ESE 589: Project-3

AdaBoost Algorithm

**Vishwas Chaudhary (ID: 113158058)   
Anand Ravishankar (ID: 115075810)**

**Table of Contents**

1. Introduction to AdaBoost

2. Recent Advances on AdaBoost

3. Implementation details

4. Results

5. Verification

6. Appendix

**Introduction to AdaBoost**

AdaBoost, short for Adaptive Boosting, is a powerful meta-algorithm in machine learning that combines multiple weak learners into a single strong learner. This approach is based on the idea that a collection of weak learners, each individually performing slightly better than random guessing, can be combined to achieve significantly better performance.

How it Works:

1. Initialization: Assign equal weights to all training samples.
2. Iteration:
   * Train a weak learner on the weighted training data.
   * Calculate the error rate of the weak learner.
   * Update the weights of the training samples: increase the weights of misclassified samples and decrease the weights of correctly classified samples.
   * Normalize the weights so they sum to 1.
3. Repeat steps 2 and 3 for a specified number of iterations.
4. Prediction: Combine the predictions of all weak learners using a weighted voting scheme, where the weights are determined by the error rate of each weak learner.

Key Features:

* Adaptive learning: Adaptively adjusts the weights of the training samples based on their classification difficulty. This ensures that the weak learners focus on the harder-to-classify samples in subsequent iterations.
* Boosting performance: Gradually improves the overall accuracy by combining multiple weak learners, each contributing its expertise.
* Versatile: Can be used with various weak learners like decision trees, stumps, or support vector machines.

**Recent Advances on AdaBoost**

While Adaboost remains a fundamental algorithm in boosting, research continues to explore its potential through various advancements and adaptations. Here are some of the recent developments:

1. Cost-sensitive Adaboost:

Traditional Adaboost assigns equal weight to all misclassified samples, neglecting the potential cost associated with different types of errors. Cost-sensitive variants like AdaCost and GentleBoost modify the weight update process to consider the cost of misclassification, prioritizing learning from more impactful errors. This is especially useful in scenarios where misclassifying certain samples is significantly more detrimental than others.

2. Online Adaboost:

Classic Adaboost requires access to the entire training data at once, limiting its applicability to real-time or streaming data situations. Online Adaboost variants like OzaBoost and HedgeBoost overcome this by processing data sequentially and updating the model weights with each new sample. This enables Adaboost to adapt to evolving data patterns and learn continuously.

3. Ensemble methods with Adaboost:

Combining Adaboost with other ensemble algorithms like bagging and stacking can further enhance its performance. Bagging-Adaboost utilizes different weak learners on different subsets of the training data, reducing variance and improving robustness. Stacking-Adaboost combines the predictions of multiple Adaboost models trained on different data features, leading to more comprehensive and accurate predictions.

4. Regularized Adaboost:

To address the potential for overfitting, regularized Adaboost variants like LogitBoost and SmoothBoost incorporate penalties into the learning process. These penalties discourage the model from assigning excessive weights to specific weak learners, promoting balance and preventing overfitting.

**Implementation Details**

The project was implemented in Python and the code repository can be found here: [VishChaudhary/ESE-589-Project-3 (github.com)](https://github.com/VishChaudhary/ESE-589-Project-3). In this section we will go through our entire code base and explain the functionality.

1. Main.py

1. def main():

2. file = 'data/iris\_categorical.csv'

3.

4. num\_classifiers = 4

5. header, original\_data = read\_csv\_without\_libraries(file)

6. num\_classes = np.max([val[-1] for val in original\_data]) + 1

7. train\_data, x\_test, y\_test = train\_test\_split(original\_data)

8.

9. ada\_x\_train = [val[:-1] for val in train\_data]

10. ada\_y\_train = [val[-1] for val in train\_data]

11.

