Name: Hazem Taha Student ID: 400550393

Assignment 3 Report

Dataset Analysis:

The dataset consists of a total of 58 columns, 57 features and 1 target, extracted from emails, categorized for spam detection. It includes:

- **Word Frequency Attributes (48 features):** These are continuous variables ranging from 0 to 100, representing the percentage of words in the email matching a specific WORD. The calculation is 100 * (number of times the WORD appears in the email) / total number of words in the email. A "word" is defined as any string of alphanumeric characters bounded by non-alphanumeric characters or end-of-string.
- **Character Frequency Attributes (6 features)**: These are continuous variables also ranging from 0 to 100, indicating the percentage of characters in the email that match a specific CHAR. The formula used is 100 * (number of CHAR occurrences) / total characters in the email.
- **Capital Run Length Attributes (3 features)**:
    - capital_run_length_average: A continuous variable starting from 1, representing the average length of uninterrupted sequences of capital letters.
    - capital_run_length_longest: A continuous integer starting from 1, indicating the length of the longest uninterrupted sequence of capital letters.
    - capital_run_length_total: A continuous integer starting from 1, denoting the sum of lengths of uninterrupted sequences of capital letters, which equals the total number of capital letters in the email.

Class Attribute (1 Target): A nominal attribute with values {0,1}, where 1 denotes that the email was considered spam (i.e., unsolicited commercial email) and 0 denotes non-spam.

Records: The dataset contains 4601 rows, 391 of which are duplicates and have been dropped.

Question 1: Training decision trees

To find the optimal number of maximum number of leaves for the decision tree classifier, 5-folds cross validation was done to test tress with [2:400] maximum number of leaves. Figure 1 shows the resulting average misclassification rate (zero/one loss) for each candidate tree.

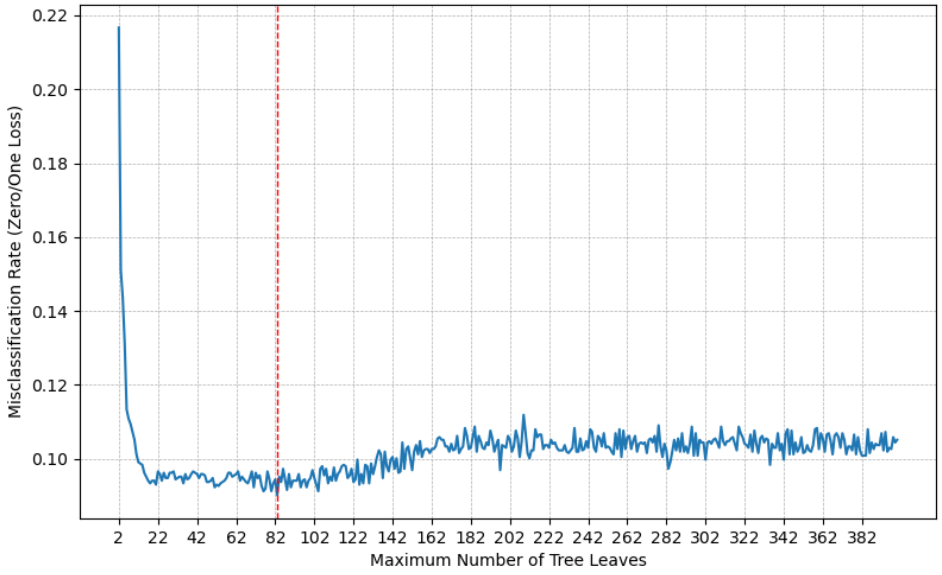

Figure 1: Average Misclassification Rate for Decision Trees using 5-folds Cross-validation.

The best classifier was found to have maximum number of leaves of (83) with average misclassification rate in cross-validation of around (0.09).

The misclassification rate on test set using the best-found decision tree is around (0.084).

Question 2: Training Bagging Classifiers:

The bagging algorithm has been manually implemented and the base classifiers used were Sklearn's implementation of decision trees. Every base classifier has been trained on an N number of resembled datapoint with repetition from the training set (which is of size N). Figure 2 shows the test set misclassification rate of the different sized bagging classifiers.

