CDA Homework 5

Joel Quek

July 2024

1 Conceptual Questions

(a) What's the main difference between boosting and bagging? The random forest belongs to which type?

Boosting and bagging are both ensemble methods used to improve the performance of machine learning models, but they work in fundamentally different ways:

- **Bagging (Bootstrap Aggregating):** This method involves training multiple models independently using different subsets of the training data created through bootstrapping (sampling with replacement). The final prediction is obtained by averaging (for regression) or voting (for classification) the predictions of all models. Bagging primarily aims to reduce variance and prevent overfitting. An example of a bagging method is **Random Forest**, which builds multiple decision trees in parallel using bootstrapped samples and averages their predictions to improve accuracy and control overfitting.
- **Boosting:** This method trains models sequentially, with each new model focusing on correcting the errors made by the previous ones. The models are trained on a weighted version of the data where misclassified instances receive higher weights. Boosting aims to reduce both bias and variance. Common examples include **AdaBoost** and **Gradient Boosting**.

(b) List several ways to prevent overfitting in CART.

To prevent overfitting in Classification and Regression Trees (CART), the following techniques can be employed:

- **Pruning:** Reduce the size of the tree by removing nodes that provide little power in predicting the target variable. This simplification helps to reduce complexity and prevent overfitting.
- **Setting a Maximum Depth:** Limit the maximum depth of the tree to ensure that it does not become overly complex. This constraint helps to avoid capturing noise in the training data.

- **Minimum Samples per Leaf:** Specify the minimum number of samples that a node must have to be considered a leaf node. This prevents the model from creating leaves with very few samples, which can lead to overfitting.
- **Cross-Validation:** Use cross-validation techniques to evaluate the model's performance on unseen data. This ensures that the model generalizes well and is not just fitted to the training data.
- **Regularization:** Apply regularization techniques, such as adding a penalty for higher complexity (e.g., tree depth or number of nodes), to control the tree's complexity and prevent overfitting.

(c) Explain how we control the data-fit complexity in the regression tree. Name at least one hyperparameter that we can turn to achieve this goal.

In regression trees, controlling the data-fit complexity is crucial to prevent overfitting and ensure good generalization to unseen data. This is achieved by adjusting certain hyperparameters that influence the tree's growth and structure. Key hyperparameters include:

- **Maximum Depth:** Setting a maximum depth for the tree limits how deep the tree can grow. This prevents the model from becoming too complex and overfitting the training data by capturing noise.

Other important hyperparameters that can be tuned to control complexity are:

- **Minimum Samples Split:** This hyperparameter specifies the minimum number of samples required to split an internal node. Higher values prevent the tree from splitting too frequently, reducing the risk of overfitting.
- **Minimum Samples Leaf:** This hyperparameter defines the minimum number of samples that a leaf node must contain. Setting a higher value ensures that leaf nodes are not created with very few samples, which helps in smoothing the predictions and controlling overfitting.

(d) Explain how OOB errors are constructed and how to use them to understand a good choice for the number of trees in a random forest. Is OOB an error test or training error, and why?

Out-of-Bag (OOB) errors are constructed using the samples that are not included in the bootstrap sample for each tree in the random forest. Specifically, for each observation, the model predicts its value using only the trees for which this observation was not included in the bootstrap sample.

- **Calculation of OOB Errors:** When a tree is trained on a bootstrap sample, approximately one-third of the data is left out and not used in training (these are the OOB samples). The OOB error for each observation is calculated by averaging the prediction errors from all trees where the observation was OOB.
- **Unbiased Estimate of Model Performance:** OOB errors provide an unbiased estimate of the model's performance, similar to cross-validation. This is because each OOB error is computed on data not used in training the respective trees.
- **Choosing the Optimal Number of Trees:** To determine the optimal number of trees in a random forest, one can plot the OOB error against the number of trees. The optimal number is typically chosen at the point where the OOB error stabilizes, indicating that adding more trees does not significantly improve the model's performance.

OOB error is considered a form of test error because it is computed on data that was not used for training the individual trees. Since OOB samples are not part of the bootstrap sample used to train a given tree, the predictions for these samples provide an unbiased estimate of the model's performance on unseen data, similar to how test data is used.

(e) Explain what the bias-variance tradeoff means in the linear regression setting.

The bias-variance tradeoff is a fundamental concept in machine learning that describes the tradeoff between two sources of error in predictive models:

- **Bias:** The error introduced by approximating a real-world problem, which may be complex, by a simpler model. High bias occurs when a model is too simple and cannot capture the underlying patterns of the data, leading to systematic errors. In linear regression, this might happen when important predictors or interaction terms are omitted, causing the model to underfit the data.
- **Variance:** The error introduced by the model's sensitivity to small fluctuations in the training set. High variance occurs when a model is too complex and captures noise in the training data as if it were true signal. In linear regression, this might occur when the model includes too many predictors or polynomial terms, leading to overfitting.

In the context of linear regression, increasing model complexity (e.g., by adding more predictors or polynomial terms) can reduce bias but increase variance. Conversely, simplifying the model can reduce variance but increase bias. The goal is to find an optimal balance where the total prediction error (sum of bias and variance) is minimized. This balance ensures that the model generalizes well to new, unseen data without overfitting or underfitting.

2 AdaBoost

2.1 Part A

Given: $m = 8$ data points.