12. # Add initial weight to data

13. weighted\_train\_data = initial\_weights(train\_data)

Lines 2-5 define the data and number of constitunent classifiers in AdaBoost and reads the CSV data using the read\_csv\_without\_libraries function. This function returns two lists: one for headers and one for data. Line 6-7 calculates the maximum class label from the data and splits the data into training and testing sets using the train\_test\_split function. This function takes the data and an optional split ratio as input. The remaining code splits the data and the label and assigns an initial weight to the data (which is a uniform distribution).

1. for i in range(0, num\_classifiers):

2. resampled\_data, weights = resample(weighted\_train\_data)

3. x\_train, y\_train = generate\_training\_sets(resampled\_data)

4. if i == 0:

5. predictions, mlp\_model = MLP(x\_train,y\_train)

6. classifier\_models[i] = mlp\_model

7. if i == 1:

8. predictions, svm\_model = svm\_classifier(x\_train, y\_train)

9. classifier\_models[i] = svm\_model

10.

11. if i == 2:

12. predictions, decision\_tree\_model = decision\_tree(x\_train,y\_train)

13. classifier\_models[i] = decision\_tree\_model

14.

15. if i == 3:

16. predictions, our\_decision\_tree\_model = our\_decision\_tree(x\_train,y\_train)

17. classifier\_models[i] = our\_decision\_tree\_model

18.

19. was\_misclassified, error = was\_missclass(predictions, y\_train, weights=weights, x\_train=x\_train, weighted\_original\_data=weighted\_train\_data)

21.

22. if error == 0:

23. error = 1e-6

24.

25. classifier\_errors[i] = error

26. weights, weighted\_train\_data = update\_weights(was\_misclassified, error, weights, weighted\_train\_data, x\_train)

Then we loop through each classifier:

Lines 2-3: Resample the training data with replacement according to their weights. This ensures that data points with higher weights are more likely to be selected for training. Extract features and labels from the resampled data. Then we train a weak learner depending on the current iteration: Lines 4-17: If it's the first iteration (i == 0), train an MLP model using the MLP function. If it's the second iteration (i == 1), train an SVM model using the svm\_classifier function. If it's the third iteration (i == 2), train a decision tree model using the decision\_tree function. If it's the fourth iteration (i == 3), train our custom decision tree model using the our\_decision\_tree function.

Lines 19-23: Calculate the misclassified data points and error rate for the current classifier using the was\_missclass function. Line 26: Update the weights of the training data points based on their misclassification errors using the update\_weights function. This function penalizes misclassified data points and rewards correctly classified ones.

1. classifier\_weights = [0]\*num\_classifiers

2. length\_test = len(x\_test)

3. predict\_array = []

4. #num\_classes = np.max(y\_test) + 1

5. for j in range(0, length\_test):

6. classifier\_prediction = [0] \* num\_classes

7. for i in range(0, num\_classifiers):

8. classifier\_weights[i] = np.log((1-classifier\_errors[i])/(classifier\_errors[i]))

9. if i == 0:

10. prediction = np.argmax(classifier\_models[i].predict(np.array(x\_test[j]).reshape(1,-1),verbose=0))

11.

12. elif i == 3:

13. prediction = classifier\_models[i].predict(np.array(x\_test[j]).reshape(1, -1))[0][0]

14.

15. else:

16. prediction = classifier\_models[i].predict(np.array(x\_test[j]).reshape(1,-1))[0]

17.

18. classifier\_prediction[prediction] += classifier\_weights[i]

19.

20. print(f'The predicted class is: {np.argmax(classifier\_prediction)} and the actual class is: {y\_test[j]}')

21. predict\_array.append(np.argmax(classifier\_prediction))

22.

23. predict\_array = np.array(predict\_array)

24. y\_test = np.array(y\_test)

25.

Then we loop through the test data. This loop does the following:

1. Initialize an empty list to store predictions from each classifier.

2. Loop through each weak learner and make a prediction on the current test data point:

3. Use the trained model from the current weak learner to make a prediction.

4. Adjust the prediction based on the classifier weight.

5. Add the adjusted prediction to the list of predictions.

6. Predict the final class based on the weighted majority vote of all classifiers.

7. Print the predicted class and the actual class for comparison.

8. Add the predicted class to a list for later evaluation.

Finally, we convert the predicted class list and actual class list to NumPy arrays for easier comparison.

