SENG 474 Data Mining - Assignment 1

Andrew Braun - V00955955

January 31, 2025

1 Component 1: Experiment And Analysis

1.1 Introduction

The aim of this project was to evaluate and compare the performance of three supervised learning algorithms—Decision Trees (with pruning), Random Forests, and Boosted Decision Trees—on a binary classification task. These methods were applied to the Spambase dataset, which categorizes emails as either "spam" or "not spam." The dataset, sourced from the UCI repository with additional derived features, contains various attributes that reflect the characteristics of email content.

The experiments were conducted using *Python* and the *Scikit-learn* library, along with other supporting libraries, utilizing their built-in implementations of the algorithms. To ensure a robust comparison, a custom *k*-fold cross-validation method was developed to assess model performance. Each algorithm was tested under different hyperparameter configurations to examine their impact on training and test errors. Additionally, the relationship between model performance and training set size was explored to better understand the generalization capabilities of each approach.

This report outlines the findings from these experiments, emphasizing the strengths and limitations of each method and offering insights into their practical use for spam classification. Through systematic experimentation and analysis, we aim to illustrate how tuning algorithm-specific parameters and employing ensemble techniques can enhance predictive accuracy on real-world datasets.

1.2 Part 1: Separate Analysis

In this section, we will discuss the three main methods—decision trees, random forests, and boosted decision trees—individually. The original dataset includes a large number of features, many of which were derived from additional attributes. The most critical attribute is the last one, which indicates whether an email is "spam" (1) or "not spam" (0). Notably, the dataset was organized such that all spam emails (1's) were listed first, followed by non-spam emails (0's), meaning the data was not shuffled. As a result, the first step was to shuffle the data randomly and then split it into an 80-20 ratio. The 80% portion was used for training the models, while the remaining 20% was reserved for testing.

The analysis of each method follows a straightforward procedure. First, a key hyperparameter is identified (e.g., maximum depth for decision trees, number of trees in a random forest, etc.). This hyperparameter is then varied across different values to observe how

the method performs under each setting. The primary reason for varying these hyperparameters is to address overfitting, which occurs when a model memorizes the training data too closely. While this can lead to near-perfect performance on the training set, it often results in poor generalization to new, unseen data. To evaluate performance, error rates were used as the primary metric. For decision trees, an additional step of reduced error pruning was applied, which will be discussed in detail in the following section.

1.2.1 Decision Trees

The process of creating a decision tree involves selecting the best attribute to split the data on, using metrics like Gini impurity or entropy. Both Gini and entropy are measures of impurity in classification problems, helping to determine how mixed the classes are within a subset of data. Entropy aims to split the data in a way that maximizes information gain, reducing uncertainty and improving class separation. Information gain measures the reduction in entropy (uncertainty) achieved by splitting the data based on a specific attribute, aiming to maximize the clarity and separation between classes in the resulting subsets. Gini impurity, on the other hand, quantifies the likelihood of misclassifying a randomly chosen element if it were labeled randomly according to the class distribution in the dataset. When analyzing decision trees, we compared the performance of trees built using Gini versus entropy.

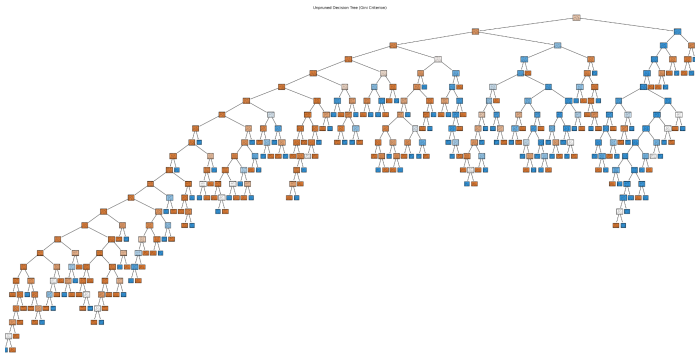


Figure 1: Gini criterion unpruned decision tree

First, we trained two unpruned decision trees on the training data—one using Gini and the other using entropy. Figures 1 and 2 show the fully grown trees for both criterions. Between the two, the entropy-based tree performed slightly better with a test error of 7.83%, whereas the gini based tree had a test error of 8.59%.

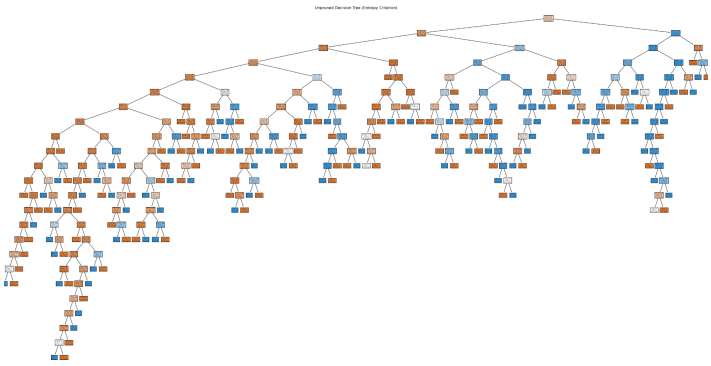


Figure 2: Entropy criterion unpruned decision tree.

To improve the decision trees and avoid overfitting—where the model memorizes the training data but fails to generalize to new data—we experimented with different hyperparameters. The first hyperparameter we varied was max depth, which controls how deep the tree can grow. Deeper trees can capture more complex patterns but are also more prone to overfitting. We trained trees

with different max depths and evaluated their performance on the test data. The results are shown in Figure 3.

For the Gini-based tree, the test error was minimized at a max depth of 16, with an error of 7.17%. For the entropy-based tree, the test error was minimized at a max depth of 18, with an error of 7.61%. These results show that limiting the tree depth helped reduce overfitting, as the test errors decreased significantly compared to the unpruned trees. At these optimal depths, the trees became more general and made fewer mistakes on unseen data.