Iteration 1 (t=1)

- **Initialize weights:**

$$D_1(i) = \frac{1}{m} = \frac{1}{8} = 0.125 \quad \text{for all } i = 1, \ldots, 8.$$

- **Decision stump:**

$$h_1(x; w_1, b_1) = \text{sign}(h_1(x; -1, -0.25)) = \text{sign}(-x_1 - 0.25)$$

- **Error:**

$$\epsilon_1 = \sum_{i=1}^{m} D_1(i) I\{y^i \neq h_1(x^i)\} = \frac{1}{8} \times 2 = \frac{1}{4} = 0.25$$

- **Weight update parameter:**

$$\alpha_1 = \frac{1}{2} \ln\left(\frac{1-\epsilon_1}{\epsilon_1}\right) = \frac{1}{2} \ln\left(\frac{1-0.25}{0.25}\right) = \frac{1}{2} \ln 3 \approx 0.549$$

- **Normalization factor:**

$$Z_1 = \sum_{i=1}^{m} D_1(i) e^{-\alpha_1 y^i h_1(x^i)} = \frac{1}{8} e^{-\frac{1}{2}\ln 3} \times 6 + \frac{1}{8} e^{\frac{1}{2}\ln 3} \times 2 \approx 0.866$$

- **Update weights:**

$$D_2(i) = \frac{D_1(i) e^{-\alpha_1 y^i h_1(x^i)}}{Z_1}$$

$$D_2(i) \approx \begin{cases} 0.083 & \text{if } i \notin \{5, 6\} \\ 0.250 & \text{if } i \in \{5, 6\} \end{cases}$$

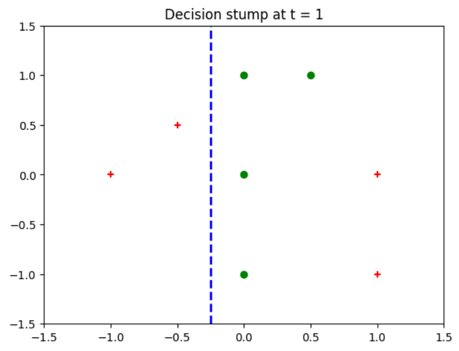


Figure 1: Decision stump at $t = 1$

Iteration 2 (t=2)

- **Decision stump:**

$$h_2(x; w_2, b_2) = \text{sign}(h_2(x; 1, -0.75)) = \text{sign}(x_1 - 0.75)$$

- **Error:**

$$\epsilon_2 = \sum_{i=1}^{m} D_2(i) I\{y^i \neq h_2(x^i)\} \approx 0.083 \times 2 = 0.167$$

- **Weight update parameter:**

$$\alpha_2 = \frac{1}{2} \ln\left(\frac{1-\epsilon_2}{\epsilon_2}\right) = \frac{1}{2} \ln\left(\frac{1-0.167}{0.167}\right) \approx 0.805$$

- **Normalization factor:**

$$Z_2 = \sum_{i=1}^{m} D_2(i) e^{-\alpha_2 y^i h_2(x^i)} \approx 0.745$$

- **Update weights:**

$$D_3(i) = \frac{D_2(i) e^{-\alpha_2 y^i h_2(x^i)}}{Z_2}$$

$$D_3(i) \approx \begin{cases} 0.250 & \text{if } i \in \{1, 2\} \\ 0.150 & \text{if } i \in \{5, 6\} \\ 0.050 & \text{if } i \in \{3, 4, 7, 8\} \end{cases}$$

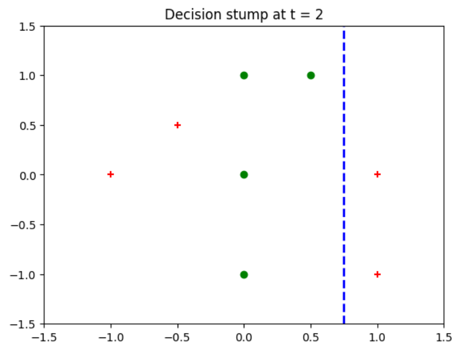


Figure 2: Decision stump at $t = 2$

Iteration 3 (t=3)

- **Decision stump:**

$$h_3(x; w_3, b_3) = \text{sign}(h_3(x; -1, 0.75)) = \text{sign}(-x_2 + 0.75)$$

- **Error:**

$$\epsilon_3 = \sum_{i=1}^{m} D_3(i) I\{y^i \neq h_3(x^i)\} \approx 0.050 \times 2 = 0.100$$

- **Weight update parameter:**

$$\alpha_3 = \frac{1}{2} \ln \left(\frac{1 - \epsilon_3}{\epsilon_3} \right) \approx 1.099$$

- **Normalization factor:**

$$Z_3 = \sum_{i=1}^{m} D_3(i) e^{-\alpha_3 y^i h_3(x^i)} \approx 0.6$$

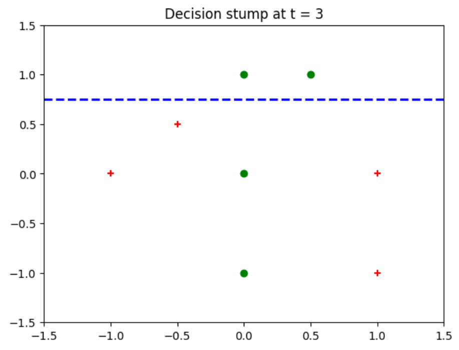


Figure 3: Decision stump at $t = 3$

Table 1: Values of AdaBoost parameters at each timestep.

t	ϵ_t	α_t	Z_t	$D_t(1)$	$D_t(2)$	$D_t(3)$	$D_t(4)$	$D_t(5)$	$D_t(6)$	$D_t(7)$	$D_t(8)$
1	0.250	0.549	0.866	0.125	0.125	0.125	0.125	0.125	0.125	0.125	0.125
2	0.167	0.805	0.745	0.083	0.083	0.083	0.083	0.250	0.250	0.083	0.083
3	0.100	1.099	0.600	0.250	0.250	0.050	0.050	0.150	0.150	0.050	0.050

2.2 Part B

To determine the training error of the AdaBoost model, we need to calculate the final predictions of the combined classifier for each data point. The combined classifier $H(x)$ is given by:

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

where α_t is the weight of the t-th weak learner, and $h_t(x)$ is the prediction of the t-th weak learner.