1. for i in range(0, length\_test):

2. if predict\_array[i] != y\_test[i]:

3. incorrect\_sum += 1

4.

5. print(f'{incorrect\_sum} out of {length\_test} values misclassified')

Finally we calculate the number of misclassified data points in the test set.

The implementation discussed thus far describes the adaboost described in the referernce textbook. However, plently of online resources as well as the seminal paper on AdaBoost desribe a key step while using Decision trees as weak learners called **Stumping**.

Stumping refers to having decision trees of height 1 as weak learners. In boosting, the trees are fitted sequentially, with each one trained on (in some sense) the residuals from the previous classifier. Once a boosted ensemble starts overfitting, it will keep overfitting; the errors won't just cancel out. For this reason, it's worth having individual trees be short when boosting and tall when bagging.

We have also implemented this method of stumping in the adaboost.py file:

1. Adaboost.py:

1. class Stump(DecisionTreeClassifier\_2):

2.

3. def \_\_init\_\_(self, min\_samples\_split):

4. super().\_\_init\_\_(max\_depth=1, min\_samples\_split=min\_samples\_split)

5. self.weight = None

6.

* This class is defined as a subclass of DecisionTreeClassifier\_2 (which is our custom decision tree from Project-2).
* It has an \_\_init\_\_ method that takes the parameter min\_samples\_split and initializes the superclass (DecisionTreeClassifier\_2) with a maximum depth of 1 and the specified min\_samples\_split.
* It has an attribute weight initialized to None, which is same as the initial weight assigned to individual learners.

1. class AdaBoostClassifier:

2.

3. def \_\_init\_\_(self, n\_estimators=100, min\_samples\_split=2):

4. self.n\_estimators = n\_estimators

5. self.min\_samples\_split = min\_samples\_split

6. self.stumps = []

7. self.sample\_weights = []

8.

9. def \_build\_forest\_stump(self, X, y):

10. stump = Stump(self.min\_samples\_split)

11. stump.fit(X, y)

12. return stump

13.

This is the overall class definition for the classifier with stumps. The \_\_init\_\_ method initializes the object with default values for n\_estimators (number of weak classifiers) and min\_samples\_split. It also initializes empty lists for stumps and sample\_weights. The \_build\_forest\_stump creates and trains a single stump using the provided data.

1. def \_build\_forest(self, X, y):

2. self.sample\_weights = np.ones(len(X)) / len(X)

3.

4. for \_ in range(self.n\_estimators):

5. random\_indices = np.random.choice(len(X), len(X), p=self.sample\_weights)

6. stump = self.\_build\_forest\_stump(X[random\_indices], y[random\_indices])

7. y\_pred = stump.predict(X)

8.

9. misclassified = np.array([int(i) for i in (y\_pred != y)])

10. error = np.sum(self.sample\_weights \* misclassified) / np.sum(self.sample\_weights)

11. stump.weight = 0.5 \* math.log((1 - error) / error)

12.

13. self.sample\_weights[misclassified == 0] \*= np.exp(-stump.weight)

14. self.sample\_weights[misclassified == 1] \*= np.exp(stump.weight)

15. self.sample\_weights /= np.sum(self.sample\_weights)

16.

17. self.stumps.append(stump)

18.

* This method builds the ensemble of weak classifiers (decision stumps) iteratively.
* It initializes sample weights and then iterates for n\_estimators times.
* In each iteration, it selects a random subset of data based on the sample weights, trains a decision stump, calculates the weighted error, assigns a weight to the stump, and updates the sample weights based on the performance of the stump.
* The trained stump is then added to the list of stumps.

1. def fit(self, X, y):

2. self.\_build\_forest(np.array(X), np.array(y))

3.