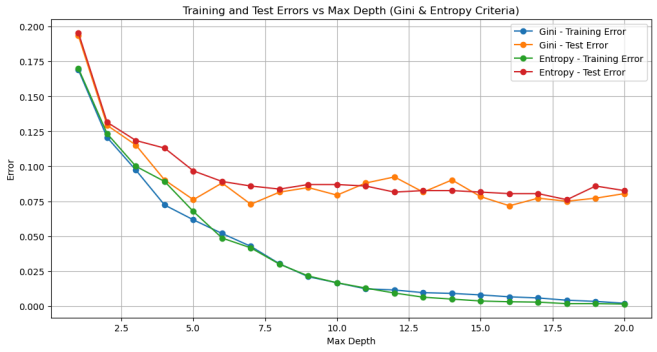


Figure 3: Plot of training and testing error vs max depth.

Next, we explored how the size of the training data affects performance. Instead of using 100% of the training data, we trained decision trees on random subsets of the data (e.g., 10%, 20%, etc.) and evaluated their performance on the test set. The results are shown in Figure 4.

For the Gini-based tree, the test error was minimized when trained on 80% of the training data, resulting in a test error of 7.72%. For the entropy-based tree, the test error was also minimized at 80% of the training data, with a test error of 7.07%. Interestingly, using only 80% of the training data led to better generalization, as the test errors were lower compared to using the full dataset. This aligns with the principle of Occam's razor, which suggests that simpler models are often preferable when performance is comparable.

Figure 4: Training and test error vs percentage of training data used

The final step for decision trees was reduced error pruning, which we will refer to simply as pruning. Pruning involves using a validation set to determine how well the tree generalizes during the pruning process. In this case, we split the training data further into 80% for training and 20% for validation, resulting in a final data split of 64% training, 16% validation, and 20% test. The pruning process works as follows: first, a full tree is grown on the training data. Then, starting from the bottom, leaf nodes are iteratively removed if their removal improves or maintains the tree's performance on the validation set. This process continues until no further improvements are made.

Figures 5 and 6 show two visualizations for each criterion: the error evolution during pruning and the final pruned tree for both Gini and entropy. As pruning progresses, the training and test errors begin to converge, indicating that the tree is becoming more general and less overfit. The final pruned trees are much simpler, though they remain somewhat unbalanced. Despite this, the pruning process resulted in slightly better test errors, with both Gini and entropy obtaining a test error of 7.50%.

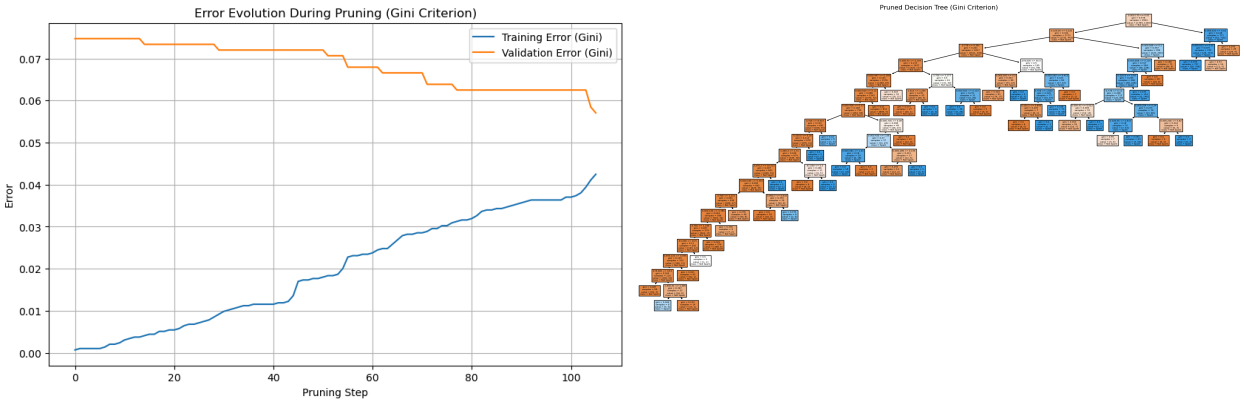


Figure 5: Error evolution of Gini decision tree during pruning process and final decision tree.

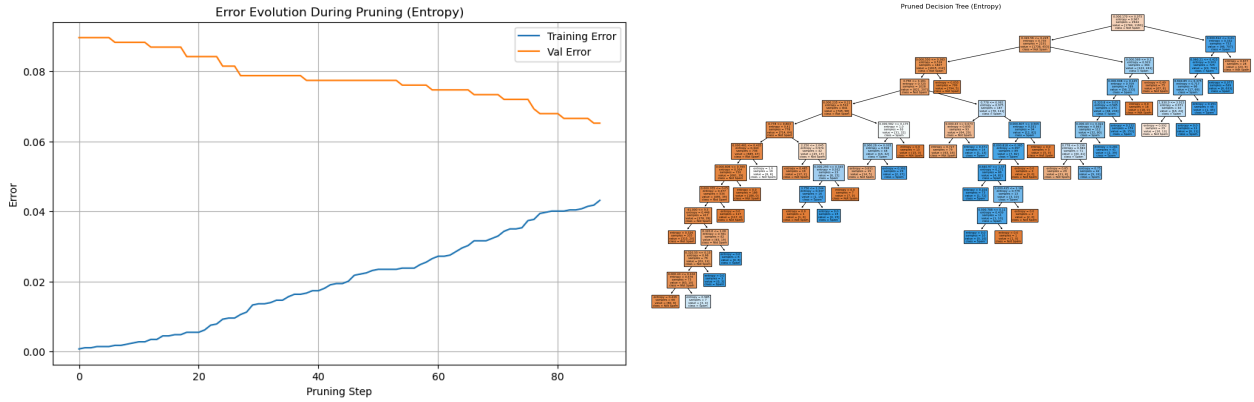


Figure 5: Error evolution of entropy decision tree during pruning process and final decision tree.