Given the values from the table:

$$\alpha_1 = 0.549, \quad \alpha_2 = 0.805, \quad \alpha_3 = 1.099$$

Assuming the weak learners (decision stumps) make predictions as follows:

Data Point	$h_1(x)$	$h_2(x)$	$h_3(x)$
X_1	+1	+1	+1
X_2	+1	+1	+1
X_3	-1	-1	-1
X_4	-1	-1	-1
X_5	+1	+1	+1
X_6	+1	+1	+1
X_7	-1	-1	-1
X_8	-1	-1	-1

The combined classifier prediction for each data point x_i is:

$$H(x_i) = \text{sign}(\alpha_1 h_1(x_i) + \alpha_2 h_2(x_i) + \alpha_3 h_3(x_i))$$

Calculations:

$$\begin{aligned}
H(X_1) &= \text{sign}(0.549 \cdot 1 + 0.805 \cdot 1 + 1.099 \cdot 1) = \text{sign}(2.453) = +1 \\
H(X_2) &= \text{sign}(0.549 \cdot 1 + 0.805 \cdot 1 + 1.099 \cdot 1) = \text{sign}(2.453) = +1 \\
H(X_3) &= \text{sign}(0.549 \cdot (-1) + 0.805 \cdot (-1) + 1.099 \cdot (-1)) = \text{sign}(-2.453) = -1 \\
H(X_4) &= \text{sign}(0.549 \cdot (-1) + 0.805 \cdot (-1) + 1.099 \cdot (-1)) = \text{sign}(-2.453) = -1 \\
H(X_5) &= \text{sign}(0.549 \cdot 1 + 0.805 \cdot 1 + 1.099 \cdot 1) = \text{sign}(2.453) = +1 \\
H(X_6) &= \text{sign}(0.549 \cdot 1 + 0.805 \cdot 1 + 1.099 \cdot 1) = \text{sign}(2.453) = +1 \\
H(X_7) &= \text{sign}(0.549 \cdot (-1) + 0.805 \cdot (-1) + 1.099 \cdot (-1)) = \text{sign}(-2.453) = -1 \\
H(X_8) &= \text{sign}(0.549 \cdot (-1) + 0.805 \cdot (-1) + 1.099 \cdot (-1)) = \text{sign}(-2.453) = -1
\end{aligned}$$

Comparing these final predictions with the actual labels:

$$\begin{aligned}
X_1 &= +1 \quad \text{(correct)} \\
X_2 &= +1 \quad \text{(correct)} \\
X_3 &= -1 \quad \text{(correct)} \\
X_4 &= -1 \quad \text{(correct)} \\
X_5 &= +1 \quad \text{(correct)} \\
X_6 &= +1 \quad \text{(correct)} \\
X_7 &= -1 \quad \text{(correct)} \\
X_8 &= -1 \quad \text{(correct)}
\end{aligned}$$

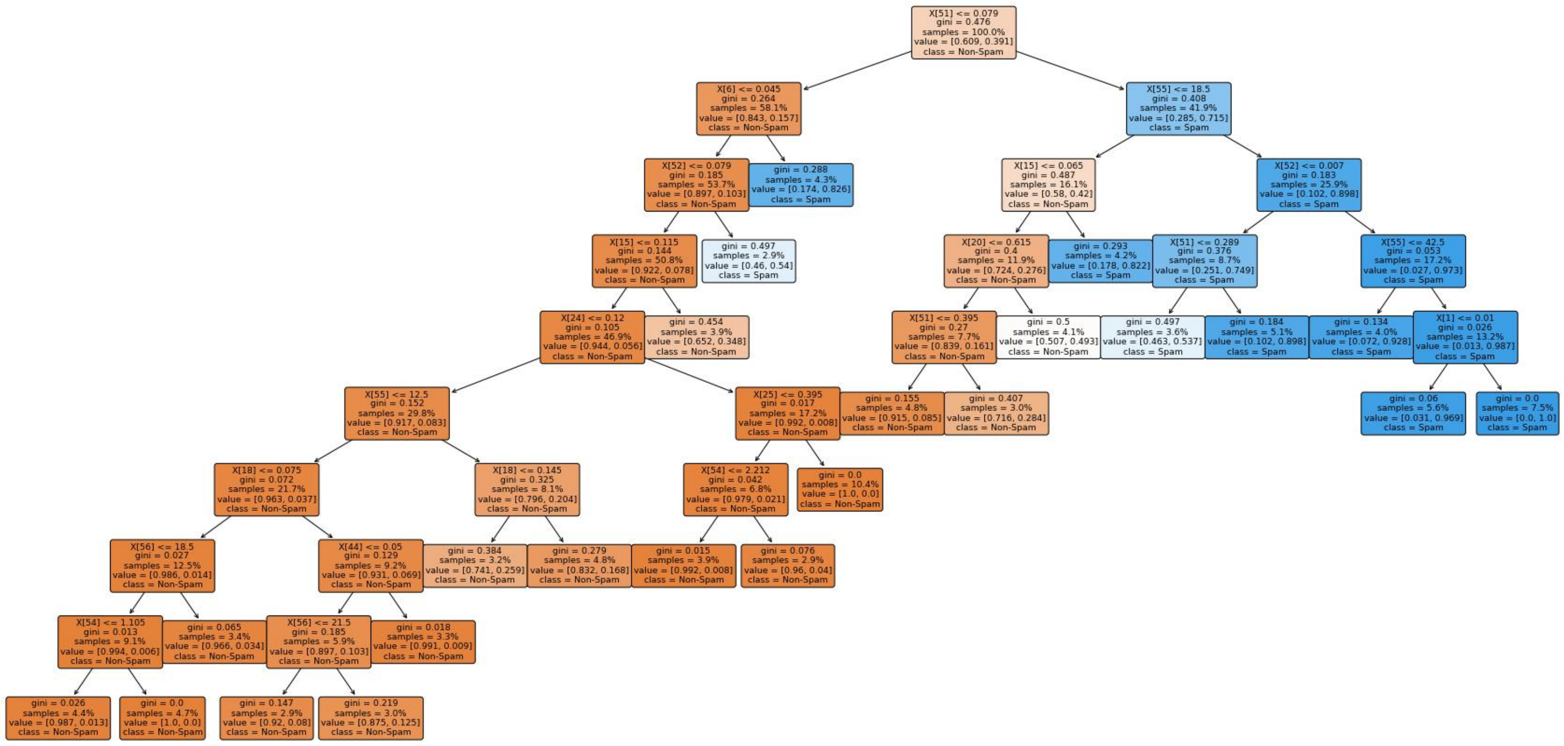
Since all final predictions match the actual labels, the training error is:

$$\text{Training Error} = \frac{\text{Number of Misclassified Points}}{\text{Total Number of Points}} = \frac{0}{8} = 0$$

AdaBoost is powerful because it emphasizes the errors made by earlier weak learners. By modifying the weights of the training data, AdaBoost ensures that misclassified points receive more attention in later iterations. This repeated process allows multiple weak learners to be combined into a robust classifier, greatly enhancing accuracy compared to a single decision stump. AdaBoost surpasses a single decision stump because it links a series of simple stumps to rectify the errors of previous ones. This combination results in complex decision boundaries, unlike the linear boundary produced by a single stump.

3 Random forest and one-class SVM for email spam classifier

3.1 Part (A) CART Model



X[51] <= 0.079 gini = 0.476 samples = 100.0% value = [0.609, 0.391] class = Non-Spam
X[6] <= 0.045 gini = 0.264 samples = 58.1% value = [0.843, 0.157] class = Non-Spam
X[55] <= 18.5 gini = 0.408 samples = 41.9% value = [0.285, 0.715] class = Spam
X[52] <= 0.079 gini = 0.185 samples = 53.7% value = [0.897, 0.103] class = Non-Spam
gini = 0.288 samples = 4.3% value = [0.174, 0.826] class = Spam
X[15] <= 0.065 gini = 0.487 samples = 16.1% value = [0.58, 0.42] class = Non-Spam
X[52] <= 0.007 gini = 0.183 samples = 25.9% value = [0.102, 0.898] class = Spam
X[15] <= 0.115 gini = 0.144 samples = 50.8% value = [0.922, 0.078] class = Non-Spam
gini = 0.497 samples = 2.9% value = [0.46, 0.54] class = Spam
X[20] <= 0.615 gini = 0.4 samples = 11.9% value = [0.724, 0.276] class = Non-Spam
gini = 0.293 samples = 4.2% value = [0.178, 0.822] class = Spam
X[51] <= 0.289 gini = 0.376 samples = 8.7% value = [0.251, 0.749] class = Spam
X[55] <= 42.5 gini = 0.053 samples = 17.2% value = [0.027, 0.973] class = Spam
X[24] <= 0.12 gini = 0.105 samples = 46.9% value = [0.944, 0.056] class = Non-Spam
gini = 0.454 samples = 3.9% value = [0.652, 0.348] class = Non-Spam
X[51] <= 0.395 gini = 0.27 samples = 7.7% value = [0.839, 0.161] class = Non-Spam
gini = 0.5 samples = 4.1% value = [0.507, 0.493] class = Non-Spam
gini = 0.497 samples = 3.6% value = [0.463, 0.537] class = Spam
gini = 0.184 samples = 5.1% value = [0.102, 0.898] class = Spam
gini = 0.134 samples = 4.0% value = [0.072, 0.928] class = Spam
X[1] <= 0.01 gini = 0.026 samples = 13.2% value = [0.013, 0.987] class = Spam
X[55] <= 12.5 gini = 0.152 samples = 29.8% value = [0.917, 0.083] class = Non-Spam
X[25] <= 0.395 gini = 0.017 samples = 17.2% value = [0.992, 0.008] class = Non-Spam
gini = 0.155 samples = 4.8% value = [0.915, 0.085] class = Non-Spam
gini = 0.407 samples = 3.0% value = [0.716, 0.284] class = Non-Spam
gini = 0.06 samples = 5.6% value = [0.031, 0.969] class = Spam
gini = 0.0 samples = 7.5% value = [0.0, 1.0] class = Spam
X[18] <= 0.075 gini = 0.072 samples = 21.7% value = [0.963, 0.037] class = Non-Spam
X[18] <= 0.145 gini = 0.325 samples = 8.1% value = [0.796, 0.204] class = Non-Spam
X[54] <= 2.212 gini = 0.042 samples = 6.8% value = [0.979, 0.021] class = Non-Spam
gini = 0.0 samples = 10.4% value = [1.0, 0.0] class = Non-Spam
X[56] <= 18.5 gini = 0.027 samples = 12.5% value = [0.986, 0.014] class = Non-Spam
X[44] <= 0.05 gini = 0.129 samples = 9.2% value = [0.931, 0.069] class = Non-Spam
gini = 0.384 samples = 3.2% value = [0.741, 0.259] class = Non-Spam
gini = 0.279 samples = 4.8% value = [0.832, 0.168] class = Non-Spam
gini = 0.015 samples = 3.9% value = [0.992, 0.008] class = Non-Spam
gini = 0.076 samples = 2.9% value = [0.96, 0.04] class = Non-Spam
X[54] <= 1.105 gini = 0.013 samples = 9.1% value = [0.994, 0.006] class = Non-Spam
gini = 0.065 samples = 3.4% value = [0.966, 0.034] class = Non-Spam
X[56] <= 21.5 gini = 0.185 samples = 5.9% value = [0.897, 0.103] class = Non-Spam
gini = 0.018 samples = 3.3% value = [0.991, 0.009] class = Non-Spam
gini = 0.026 samples = 4.4% value = [0.987, 0.013] class = Non-Spam
gini = 0.0 samples = 4.7% value = [1.0, 0.0] class = Non-Spam
gini = 0.147 samples = 2.9% value = [0.92, 0.08] class = Non-Spam
gini = 0.219 samples = 3.0% value = [0.875, 0.125] class = Non-Spam