4. def predict(self, X):

5. return np.array([self.\_make\_prediction(x) for x in np.array(X)])

6.

7. def \_make\_prediction(self, x):

8. stump\_preds = {}

9. for stump in self.stumps:

10. pred = stump.predict([x])[0]

11. if pred in stump\_preds:

12. stump\_preds[pred] += stump.weight

13. else:

14. stump\_preds[pred] = stump.weight

15. return max(stump\_preds, key=stump\_preds.get)

16.

This method a wrapper that calls the \_build\_forest method to train the ensemble using input data X and labels y. This method predicts the class labels for the input data X by calling the \_make\_prediction method for each data point in X. This method combines the predictions from all stumps by assigning weights to each stump's prediction. It returns the class label with the highest total weight.

1. Decision\_tree.py

1. class Node:

2.

3. # default arguments set to none since the node can be decision or leaf

4. def \_\_init\_\_(self, feature\_index=None, threshold=None, left=None, right=None, value=None):

5.

6. # decision node

7. self.feature\_index = feature\_index

8. self.threshold = threshold

9. self.left = left

10. self.right = right

11.

12. # leaf node

13. self.value = value

14.

15. class DecisionTreeClassifier\_2:

16.

17. def \_\_init\_\_(self, min\_samples\_split=2, max\_depth=float('inf'), criterion='gini'):

18. # will be set as the tree when fit function is run

19. self.root = None

20.

21. self.min\_samples\_split = min\_samples\_split

22. self.max\_depth = max\_depth

23. self.criterion = criterion

24. self.tree\_depth = 0

25.

26. def \_build\_tree(self, X, y, depth=1):

27. num\_samples = np.shape(X)[0]

28.

29. # used later for print\_tree function

30. if depth > self.tree\_depth:

31. self.tree\_depth = depth

32.

33. # if the current node is a decision node

34. if num\_samples >= self.min\_samples\_split and self.max\_depth >= depth:

35. # get the best split with the maximum information gain

36. best\_split = self.\_get\_best\_split(X, y)

37.

38. if best\_split['information\_gain'] > 0:

39. # recursively generate the the subtrees

40. left\_subtree = self.\_build\_tree(depth=depth+1,\*\*best\_split['left'])

41. right\_subtree = self.\_build\_tree(depth=depth+1,\*\*best\_split['right'])

42.

43. return Node(best\_split['feature\_index'], best\_split['threshold'], left\_subtree, right\_subtree)

44.

45. Y = list(y)

46. value = max(Y, key=Y.count)

47. return Node(value=value)

48.

49. def \_get\_best\_split(self, X, y):

50. num\_features = np.shape(X)[1]

51. max\_gain = -float('inf')

52. best\_split = {}

53.

54. # iterates through each possible feature and threshold combination

55. for feature\_index in range(num\_features):

56. for threshold in np.unique(X[:, feature\_index]):

57. # condition for dividing the samples

58. left\_c = X[:, feature\_index] <= threshold

59. right\_c = X[:, feature\_index] > threshold

60.

61. # divides all the samples into two nodes based on the feature\_index and threshold

62. left\_X, right\_X = X[left\_c], X[right\_c]

63. left\_y, right\_y = y[left\_c], y[right\_c]

64.

65. # calculate the information gained by a unique split

66. information\_gain = self.\_get\_information\_gain(y, left\_y, right\_y)

67. if information\_gain > max\_gain:

68. best\_split['feature\_index'] = feature\_index

69. best\_split['threshold'] = threshold

70. best\_split['left'] = {'X': left\_X, 'y': left\_y}

71. best\_split['right'] = {'X': right\_X, 'y': right\_y}

72. best\_split['information\_gain'] = information\_gain

73. max\_gain = information\_gain

74.

75. return best\_split

76.

77. def \_get\_information\_gain(self, y, left, right):

78. l\_weight = len(left) / (len(left) + len(right))

79. r\_weight = 1 - l\_weight

80.