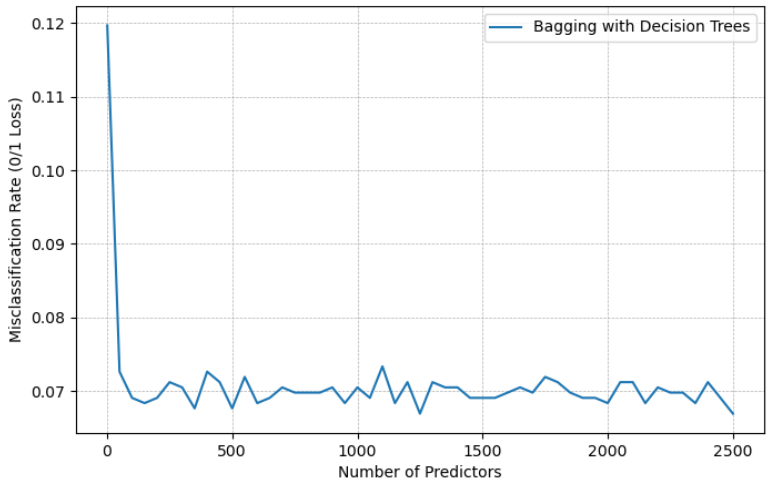

Figure 2: Bagging with Decision Trees Performance

Question 3: Training Random Forest Classifiers:

Using a pre-implemented Random Forest algorithm from Sklearn, figure 3 shows the test set misclassification rate of the different sized RF classifiers.

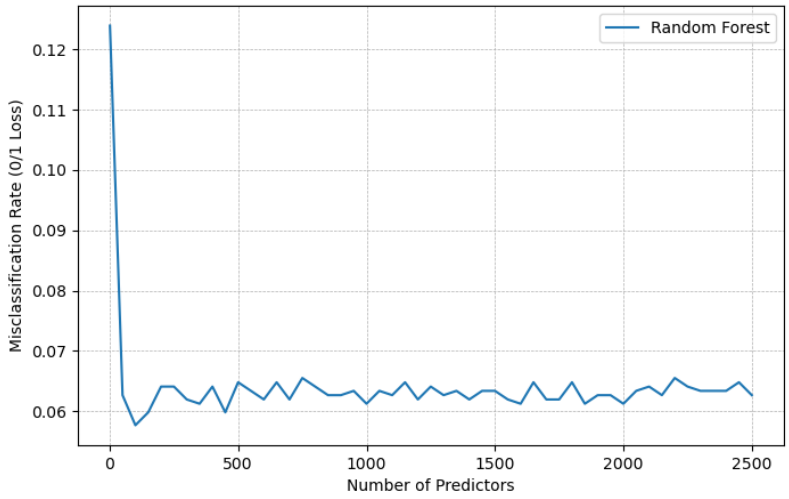

Figure 3: Random Forest Performance.

Questions 4-6 (Adaboost):

Note that all the below implementations of the Adaboost algorithm have been encapsulated within the same function `TrainTest_Adaboost.` This function takes a hyperparameter `baseType` that decides the type of base classifier that builds up the ensemble. However, the algorithm itself is the same for all types of base classifiers.

Note that I implemented two ways to update the weightes (β), the method in the lectures and another one that is more popular and found in the algorithm standard.

$$\beta_{n+1} = \beta_n e^{-(\alpha \times y_{actual} \times y_{predicted})}$$

If $y_{actual} = y_{predicted}$ then β get's decreased, and vice versa. Finally, all the β values have to be normalized by dividing them by the sum of weights.

Question 4: Training Adaboost Classifiers with Decision Stumps:

It's worth noting that implementing a decision stump (and any form of decision trees) for use in Adaboost is different from implementing a typical decision stump. The reason behind that is that the data used to split the stump is weighted and the weights must be taken into account if the stump is to make use of the previous stump's mistakes. There are 2 main differences in the training the stump that I will discuss below:

1) **Weighted impurity index:**

Instead of trying to find the split that maximizes the reduction of an impurity index (ex: entropy, Gini index), the stump should use a weighted impurity index (ex: weighted Gini). Giving more contribution in the impurity index to some points that were previously misclassified than others that were previously correctly classified.

The weighted Gini index for a dataset split is calculated for each split group using the formula:

$$Gini_{Weighted}\,(group) = 1 - \sum_{i=1}^{J} \left(\frac{\sum_{k \in C_i} w_k}{\sum_{k \in S} w_k}\right)^2$$

where:

- J is the number of classes in the group.
- C_i is the set of samples in the group that belong to class i.
- w_k is the weight of the kth sample.
- S is the total set of samples in the group.

Note that the math is equivalent to the one in the lectures, but the lecture's equation assumes equal weights for all samples unlike in Adaboost where each sample has a weight.

The plan was to maximize the reduction in impurity calculated as:

$$Gini_{Weighted}\,(Parent) - (\,Weight_{Group_{left}} Gini_{Weighted}\,(Group_{left}) + Weight_{Group_{right}}\,Gini_{Weighted}\,(Group_{right}))$$

Note that the group's impurity is weighted by the group's proportion of the total weight (which should be always 1). Maximizing the above equation is equivalent to minimizing the sum of weighted impurities of the split groups directly, and that's what I did in my code.