In conclusion, decision trees are a valuable and interpretable method for classification tasks, but their performance is limited by the number of available hyperparameters and the risk of overfitting. The best performance achieved in these experiments was with the entropy-based decision tree trained on 80% of the data, which achieved a test error of 7.07%. This result is particularly interesting because the entropy-based tree also performed best in its unpruned form, suggesting that entropy may be a more effective criterion for this dataset.

1.2.2 Random Forests

A random forest is an ensemble learning method that fits multiple decision trees on various sub-samples of the original dataset and averages their predictions. This approach improves accuracy and helps control overfitting. To construct a random forest, we use the following algorithm. First, a randomly selected subset of the original training set is chosen, with replacement, to train each decision tree. This ensures that each tree is trained on a different and random subset of the data. In our case, however, we set this subset to be the entire training set unless we explicitly vary this parameter. Second, at each split in a tree, a random subset of features is selected to determine the best split. This randomness helps prevent correlation between trees, ensuring diversity in the forest. Each tree is grown fully without pruning. As with decision trees, we vary hyperparameters such as the maximum depth of each tree, the number of trees in the forest, the number of features considered at each split, the percentage of the training set used for each tree, and the overall percentage of the training data used. Unlike decision trees, we do not prune the trees in a random forest.

First, we built a base random forest with no varied hyperparameters, to establish a benchmark. We constructed two random forests: one using the Gini criterion and the other using entropy. Both forests consisted of 100 trees, and the number of features considered at each split was set to $\sqrt{d}$, where d the total number of features in the dataset. The Gini-based forest achieved a test error of 5.87% with an average tree depth of 33.84, while the entropy-based forest achieved a test error of 5.54% with an average tree depth of 30.98. For simplicity, we proceeded with the Gini criterion, as it is the default in scikit-learn.

The first hyperparameter we varied was the maximum depth of each tree. Figure 6 shows how the test error changes with different maximum depths. The error was minimized at a maximum depth of 36, with a test error of 5.43%. This is interesting because the average depth of trees in the base forest was 33.84, suggesting that the base forest's trees that were too shallow were affecting error. The oscillation in test error at different depths indicates that trees of certain depths have a significant influence on the forest's overall performance.

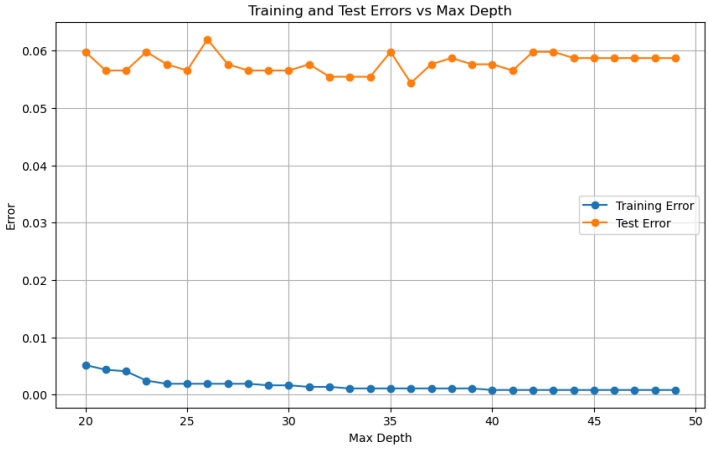
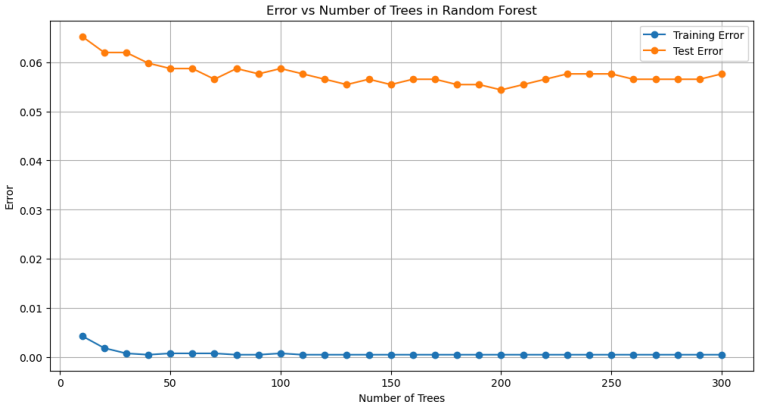


Figure 6: Training and test error vs max depth for random forests

Next, we varied the number of trees in the forest. The base forest used 100 trees, but increasing this to 200 trees resulted in a lower test error of 5.43%, as shown in Figure 7. The downward trend in error as the number of

Figure 7: Training and test error vs random forest size.



trees increases is expected because random forests improve generalizability by averaging predictions across multiple trees. More trees lead to a more accurate and stable average.

The next hyperparameter we explored was the number of features considered when determining the best split at each node in the trees. As previously mentioned, the standard number of features used is $\sqrt{d}$, which represents the total number of features in the dataset. To test the impact of this parameter, we selected a range of values centered around $\sqrt{d}$. Specifically, we used a spread of values ranging from half of $\sqrt{d}$ to $2*\sqrt{d}$. This ensured a reasonable range without extreme values. The lower bound was set to $\sqrt{d}$ minus the variation, and the upper bound was $\sqrt{d}$ plus the variation. We then evenly spaced out the values within this range for testing. Figure 8 illustrates how the random forest's performance changes with different numbers of features. The standard number of features in our dataset was 34, but the optimal number of features turned out to be 39, achieving a test error of 5.33%. This suggests that considering slightly more features than the default can improve the model's performance.