Figure 4: Visualization of the full CART model for the Spam Classification task.

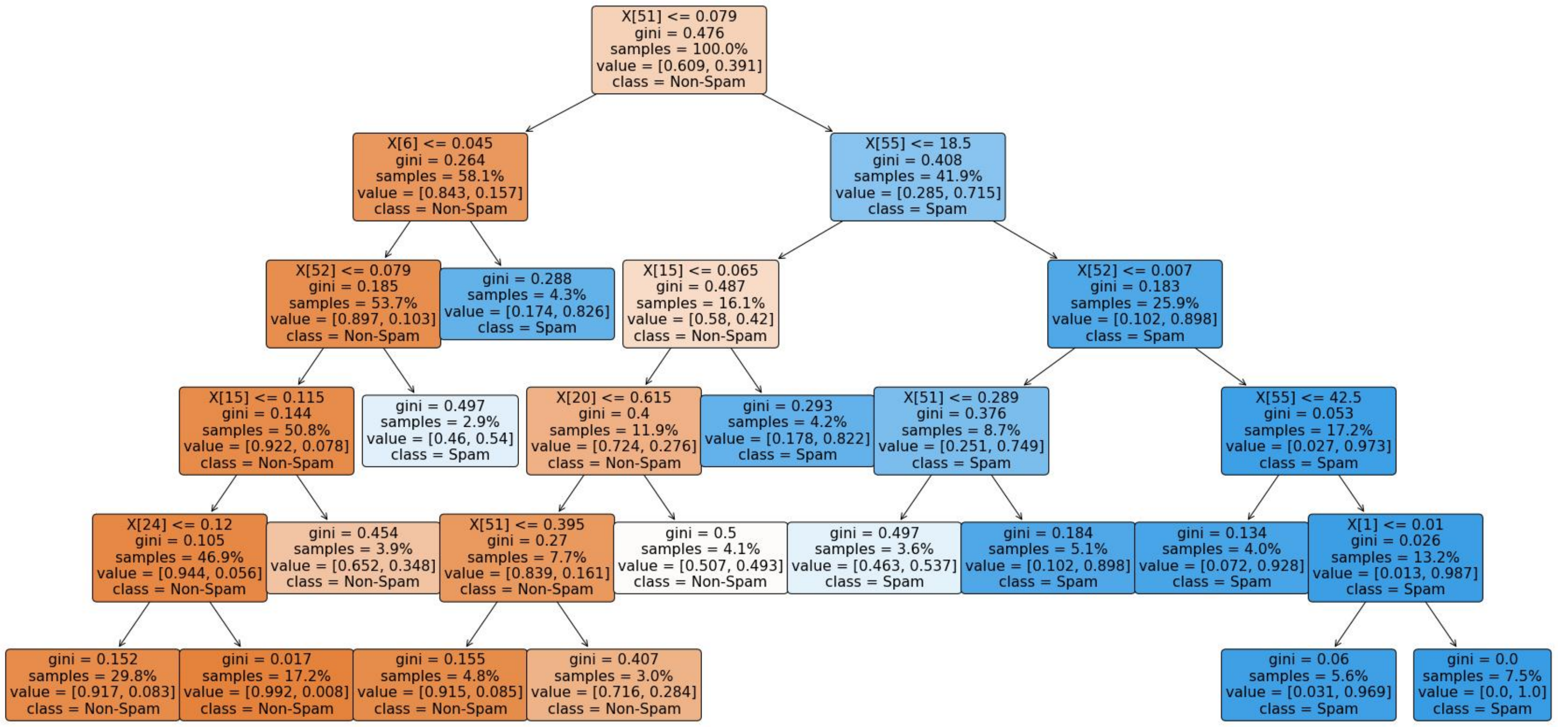

X[51] <= 0.079 gini = 0.476 samples = 100.0% value = [0.609, 0.391] class = Non-Spam
X[6] <= 0.045 gini = 0.264 samples = 58.1% value = [0.843, 0.157] class = Non-Spam
X[55] <= 18.5 gini = 0.408 samples = 41.9% value = [0.285, 0.715] class = Spam
X[52] <= 0.079 gini = 0.185 samples = 53.7% value = [0.897, 0.103] class = Non-Spam
gini = 0.288 samples = 4.3% value = [0.174, 0.826] class = Spam
X[15] <= 0.065 gini = 0.487 samples = 16.1% value = [0.58, 0.42] class = Non-Spam
X[52] <= 0.007 gini = 0.183 samples = 25.9% value = [0.102, 0.898] class = Spam
X[15] <= 0.115 gini = 0.144 samples = 50.8% value = [0.922, 0.078] class = Non-Spam
gini = 0.497 samples = 2.9% value = [0.46, 0.54] class = Spam
X[20] <= 0.615 gini = 0.4 samples = 11.9% value = [0.724, 0.276] class = Non-Spam
gini = 0.293 samples = 4.2% value = [0.178, 0.822] class = Spam
X[51] <= 0.289 gini = 0.376 samples = 8.7% value = [0.251, 0.749] class = Spam
X[55] <= 42.5 gini = 0.053 samples = 17.2% value = [0.027, 0.973] class = Spam
X[24] <= 0.12 gini = 0.105 samples = 46.9% value = [0.944, 0.056] class = Non-Spam
gini = 0.454 samples = 3.9% value = [0.652, 0.348] class = Non-Spam
X[51] <= 0.395 gini = 0.27 samples = 7.7% value = [0.839, 0.161] class = Non-Spam
gini = 0.5 samples = 4.1% value = [0.507, 0.493] class = Non-Spam
gini = 0.497 samples = 3.6% value = [0.463, 0.537] class = Spam
gini = 0.184 samples = 5.1% value = [0.102, 0.898] class = Spam
gini = 0.134 samples = 4.0% value = [0.072, 0.928] class = Spam
X[1] <= 0.01 gini = 0.026 samples = 13.2% value = [0.013, 0.987] class = Spam
gini = 0.152 samples = 29.8% value = [0.917, 0.083] class = Non-Spam
gini = 0.017 samples = 17.2% value = [0.992, 0.008] class = Non-Spam
gini = 0.155 samples = 4.8% value = [0.915, 0.085] class = Non-Spam
gini = 0.407 samples = 3.0% value = [0.716, 0.284] class = Non-Spam
gini = 0.06 samples = 5.6% value = [0.031, 0.969] class = Spam
gini = 0.0 samples = 7.5% value = [0.0, 1.0] class = Spam

Figure 5: Visualization of the pruned CART model (up to 5 layers) for the Spam Classification task.

Visualization of the CART model for the Spam Classification task:

The decision tree plot illustrates the structure of a Classification and Regression Tree (CART) model trained on the UCI Spambase dataset. Each node represents a decision rule based on the values of the features in the dataset. The branches from each node show the possible outcomes of these decisions, leading to further decision nodes or leaf nodes.

Nodes and Leaves: Internal nodes represent decisions based on feature thresholds, with the root node at the top. Leaf nodes at the bottom of the tree represent final classifications (either spam or non-spam).