81. if self.criterion == 'entropy':

82. return self.\_calculate\_entropy(y) - (l\_weight \* self.\_calculate\_entropy(left) + r\_weight \* self.\_calculate\_entropy(right))

83. elif self.criterion == 'gini':

84. return self.\_calculate\_gini(y) - (l\_weight \* self.\_calculate\_gini(left) + r\_weight \* self.\_calculate\_gini(right))

85. else:

86. raise ValueError('criterion can only be "entropy" or "gini"')

87.

88. def \_calculate\_entropy(self, y):

89. classes = np.unique(y)

90. entropy = 0

91. for cls in classes:

92. p\_cls = len(y[y == cls]) / len(y)

93. entropy += -p\_cls \* np.log2(p\_cls)

94. return entropy

95.

96. def \_calculate\_gini(self, y):

97. classes = np.unique(y)

98. gini = 0

99. for cls in classes:

100. p\_cls = len(y[y == cls]) / len(y)

101. gini += p\_cls \*\* 2

102. return 1 - gini

103.

104. def fit(self, X, y):

105. self.column\_names = list(X.columns) if not isinstance(X, np.ndarray) else None

106. self.root = self.\_build\_tree(np.array(X), np.array(y))

107.

108. def predict(self, X):

109. return [self.\_make\_prediction(x, self.root) for x in np.array(X)]

110.

111. def \_make\_prediction(self, x, tree):

112. if tree.value != None:

113. return tree.value

114. if x[tree.feature\_index] <= tree.threshold:

115. return self.\_make\_prediction(x, tree.left)

116. else:

117. return self.\_make\_prediction(x, tree.right)

118.

119. def \_print\_tree(self, tree, i=0):

120. if self.root == None:

121. print(f'Decision Tree has not been trained yet')

122. return

123. if tree == None:

124. return ''

125. if tree.value is not None: return tree.value

126. feature\_name = self.column\_names[tree.feature\_index]

127. return f'{feature\_name} <= {tree.threshold}\n' + \

128. f'{"|" \* i}left: {self.\_print\_tree(tree.left, i+1)}\n' + \

129. f'{"|" \* i}right: {self.\_print\_tree(tree.right, i+1)}'

130.

131. def print\_tree(self):

132. print(self.\_print\_tree(self.root))

133.

This method explain the custom decision tree implementation which was discussed in Project-2.

1. Neural Network, SVM, Decision Tree

1. def MLP(x\_train, y\_train):

2. x\_train = np.array(x\_train)

3. input\_shape = x\_train.shape

4. output\_shape = np.max(y\_train) + 1

5. y\_train = np.array(y\_train)

6. model = tf.keras.Sequential([

7. tf.keras.layers.Dense(128, activation='relu', input\_shape=(input\_shape[1],)),

8. tf.keras.layers.Dropout(0.1),

9. tf.keras.layers.Dense(64, activation='relu'),

10. tf.keras.layers.Dropout(0.1),

11. tf.keras.layers.Dense(32, activation='relu'),

12. tf.keras.layers.Dropout(0.1),

13. tf.keras.layers.Dense(output\_shape, activation='softmax')

14. ])

15. model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

16. model.fit(x\_train, y\_train, epochs=15)

17. accuracy = model.evaluate(x\_train, y\_train)[1]

18. print(f'Accuracy: {accuracy}')

19. predictions = [np.argmax(prob) for prob in model.predict(x\_train,verbose=0)]

20. return predictions, model

21.

22. def svm\_classifier(x\_train, y\_train):

23. x\_train = np.array(x\_train)

24. y\_train = np.array(y\_train)

25. model = svm.SVC()

26. model.fit(x\_train,y\_train)

27. accuracy = model.score(x\_train,y\_train)

28. predictions = model.predict(x\_train)

29. print(f'Svm accuracy: {accuracy}')

30. return predictions, model

31.

