr(\mathbf{X}, Y) - \mu| \geq \mu$, so

$$P(|mr(\mathbf{X}, Y) - \mu| \geq \mu) \geq P(mr(\mathbf{X}, Y) < 0) = P_{GE} \tag{20}$$

Which means:

$$P_{GE} \leq \frac{var(mr(\mathbf{X}, Y))}{\mu^2} \tag{21}$$

Proof of RF variance property

$$var(\mathbf{x}) = \mathbb{V}_{\theta_1,\ldots,\theta_M}[\frac{1}{M}\sum_{m=1}^{M}\psi_{\theta_m}(\mathbf{x})]$$

where $\psi_{\theta_m}|m = 1,\ldots,M$ are a set of M randomized models built on the same data, but each built from an independent random seed θ_m. We can expand the expression to fit the form $\mathbb{V}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$:

$$var(\mathbf{x}) = \frac{1}{M^2}\left[\sum_{i,j}\mathbb{E}_{\theta_i,\theta_j}[\psi_{\theta_i}(\mathbf{x})\psi_{\theta_j}(\mathbf{x})] - \mathbb{E}_{\theta_1,\ldots,\theta_M}[\sum_{m-1}^{M}\psi_{\theta_m}(\mathbf{x})]^2\right]$$

$$= \frac{1}{M^2}\left[M\mathbb{E}_{\theta}[\psi_{\theta}(\mathbf{x})^2] + (M^2 - M)\mathbb{E}_{\theta'\theta''}[\psi_{\theta'}(\mathbf{x})\psi_{\theta''}(\mathbf{x})] - M^2\mu_{\theta}^2(\mathbf{x})\right]$$

where $\mathbb{E}_{\theta}[\psi_{\theta}(\mathbf{x})^2]$ denotes the expectation of two models with the same random seed while $\mathbb{E}_{\theta'\theta''}[\psi_{\theta'}(\mathbf{x})\psi_{\theta''}(\mathbf{x})]$ denotes the expectation of two models with different random seeds. Since we know θ_m are i.i.d variables, we know that Pearson's correlation coefficient $\rho(\mathbf{x})$ for two randomized models grown from any combination of $\theta^{'}$ and $\theta^{''}$ will be the same. In this context, we can deduce Pearson's correlation coefficient to be:

$$\rho(\mathbf{x}) = \frac{\mathbb{E}_{\theta'\theta''}[\psi_{\theta'}(\mathbf{x})\psi_{\theta''}(\mathbf{x})] - \mu_{\theta}^2(\mathbf{x})}{\sigma_{\theta}^2(\mathbf{x})} \tag{22}$$

With this relation and some simplification, we get the variance of the randomized ensemble to be:

$$var(\mathbf{x}) = \rho(\mathbf{x})\sigma_{\theta}^2(\mathbf{x}) + \frac{1-\rho(\mathbf{x})}{M}\sigma_{\theta}^2(\mathbf{x}) \tag{23}$$

8.4 Testing CCR and Final Dec. Boundary Plots: Spiral Dataset

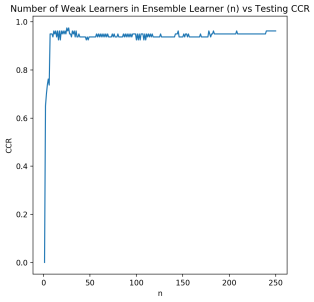


Figure 8: CCR vs Number of Trees In Ensemble Learner for AdaBoost

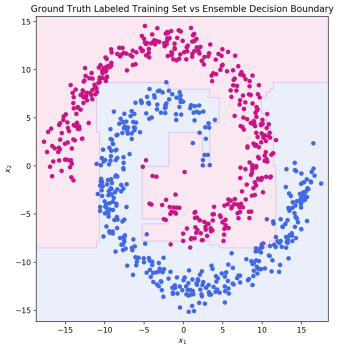


Figure 9: Final Decision Boundary AdaBoost

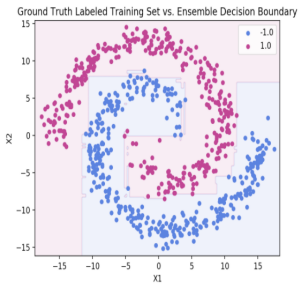


Figure 10: Final Decision Boundary RF

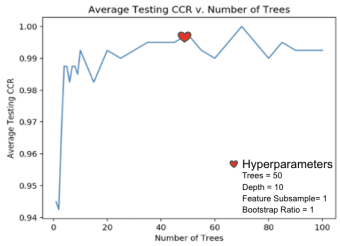


Figure 11: CCR vs Number of Trees In Ensemble Learner for RF

References

[1] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

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

[3] Y. Freund and R. E. Schapire. A short introduction to boosting. In *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1401–1406. Morgan Kaufmann, 1999.

[4] G. Louppe. Understanding random forests: From theory to practice, 2015.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[6] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning - From Theory to Algorithms*. Cambridge University Press, 2014.