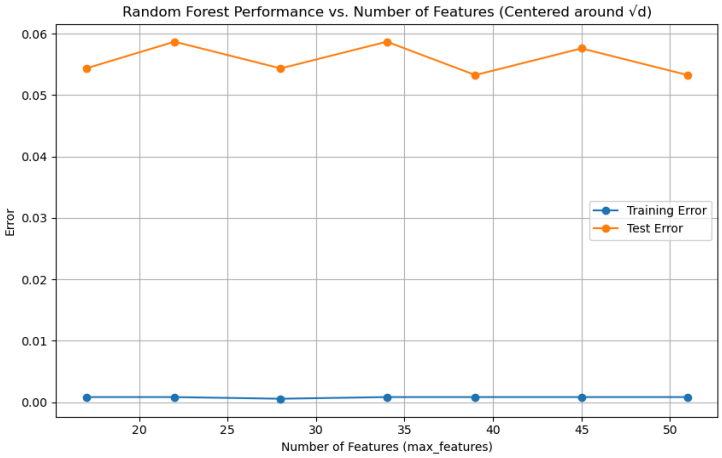
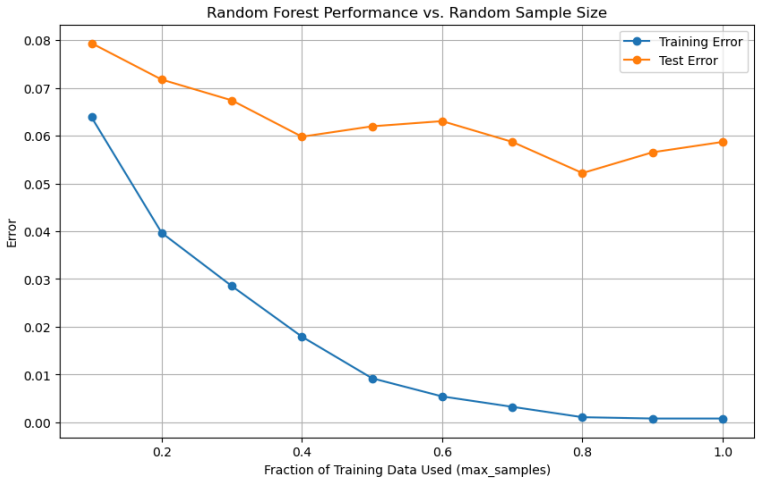


Figure 8: Training and test error vs number of features used for random forests

In our next experiment, we examined how the percentage of the training data used for each tree, denoted as n, affects the model's error. For example, if n is 60%, the first decision tree is trained on a random 60% subset of the training data, the second tree on another random 60% subset, and so on. These subsets are sampled with replacement, meaning they may overlap. Figure 9 shows how the error changes as we vary the percentage of the training data used for each tree. The error exhibits an overall downward trend, reaching its minimum at 80%. When each tree is trained on a random 80% of the training data, the model achieves its lowest error of 5.22%.

Another interesting observation is the parabolic shape of the training error curve. This pattern arises because, when n is small,

Figure 9: Training and test error vs sample size for random forests



each tree is trained on a limited portion of the data, reducing its ability to to perform well on the training set. As *n* increases, the trees are exposed to a more representative sample of the data, enabling them to learn more meaningful patterns and reducing the error. However, as n increases, the model learns the training too well and becomes overfitted, leading to this downward parabolic curve that we notice.

In our final experiment, we measured how the error changes as we vary the percentage of the overall training data used. This is distinct from the previous exercise. Here, we took a random percentage of the training data and trained each tree on 100% of that subset. The results, shown in Figure 10, indicate that using 100% of the training data yields the best performance. This outcome makes sense because random forests rely on the principle of generalizability through ensemble learning. By training each tree on the full dataset, the forest has access to more information, improving the accuracy of the majority vote when making predictions on the test set. In contrast, a single decision tree acts as a lone decision maker, lacking the collective strength of multiple trees. As a result, the minimum error achieved in this experiment was 5.87%, demonstrating the effectiveness of using the entire dataset for training in a random forest framework.

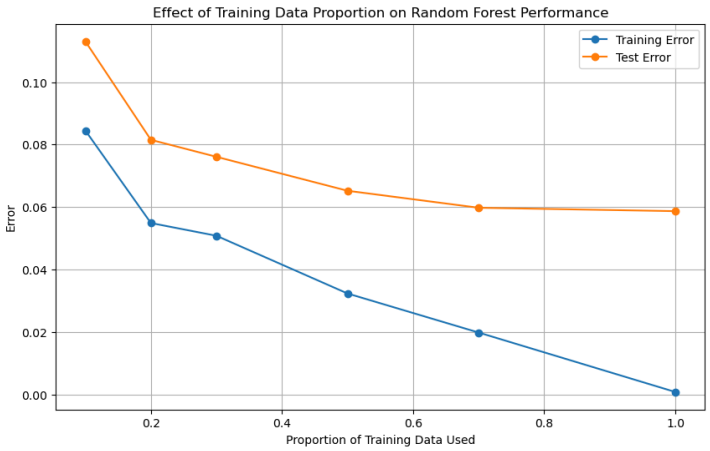


Figure 10: Training and test error vs percentage of training data used for random forest

In conclusion, random forests are an effective method for achieving generalization through their majority voting mechanism. In this section, the best performance was achieved when we varied the percentage of the training data used to train each tree, resulting in an error of 5.22% at a sample size of 80%. Notably, every experiment conducted with random forests outperformed the corresponding experiments with decision trees, highlighting the superior generalization and robustness of random forests for this classification task.

1.2.3 Boosted decision trees (AdaBoost)

In this section, we explore a different type of classifier known as boosted decision trees. These classifiers use an iterative algorithm to sequentially improve performance. In our case, we employed the AdaBoost algorithm. The process begins with a weak learner, which, for us, is a decision stump—a decision tree with only one node. The goal of this weak learner is to perform slightly better than random chance, which we can think of as achieving an error rate of just under 49.99%. Initially, each data point in the training set is assigned an equal weight, and the weak learner is trained on this weighted data. After the first iteration, the weights are adjusted: data points that were misclassified have their weights increased, while correctly classified points have their weights decreased. This adjustment ensures that subsequent weak learners focus more on the previously misclassified data. The algorithm repeats this process, regrowing the weak learner on the updated weights each time, and ultimately produces an ensemble of trees. This iterative weighting and training process allows the model to progressively improve its accuracy.