2) **Labeling the split regains (groups):**

The traditional way of labeling the two split groups was to label them with the class with majority votes among the samples in each group. However, this implicitly assumes that all the datapoints in a group have an equal weight which is not the case in Adaboost. The modified version should be as follows: the trained label of a certain group after finding the split is the label that has the maximum sum of weights among the group's samples.

$$argmax_i \sum_{k \in C_i} w_k$$

where:

- C_i is the set of samples in the group that belong to class i.
- w_k is the weight of the kth sample.

Figure 4 shows the test set misclassification rate of the different sized Adaboost classifiers with decision stumps as base predictors.

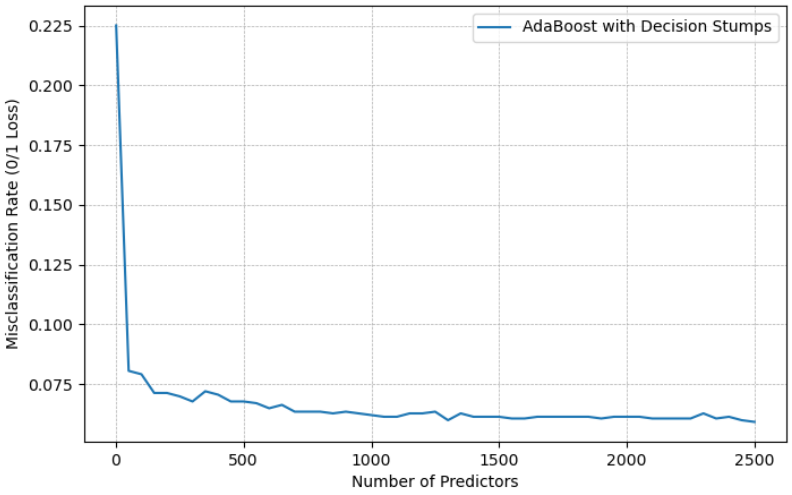

Figure 4: Adaboost with Decision Stumps Performance.

Note that in fitting a decision stump, I defined a hyperparameter `num_features_to_consider` which decides the size of a random subset of features to consider while searching for the best split because searching in all the features can take very long time. If the hyperparameter is set to `None`, all features will be considered. In my final solution, I used it as `None,` and I let the code run for around 16 hours to get accurate results.

Question 5-6: Training Adaboost Classifiers with Decision Trees

Using my implementation of the Adaboost algorithm with a Decision Tree base predictor from Sklearn, figure 5 shows the test set misclassification rate using decision trees with maximum 10 leaves and figure 6 shows the same for unrestricted trees.

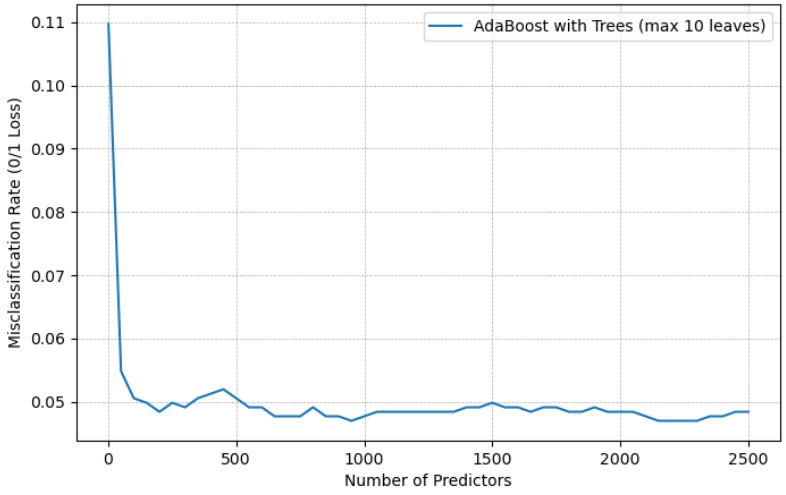

Figure 5: Adaboost with Trees (max 10 leaves) Performance.

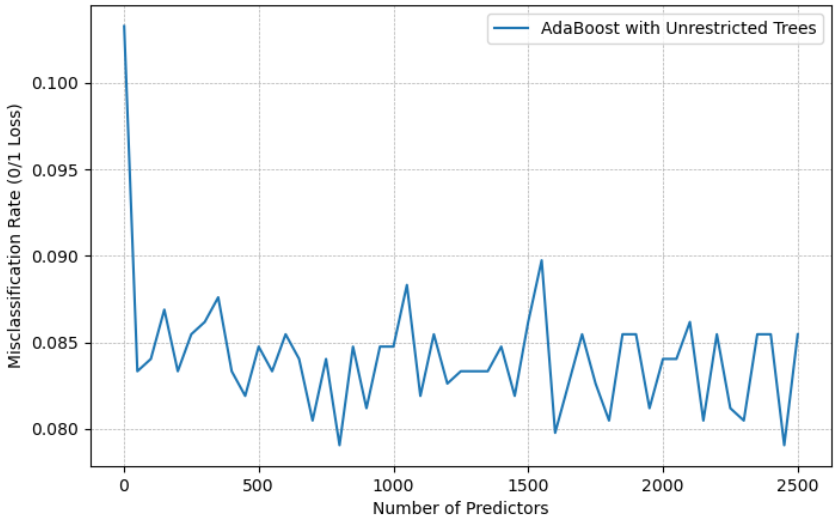

Figure 6: Adaboost with Trees Performance.