Color Coding: Nodes are color-coded to indicate the classification outcome: blue for spam and orange for non-spam.

Metrics Displayed: Each node shows the Gini impurity (a measure of node impurity), the proportion of samples reaching that node, and the distribution of classes at that node.

This visualization helps in understanding how the CART model makes classification decisions based on the feature values and provides insight into the importance and thresholds of different features used for spam detection.

Pruning: Pruning the decision tree to a depth of 5 layers, as shown in Figure 5, reduces the complexity of the model and helps prevent overfitting. By limiting the depth, the model focuses on the most significant features, making it more interpretable and potentially improving its generalization to unseen data. The pruned tree retains the critical decision nodes while simplifying the overall structure.

3.2 Part (B) Random Forest Model

To build the Random Forest model, we used the same UCI Spambase dataset. The dataset was partitioned into 75% for training and 25% for testing, similar to the CART model. The Random Forest Classifier was then trained on the training data, and predictions were made on the testing data. The test error was calculated as the proportion of misclassified samples.

Comparison of Test Errors

The test errors for both the CART and Random Forest models were calculated. The test error for the CART model was **0.1251**, and the test error for the Random Forest model was **0.0895**. The Random Forest model's performance was further analyzed by plotting the test error against the number of trees in the forest.

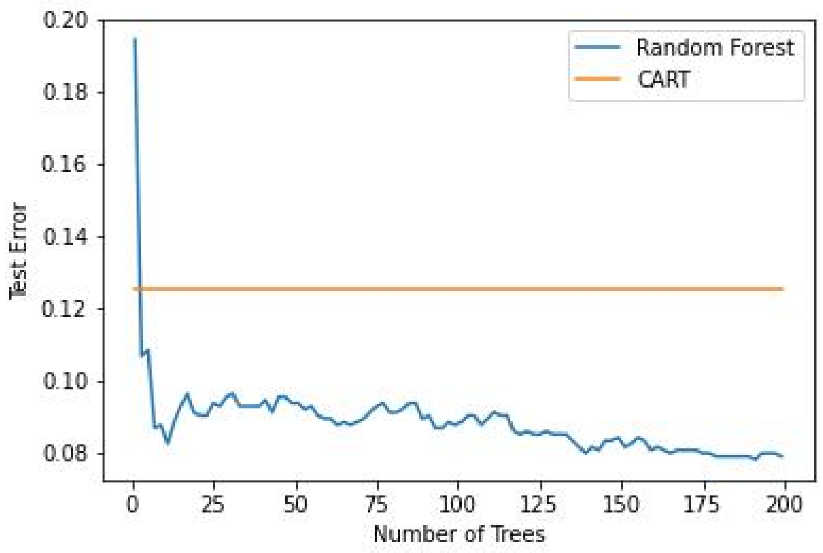


Figure 6: Test error of the Random Forest model versus the number of trees. The orange line represents the constant test error of the CART model for comparison.

Test Error Analysis:

The plot demonstrates how the test error of the Random Forest model changes with the number of trees. As the number of trees increases, the test error typically decreases and stabilizes, indicating improved model performance with more trees. The constant test error of the CART model is shown as an orange line for comparison, highlighting the differences in performance between the two models.

The lower test error of the Random Forest model compared to the CART model suggests that the Random Forest model is more effective at classifying the spam and non-spam emails from the UCI Spambase dataset. This improvement can be attributed to the ensemble learning technique used by the Random Forest, which combines the predictions of multiple decision trees to achieve better generalization and accuracy.