As with previous methods, we first evaluated the performance of a base boosted decision tree on the test set after training it on the training set. We used 100 boosting iterations and started with a decision stump as the weak learner. For the Gini criterion, the test error was 6.20%, while the entropy criterion resulted in a slightly higher test error of 6.41%. Interestingly, the final training errors were 2.69% for Gini and 2.74% for entropy. This contrasts with decision trees and random forests, which achieved training errors close to 0%, indicating significant overfitting. In the case of boosted trees, however, the model does not appear to overfit as much initially. For simplicity, as with random forests, we will proceed using the default criterion in scikit-learn, which is the Gini criterion.

In our first experiment, we varied the maximum number of boosting iterations to observe its impact on performance. As shown in Figure 11, the minimum test error was achieved at 50 iterations. For clarity, we tested a range of iterations starting from 10, increasing to 50, and then incrementing by 50 up to 500. The minimum test error of 5.76% occurred at the lower end of this range, specifically at 50 iterations. Unsurprisingly, as the number of boosting iterations increased, the training error continued to decrease, indicating overfitting. This demonstrates that while more iterations improve performance on the training data, they can harm generalization to the test set.

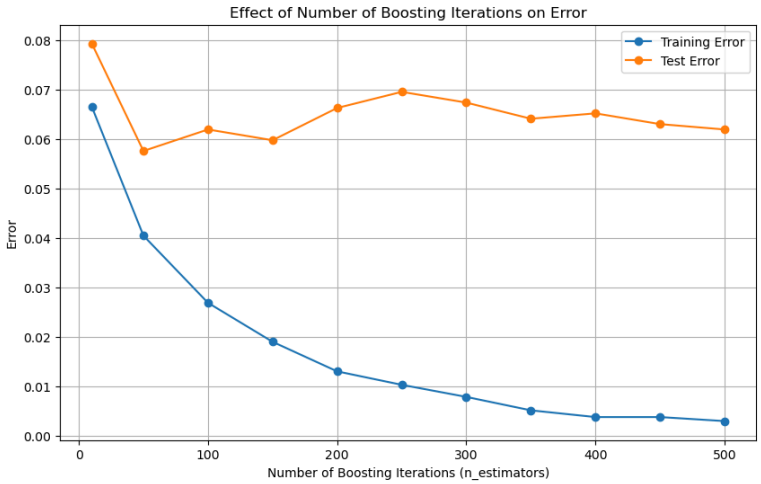


Figure 11: Training and test error vs boosting iterations

Next, we experimented with varying the maximum depth

of the initial weak learner. To clarify, if the maximum depth is set to 2, a decision tree of depth 2 is grown and then incorporated into the AdaBoost algorithm, where it is iteratively retrained on updated weights. As shown in Figure 12, the minimum test error was achieved when the weak learner had a maximum depth of 10. This improvement over smaller depths is due to the increased complexity that a deeper tree can capture, allowing it to model more intricate patterns and relationships in the data compared to a simple decision stump. At this depth, the model achieved a test error of 4.67%. However, as the depth of the weak learner increases beyond 10, the error begins to worsen, a clear sign of overfitting. Another notable observation is the sharp drop in training error when increasing the depth from a decision stump (depth 1) to a tree of depth 2. This indicates that even a small increase in depth can lead to significant overfitting, highlighting the sensitivity of the model to the complexity of the weak learner.

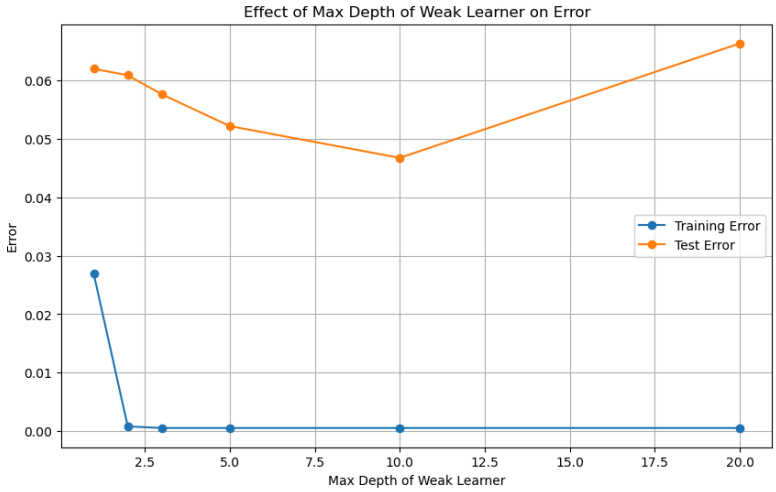


Figure 12: Training and test error vs weak learner depth

In another experiment, we varied the percentage of the training data used to train the boosted tree ensemble. Similar to previous methods, we took a random sample from the training data and trained the boosted tree on this subset. While using 100% of the data can sometimes lead to overfitting, in our case, it resulted in the best test error, as shown in Figure 13. The minimum test error achieved was 6.22%. One reason for this outcome is that having access to the full dataset allows the algorithm to learn more robust and generalizable patterns, reducing the impact of noise or biases in the data. Additionally, unlike other methods, boosted trees do not exhibit extreme overfitting, as evidenced by the fact that the training error does not approach zero. This suggests that the model generalizes well even when trained on the entire dataset. As a result, using 100% of the training data provided the best performance, achieving the lowest test error.