Final Comparison:

Figure 7 shows the final comparison between all the trained classifiers. The below can be concluded from the figure:

- All ensembles start with a higher error rate when the number of predictors is very low, this is due to the models not having enough predictors to make accurate predictions.
- As the number of predictors increases, the misclassification rates for all methods seem to decrease and then stabilize. This stabilization indicates that adding more predictors beyond a certain point does not significantly improve performance.
- AdaBoost with restricted number of ten leaves has the lowest misclassification rates across almost all numbers of predictors, and AdaBoost with unrestricted number of leaves has the highest misclassification rates nearly equal to training a normal decision tree classifier. This can be explained by the fact that unrestricted trees can grow very complex and may fit the noise in the training data, leading to overfitting. Also, AdaBoost, by definition, works by combining multiple weak learners to create a strong classifier. If each individual tree is too complex, the boosting process may not be as effective. Therefore, I will not consider Adaboost with unrestricted trees as a valid Adaboost implementation in my comparison.
- AdaBoost with decision stumps, although didn't achieve the best overall performance in the given range, yet it shows the best learning curve that keeps decreasing in loss with adding more predictors. It's a solid example of the foundation behind Adaboost that combining multiple weak learners to create a strong classifier. It takes longer time than AdaBoost with restricted number of ten leaves because the predictors here are weaker.
- It's obvious that all "valid" boosting implementations have got better performance at large number of predictors than the bagging and random forests. However, for smaller numbers of predictors, bagging and random forests gave better results than Adaboost with decision stumps. Again, this is due to the fact that decision stumps are too weak, and they need large amounts of predictors to get better performance. This can be proved by looking at Adaboost with ten leaves which converged more quickly and gave better performance than bagging and random forests at very small among of predictors.
- It's obvious that random forests gave better performance than bagging across the entire range of number of predictors. Making bagging the second worst method used after the single decision tree.

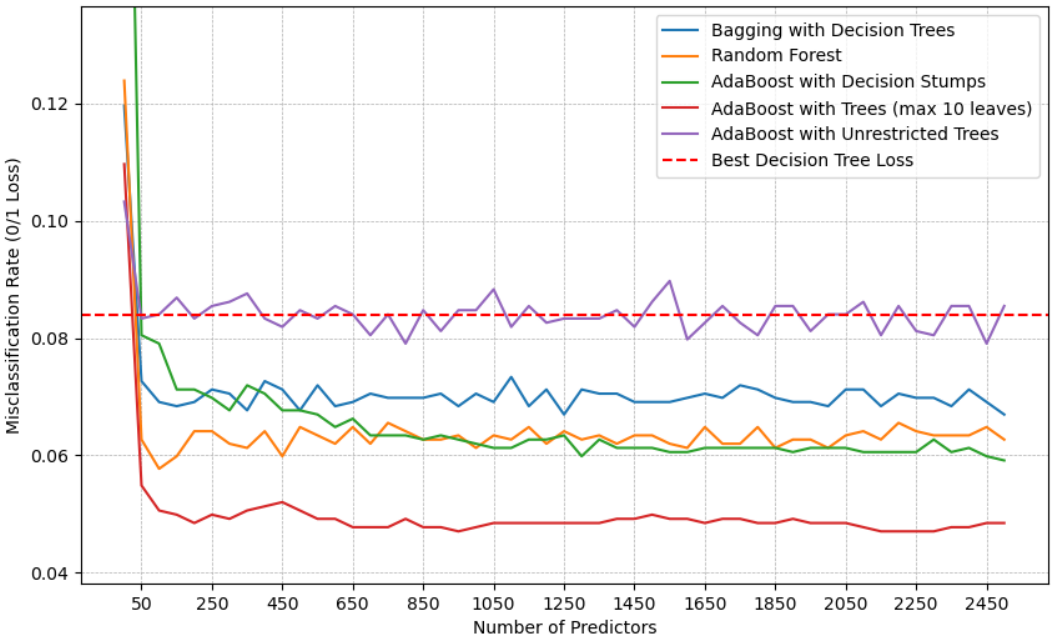

Figure 7: Classifier Performance Comparison.