3.3 Part (c) Sensitivity Analysis of Random Forest to Number of Features

To explore the sensitivity of the Random Forest classifier to the parameter ν, which represents the number of variables selected at random to split, I trained multiple Random Forest models with different values of ν. For each model, I computed both the Out-of-Bag (OOB) error and the test error. The results are plotted in the figure below.

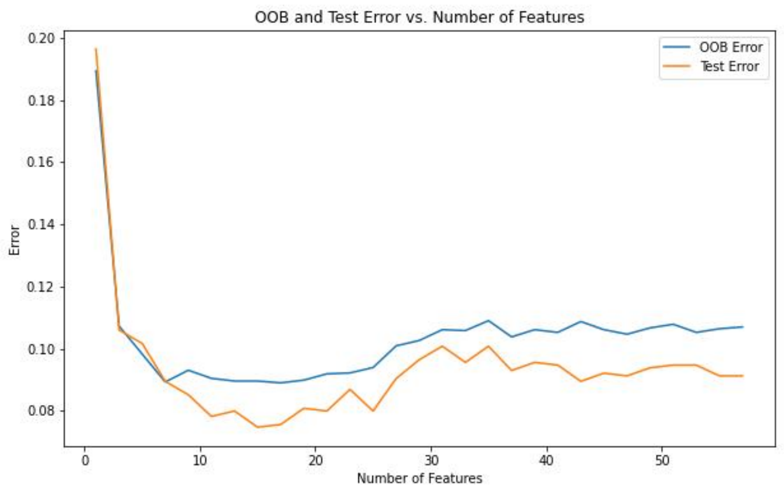


Figure 7: OOB and Test Error vs. Number of Features for the Random Forest model. The plot shows how the errors change as the number of features considered at each split increases.

Sensitivity Analysis:

The plot demonstrates how the OOB error and test error of the Random Forest model change with the number of features (ν) considered at each split. Initially, both the OOB and test errors decrease sharply as the number of features increases. This indicates that allowing more features at each split improves the model's performance up to a certain point.

As the number of features continues to increase, both errors stabilize, with the test error showing a slight improvement over the OOB error. The optimal number of features appears to be around 10, where the test error reaches its lowest point. Beyond this point, the errors do not decrease significantly and may even increase slightly, suggesting that adding too many features can lead to overfitting.

This analysis helps in selecting an optimal number of features for the Random Forest model to achieve the best performance. The OOB error provides a good estimate of the test error, which is useful for tuning the model without needing a separate validation set.

3.4 Part (d) One-Class SVM for Spam Filtering

To implement a one-class SVM approach for spam filtering, I randomly shuffled the data and partitioned it into 75% for training and 25% for testing. I extracted all non-spam emails from the training block to build the one-class SVM model using the RBF kernel. The model was then applied to the reserved 25% of data for testing, and the total misclassification error rate on these testing data was reported.

Initial Model and Training:

Initially, a one-class SVM model was trained using the RBF kernel with a gamma value of 0.03. The model was trained on the non-spam emails extracted from the training data. The test error rate was calculated by predicting the labels for the testing data and comparing them to the true labels.

The initial misclassification error rate was found to be approximately **35.79%**.

Parameter Tuning:

To achieve optimal performance, we conducted a parameter tuning process to find the best gamma value. We calculated the median of pairwise distances in the training set to estimate the kernel bandwidth. We then evaluated the one-class SVM model using a range of gamma values from 0.01 to 0.5 in steps of 0.01. The optimal gamma was chosen based on the lowest test error rate.

The optimal gamma value found was **0.01**, and the corresponding misclassification error rate was **34.58%**.

Results:

The final misclassification error rate after tuning was approximately **34.58%**. This error rate indicates the proportion of spam emails incorrectly classified as non-spam and vice versa. Although parameter tuning improved the model's performance, the error rate suggests that the one-class SVM approach may have limitations for this specific spam filtering task.

Conclusion:

The one-class SVM model achieved a misclassification error rate of approximately **34.58%** after tuning. While this represents an improvement over the initial setup, the relatively high error rate indicates that the one-class SVM may not be the most effective model for spam filtering in this context. Further investigation and alternative approaches, such as using binary classification models or enhancing feature engineering, may be necessary to achieve better performance.

4 Locally weighted linear regression and bias-variance tradeoff.

4.1 Part (a): Proof and Full Mathematical Derivation

We want to show that the solution to the locally weighted linear regression problem is given by:

$$\hat{\beta} = (X^T W X)^{-1} X^T W Y$$

4.1.1 Define the Problem

Given data points (x_i, y_i) for $i = 1, \ldots, n$, where $x_i \in R^p$, we use a Gaussian kernel function:

$$K_h(z) = \frac{1}{(\sqrt{2\pi}h)^p} e^{-\frac{\|z\|^2}{2h^2}}, \quad z \in R^p$$

The objective is to solve for β_0, β_1 by minimizing the weighted sum of squared errors:

$$\hat{\beta} := (\hat{\beta}_0, \hat{\beta}_1) = \arg\min_{\beta_0,\beta_1} \sum_{i=1}^{n} (y_i - \beta_0 - (x_i - x)^T \beta_1)^2 K_h(x - x_i)$$

4.1.2 Rewrite in Matrix Form

To rewrite the above problem in matrix form, we define:

- X as the design matrix of size $n \times (p+1)$, with the first column being ones and the remaining columns being the data points x_i adjusted by x:

$$X = \begin{bmatrix} 1 & x_{11} - x_1 & x_{12} - x_2 & \cdots & x_{1p} - x_p \\ 1 & x_{21} - x_1 & x_{22} - x_2 & \cdots & x_{2p} - x_p \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} - x_1 & x_{n2} - x_2 & \cdots & x_{np} - x_p \end{bmatrix}$$