32. def decision\_tree(x\_train,y\_train,criterion = 'gini',max\_depth=None):

33. model = DecisionTreeClassifier()

34. model.fit(x\_train,y\_train)

35. accuracy = model.score(x\_train,y\_train)

36. print(f'Sklearn Decision Tree Accuracy: {accuracy}')

37. predictions = model.predict(x\_train)

38. return predictions,model

39.

These methods define the weak learners using open-source frameworks like Tensorflow and Sklearn.

**RESULTS AND DISCUSSION**

Confusion matrix based metrics:  
  
A confusion matrix is a nxn matrix (n = number of classes) that summarizes the performance of a classification model on a set of test data. It shows how well the model can distinguish between different classes, such as positive and negative, or dog and not dog. A confusion matrix has four cells: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Each cell represents the number of instances that were correctly or incorrectly classified by the model. The main results derived from a confusion matrix are:

1. Accuracy: This is the most intuitive performance measure and it gives you the ratio of correctly predicted instances to the total instances. High accuracy is generally desirable, but it doesn't tell you the specific types of errors (false positives vs. false negatives). In engineering terms, it's a good high-level metric, but for detailed performance evaluation, especially in imbalanced datasets, it's not sufficient.
2. Precision: Precision is the ratio of true positives to the sum of true and false positives. High precision relates to a low rate of false positives. In engineering terms, this is crucial when the cost of false positives is high — for example, in spam detection (where you don't want to mark legitimate emails as spam) or in manufacturing where false positives might mean discarding good products.
3. Recall: Also known as sensitivity or true positive rate, recall is the ratio of true positives to the sum of true positives and false negatives. High recall is critical in situations where missing a positive is costly, such as fraud detection or medical diagnostics where you don't want to miss any true cases.
4. F1-Score: The F1-score is the harmonic mean of precision and recall. It's a single metric that combines both precision and recall. It's especially useful when you need to balance precision and recall, which often have an inverse relationship. An engineering project often requires optimizing to a point where both false positives and false negatives are minimized, making the F1-score a valuable metric.

These results are shown in the following figure for each dataset, for each information spllit criterion. The real-valued data was binned uniformly into 3 groups and the train-test split was set to 70-30%

First we shall see the results of using a neural network, a SVM and decision trees (open sourced and custom) as the weak learners.

A graph of a number of blue and white bars

Description automatically generated with medium confidenceA screenshot of a graph

Description automatically generated

**Dataset-specific result discussion:**  
Balance-Scale: This dataset involves classifying the balance scale tip as either balanced or imbalanced based on certain weights. The accuracy is 0.79, indicating that the model correctly predicted the balance status in 79% of the cases. However, the precision and recall are relatively low, which suggests that the model has room for improvement, especially in correctly predicting the positive class and not misclassifying the negative class as positive.

Breast-Cancer: This dataset involves the classification of breast cancer instances as malignant or benign. The results indicate a very high performance across all metrics, with accuracy, precision, recall, and F1-score all around 0.98. This suggests that the model is very effective at correctly identifying cancer cases and has a low rate of false positives and false negatives.

Wine: This dataset is about classifying the cultivar origin of wine samples based on chemical analysis. The model shows strong performance with an accuracy and F1-score above 0.88, suggesting a good balance between precision and recall.

Yeast: This dataset is used for predicting protein localization sites in yeast. The results here indicate moderate performance with accuracy at 0.85, but with somewhat lower precision and recall, indicating some potential confusion in class predictions.

Telescope: This task is to classify images as signal (gamma-ray) or background (hadron). The model performs well with accuracy and precision both at 0.92, and an F1-score of 0.90.

Iris: One of the most famous datasets in machine learning, used for classifying iris flowers into three species. The results show excellent performance across all metrics, with the lowest being precision at 0.94.

Haberman: This refers to the survival of patients who had undergone surgery for breast cancer. The model's performance is less effective here, with accuracy at 0.73 and other metrics also lower, which could be due to the challenging nature of the prediction task.