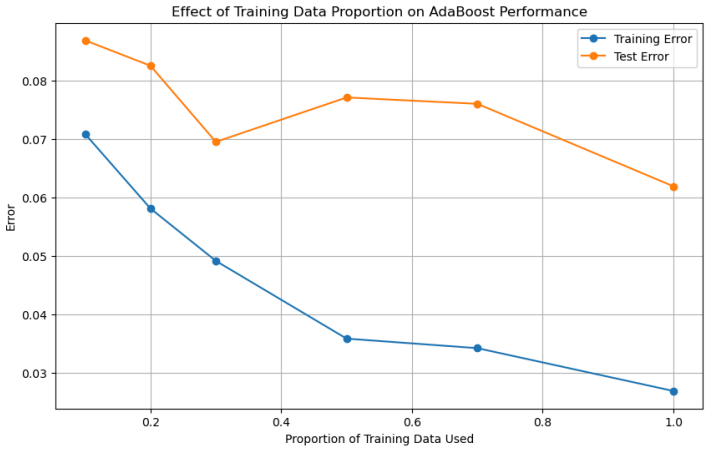


Figure 13: Training and test error vs percentage of training data used on boosted trees

In our final experiment, we varied the learning rate of the weak learner. The learning rate is a parameter that controls the contribution of each weak learner to the final prediction by determining the weight given to its output during the boosting process. It effectively sets the step size at which the model adapts to the data during training. Lower learning rates cause the model to learn more slowly, while higher rates lead to faster learning. A high learning rate can result in quicker convergence but increases the risk of overfitting, as the model may adapt too quickly to the training data, capturing noise and reducing its ability to generalize to new data. On the other hand, a low learning rate promotes slower, more stable learning, which can improve generalization and reduce overfitting. However, it often requires more boosting iterations to achieve optimal performance and may lead to longer training times.

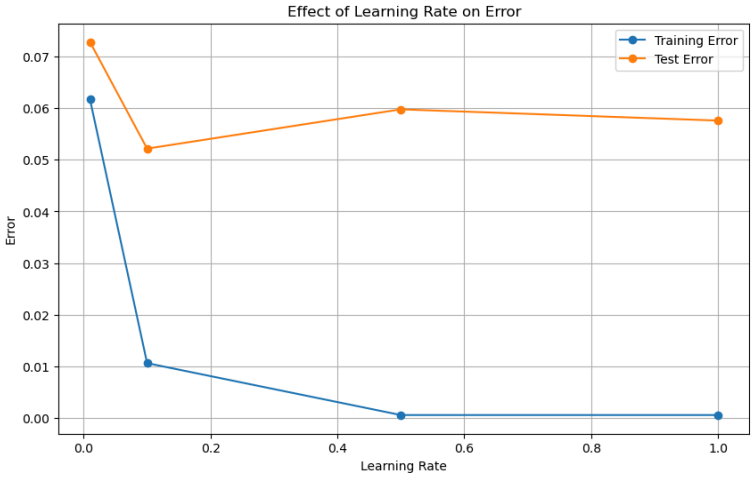


Figure 14: Training and test error vs learning rate of boosted tree

In previous experiments, we used a learning rate of 1.0. However, as shown in Figure 14, the optimal learning rate that minimizes test error is 0.1, achieving an error of 5.22%. Interestingly, the training error at this learning rate suggests that the model did not overfit but may have underfit the data, as the training error remained relatively higher. This indicates that while a lower learning rate improves generalization, it may also limit the model's ability to fully capture the underlying patterns in the training data.

In conclusion, boosted decision trees employ a boosting algorithm—AdaBoost, in this case—to transform weak learners into strong learners. The algorithm achieves this by iteratively adjusting weight distributions and retraining the weak learner on these updated weights. Through our experiments, we found that the lowest test error achieved in this section was 4.67%, which occurred when using a weak learner with a maximum depth of 10. This depth optimized the model's performance, striking a balance between capturing complex patterns and avoiding overfitting.

1.2.4 Conclusion

In conclusion, decision trees, random forests, and boosted decision trees represent a progression of techniques that build on the fundamental structure of decision trees to enhance performance and generalization. Decision trees are simple, interpretable models capable of capturing non-linear relationships, but they are prone to overfitting when grown too deep. Random forests

address this limitation by averaging multiple decision trees, reducing variance and improving robustness through bootstrapping and random feature selection. This makes them more reliable for complex tasks. Boosted decision trees further refine performance by sequentially training weak learners (decision trees) to correct the errors of previous ones. Techniques like AdaBoost improve both bias and variance, leading to superior predictive accuracy. In our analysis, boosted decision trees achieved the lowest test error of 4.67%, demonstrating their effectiveness. Each method—from basic decision trees to random forests and boosted trees—builds on the strengths of its predecessor, offering increasingly powerful tools for handling complex data while balancing model complexity and accuracy.

1.3 Part 2: Comparative Analysis

In this section, we employed the method of k-fold cross-validation, which operates as follows. We begin by randomly dividing our training set into k equal subsets, using k = 5 for simplicity. Next, we designate k-1 subsets as the new training set and use the remaining subset as the validation set. The model is trained on the k-1 subsets and validated on the remaining subset. This process is repeated k times, with each fold serving as the validation set once, and the remaining folds acting as the training set. The final performance metric is computed by averaging the results from all k iterations, providing a more reliable estimate of the model's generalization ability and reducing sensitivity to data splits.

In our experiment, we tuned two models: random forests and boosted decision trees. For random forests, the hyperparameter we tuned was the ensemble size, specifically the number of trees in the forest, while other parameters were kept at reasonable default settings. Similarly, for boosted trees, we also tuned the ensemble size, fixing the other parameters. We selected a minimum ensemble size of 10 and incremented the size of the ensemble accordingly.

1.3.1 Random Forests

Random Forests, an ensemble of decision trees, rely on the number of trees to balance bias and variance. Too few trees may result in underfitting, while too many can increase overfitting. To systematically determine the optimal ensemble size, k-fold cross-validation offers a robust method. By partitioning the training data into k subsets and iteratively training and validating the model on different folds, we can estimate the model's generalization error for various ensemble sizes. This process helps identify the ensemble size that minimizes validation error and offers insights into the model's stability across different data splits.

To aid visualization, we created a scatter plot, Figure 15, displaying the errors associated with different ensemble sizes across various folds. As a reminder, k-fold cross-validation works by partitioning the data into k subsets, training the model on k-1 subsets, and validating it on the remaining subset. The errors from each fold are then averaged to estimate the model's performance. In the plot, we can observe the errors for a specific ensemble size, such as 350,

represented by turquoise crosses. To compute the average error for this ensemble size, we track the error at each fold corresponding to the turquoise crosses, sum these individual errors, and divide the total by 5 (since k = 5 in our case). This provides the mean error across all the folds, which helps assess the generalization performance of the model for the selected ensemble size.