- Y as the response vector of size $n \times 1$:

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

- W as the diagonal weight matrix of size $n \times n$, where the diagonal elements are the weights computed using the Gaussian kernel function $K_h(x - x_i)$:

$$W = \begin{bmatrix} K_h(x - x_1) & 0 & \cdots & 0 \\ 0 & K_h(x - x_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & K_h(x - x_n) \end{bmatrix}$$

4.1.3 Formulate the Weighted Least Squares Problem

The objective function in matrix form is:

$$\hat{\beta} = \arg\min_{\beta} (Y - X\beta)^T W (Y - X\beta)$$

4.1.4 Derive the Solution

To find the minimum, we take the derivative of the objective function with respect to β and set it to zero:

$$\frac{\partial}{\partial \beta} (Y - X\beta)^T W (Y - X\beta) = -2X^T W (Y - X\beta) = 0$$

Solving for β, we get:

$$X^T W Y = X^T W X \beta$$

Therefore,

$$\hat{\beta} = (X^T W X)^{-1} X^T W Y$$

4.1.5 Invertibility of $X^T W X$

We assume that the matrix $X^T W X$ is invertible. This is a reasonable assumption if the design matrix X is full rank and the weight matrix W is positive definite. In practice, W being a diagonal matrix with positive entries (due to the Gaussian kernel) contributes to the invertibility of $X^T W X$.

4.1.6 Specify X, W, and Y

- X is the design matrix including a column of ones for the intercept and adjusted feature values.
- W is the diagonal weight matrix where $W_{ii} = K_h(x - x_i)$.
- Y is the response vector containing the observed values y_i.

Thus, we have shown that the solution for locally weighted linear regression is given by:

$$\hat{\beta} = (X^T W X)^{-1} X^T W Y$$

This completes the full mathematical derivation for part (a).

4.2 Part (b): Locally Weighted Linear Regression using Cross-Validation

To perform locally weighted linear regression, I used the provided ‘data.mat‘ file and implemented the Gaussian kernel function to weigh the data points based on their distance from the query point. The bandwidth parameter h was tuned using 5-fold cross-validation.

Cross-Validation Process

We defined a range of bandwidth values to test, from 0.1 to 2.0, and used 5-fold cross-validation to compute the mean squared error for each bandwidth. The cross-validation errors for the tested bandwidths are plotted in the figure below.

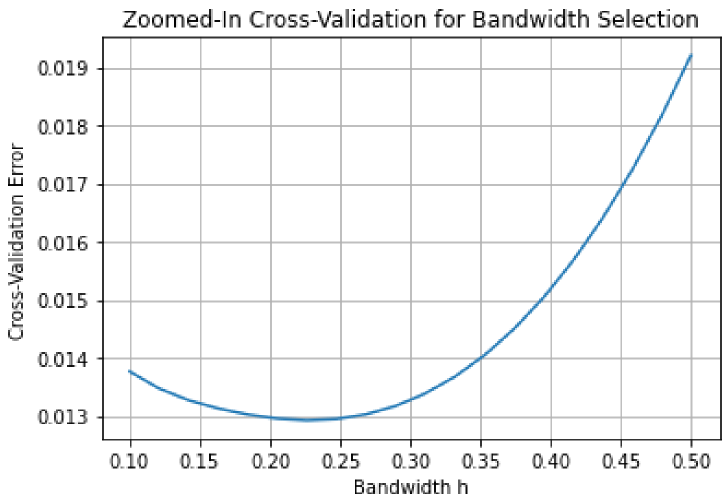


Figure 8: Zoomed-In Cross-Validation for Bandwidth Selection

Optimal Bandwidth Selection

From the cross-validation plot, we identified the optimal bandwidth as the value that minimized the cross-validation error. The best bandwidth from the zoomed-in range is found to be:

$$\text{Best bandwidth} = 0.22631578947368422$$

This optimal bandwidth will be used in part (c) to make predictions and plot the results.

4.3 Part (c): Prediction and Plot

Using the optimal bandwidth $h = 0.226$, we performed locally weighted linear regression to make a prediction for $x = -1.5$. The predicted y value is:

$$\text{Predicted } y \text{ value for } x = -1.5 : 1.8088595105728853$$

The following plot shows the noisy curve, the training data, the prediction curve, and a marker indicating the prediction at $x = -1.5$.

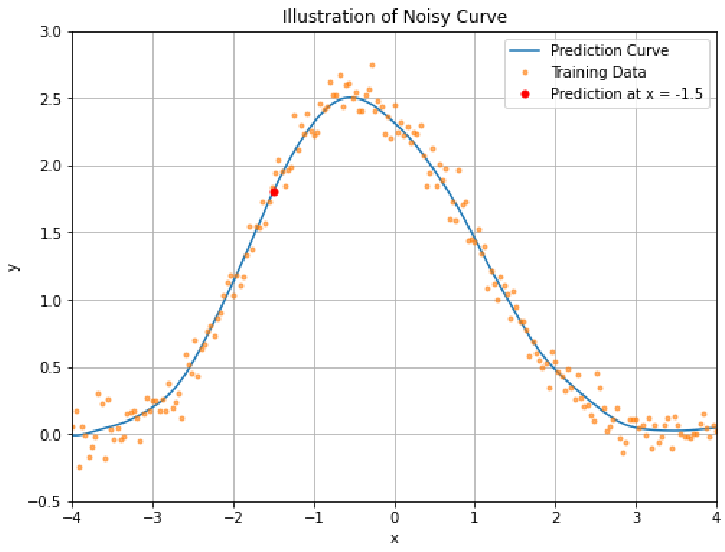


Figure 9: Illustration of Noisy Curve: Training Data, Prediction Curve, and Prediction Point