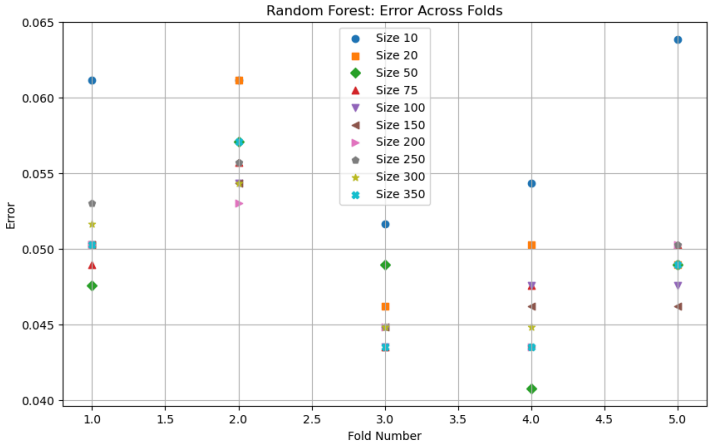


Figure 15: Scatter plot of k-fold cross validation performances for random forest ensemble sizes. Plotting fold number vs error

From this scatter plot, we can visualize the relationship between ensemble size and error by plotting the averages on a line graph. Figure 16 illustrates that an ensemble size of 150 yielded the lowest average training error during k-fold cross-validation, achieving a mean validation error of 4.84%. An interesting observation occurs at an ensemble size of 10, where the average validation error exceeds 5.80%, indicating significant poor performance as compared to the other ensemble sizes. The rapid decline in error as the ensemble size increases demonstrates that even small increments in ensemble size can lead to substantial improvements in model performance.

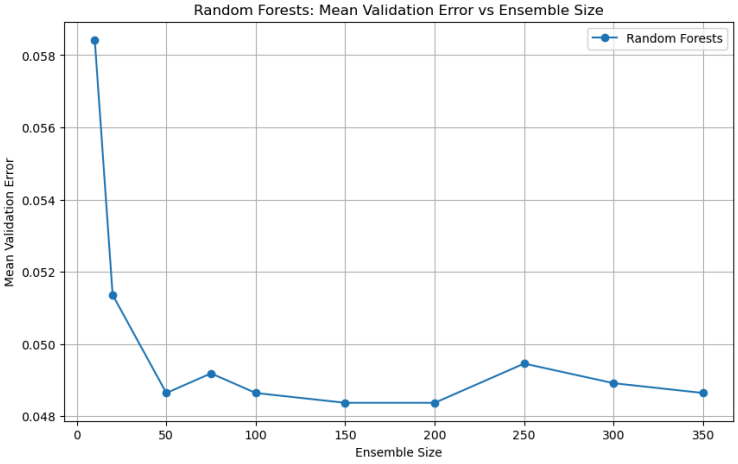


Figure 16: Error evolution of ensemble size through k-fold cross validation

1.3.2 Boosted Decision Trees (AdaBoost)

Next, we performed the same exercise on Adaboost, for boosted decision trees. We use k-fold cross-validation to tune the ensemble size for AdaBoost. By partitioning the training data into *k* folds and iteratively training and validating the model on different subsets, we can identify the ensemble size that minimizes validation error. This approach ensures a robust estimate of the model's generalization performance while avoiding overfitting. This process allows us to balance model complexity and performance, ensuring an efficient and effective AdaBoost classifier.

To visualize the performance of AdaBoost with different ensemble sizes, we constructed a scatter plot, Figure 17, displaying the errors for various ensemble sizes across different folds. In the case of k-fold cross-validation, the data is partitioned into k subsets, and the model is trained on k-1 subsets while being validated on the remaining one. The errors from each fold are then averaged to estimate the model's performance. In the plot, the errors for a particular ensemble size, such as 350, are represented by turquoise crosses. To calculate the average error for this ensemble size, we examine the error for each fold corresponding to the turquoise crosses, sum these individual errors, and divide by 5 (since k = 5). This gives us the mean error for the selected ensemble size, allowing us to evaluate how well AdaBoost generalizes across different data splits for various ensemble sizes.

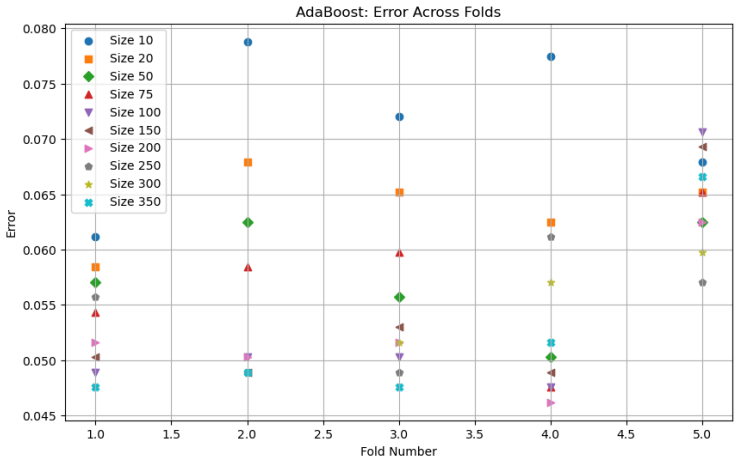


Figure 17: Scatter plot of AdaBoost at for different ensemble sizes at different folds

From the scatter plot, we can analyze the relationship between error evolution and ensemble size for boosted decision trees. Figure 18 demonstrates this relationship, revealing that an ensemble size of 200 is optimal, as it achieved the smallest error during k-fold cross-validation, with a value of 5.24%. The graph also highlights a significant drop in error as the ensemble size increases from the poor performing size of 10, where the error is notably higher. This sharp decline suggests that even modest increases in ensemble size can lead to substantial improvements in model performance.. This observation underscores the importance of selecting an appropriate ensemble size to balance computational efficiency and model accuracy.

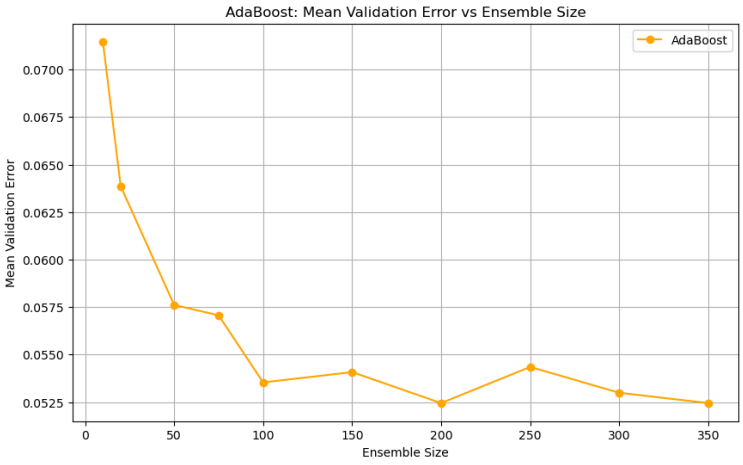


Figure 17: Error vs ensemble size during k-fold cross validation on AdaBoost

In conclusion, our analysis of AdaBoost for boosted decision trees demonstrates the importance of tuning ensemble size to optimize model performance. By employing k-fold cross-validation, we systematically evaluated the relationship between ensemble size and validation error, ensuring a robust estimate of generalization performance

while mitigating overfitting. The scatter plots (Figures 17 and 18) revealed that an ensemble size of 200 yielded the lowest validation error of 0.0524, striking an optimal balance between model complexity and accuracy. The sharp decline in error from smaller ensemble sizes (e.g., 10) to larger ones highlights the significant impact of increasing ensemble size on performance. However, beyond a certain point, such as ensemble sizes greater than 200, the error reduction plateaus, indicating diminishing returns. These findings emphasize the need to carefully select ensemble size to achieve computational efficiency and maintain model effectiveness. Overall, this exercise underscores the power of AdaBoost in improving model performance and the value of systematic tuning techniques like k-fold cross-validation in machine learning workflows.

1.3.3 Comparing Test Results

In this section, we compare the test errors of our tuned AdaBoost and random forest models to evaluate their relative performance on unseen data. Both models were carefully optimized in previous steps—Both through tuning the ensemble size—but it is crucial to note that the test set used here was entirely untouched during those tuning processes. This ensures a fair and unbiased assessment of each model's generalization capabilities. By analyzing the test errors, we aim to determine which model, AdaBoost or random forests, delivers superior predictive accuracy and robustness for the given task.

The test error results reveal that random forests achieved a lower test error of 0.0543 at an ensemble size of 200, compared to boosted decision trees, which had a higher test error of 0.0663 at the same ensemble size. This difference in performance can be attributed to the inherent characteristics of the two methods. Random forests reduce overfitting by averaging multiple decision trees trained on bootstrapped samples and random subsets of features, which enhances their generalization ability. On the other hand, boosted decision trees sequentially focus on correcting the errors of previous models, which can sometimes lead to overfitting, especially if the ensemble size is not carefully tuned or if the data contains noise. The higher test error for boosted trees suggests that the model may have overfitted to the training data, despite achieving strong performance during k-fold cross-validation.

Another factor contributing to the difference in test errors could be the sensitivity of boosted decision trees to noisy or outlier-prone data. While boosting methods like AdaBoost excel at reducing bias by iteratively improving weak learners, they can also amplify errors from noisy data points, leading to poorer generalization on unseen data. In contrast, random forests' ensemble averaging approach makes them more robust to noise, which may explain their superior performance on the test set. Additionally, the random feature selection in random forests introduces diversity among the trees, further improving their ability to generalize.

Both models performed worse on the test data compared to their performance during k-fold cross-validation on the training data. This discrepancy is expected and can be explained by the nature of model evaluation. During k-fold cross-validation, the model is trained and validated on subsets of the training data, which may not fully capture the complexity and

variability of the unseen test data. As a result, the validation error during k-fold cross-validation tends to be an optimistic estimate of the model's true generalization performance. The test set, being entirely independent and unseen during training, provides a more realistic assessment of how the model will perform in practice. The higher test errors for both models highlight the challenge of balancing model complexity and generalization, as well as the importance of using an untouched test set for final evaluation.

In conclusion, the comparison between random forests and boosted decision trees highlights the trade-offs between model complexity, generalization, and robustness. Random forests achieved a lower test error of 0.0543, demonstrating their strength in reducing overfitting and handling noisy data through ensemble averaging and random feature selection. In contrast, boosted decision trees, with a higher test error of 0.0663, may have struggled with overfitting due to their iterative error-correction approach, which can amplify noise and outliers. The discrepancy between test errors and the lower errors observed during k-fold cross-validation underscores the importance of evaluating models on independent test data to obtain a realistic measure of their generalization performance. These findings emphasize the need for careful model tuning and selection, as well as the value of using robust evaluation techniques to ensure models perform well in real-world scenarios. Ultimately, both methods offer powerful tools for predictive modeling, but their effectiveness depends on the specific characteristics of the data and the problem at hand.

1.3.4 Conclusion

In conclusion, k-fold cross-validation proved to be an invaluable tool for tuning and evaluating our models, providing a robust and reliable estimate of their generalization performance. By partitioning the training data into k subsets and iteratively training and validating the models on different folds, we mitigated the risk of overfitting to a single data split and gained a more comprehensive understanding of model behavior. This approach allowed us to systematically tune the ensemble sizes for both random forests and boosted decision trees, identifying optimal configurations that balanced model complexity and performance. The use of k-fold cross-validation not only enhanced the reliability of our results but also highlighted the importance of rigorous evaluation techniques in machine learning workflows. Ultimately, this method ensured that our models were well-suited to generalize to unseen data, laying a strong foundation for their application in real-world scenarios.