Glass: This dataset is used for classifying types of glass based on their oxide content. The model shows a moderate performance with an accuracy of 0.70 and similarly moderate precision, recall, and F1-score.

DNA: A dataset involving the classification of DNA sequences. The model's performance is good with an accuracy of 0.86 but has lower precision and recall, which could be due to the complexity of the patterns in the sequences.

Abalone: This dataset involves predicting the age of abalone from physical measurements. The model's performance is relatively lower, with an accuracy of 0.62, indicating this may be a more challenging dataset for the model.

Balloons: This dataset involves classifying whether balloons are inflated or not based on certain attributes. The accuracy and F1-score are both at 0.80, indicating a reasonably good model performance.

Car\_evaluation: This dataset is used for evaluating the acceptability of cars based on various attributes. The model performs very well with an accuracy of 0.90, and high precision and recall, indicating effective classification.

Key points:  
  
**Breast-Cancer and Iris**: The high scores across all metrics suggest the model is well-tuned to these datasets. It indicates a good balance between identifying positive cases and not being overwhelmed by false positives. This level of performance might be due to well-defined features that lead to clear decision boundaries.

**Abalone**: The lower scores indicate a challenging dataset for the model. This might be due to more complex relationships between features and the target variable, requiring more sophisticated feature engineering or a different model architecture.

**Glass and Haberman**: Moderate performance suggests some degree of complexity or class overlap in the datasets. In an engineering context, this might prompt a review of the feature selection or data preprocessing steps to improve the model's ability to distinguish between classes.

**Yeast, Telescope, and DNA**: While not low, these metrics suggest that the model has room for improvement. Engineering efforts might focus on gathering more data, feature engineering, or hyperparameter tuning.

Now we shall see the results of using multiple (100) stumped trees:

A graph of a number of red and orange bars

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

**Balance-Scale (0.74 Accuracy, 0.49 Precision, 0.53 Recall, 0.51 F1-Score):**

The model performs moderately overall.

Low precision suggests a relatively high number of false positives; the model predicts the positive class too frequently.

The balance between precision and recall (F1-Score) is also moderate, indicating room for improvement.

**Breast-Cancer (0.93 Accuracy, 0.92 Precision, 0.94 Recall, 0.93 F1-Score):**

High scores across all metrics indicate the model is very effective at classifying this dataset.

High precision and recall suggest the model accurately identifies positive cases and maintains a low false positive rate.

**Wine (0.83 Accuracy, 0.85 Precision, 0.87 Recall, 0.86 F1-Score):**

The model performs well, with high precision and recall, indicating effective classification of wine varieties.

**Yeast (0.80 Accuracy, 0.75 Precision, 0.78 Recall, 0.76 F1-Score):**

The performance is good but indicates potential difficulty in differentiating between the classes, which is common in biological datasets.

**Telescope (0.87 Accuracy, 0.84 Precision, 0.86 Recall, 0.85 F1-Score):**

The model shows strong performance, likely effectively distinguishing between celestial objects.

**Iris (0.99 Accuracy, 0.99 Precision, 0.97 Recall, 0.98 F1-Score):**

Near-perfect scores suggest the model has almost perfectly classified the well-separated iris species.

**Haberman (0.68 Accuracy, 0.65 Precision, 0.63 Recall, 0.64 F1-Score):**

The model struggles with this dataset, potentially due to overlapping features between classes or insufficient data representation.

**Glass (0.65 Accuracy, 0.62 Precision, 0.64 Recall, 0.63 F1-Score):**

Moderate performance might be due to the complexity of accurately classifying glass types.

**DNA (0.81 Accuracy, 0.80 Precision, 0.79 Recall, 0.79 F1-Score):**

The model performs well, which is notable for a complex dataset like DNA sequences.

**Abalone (0.57 Accuracy, 0.55 Precision, 0.60 Recall, 0.57 F1-Score):**

Low scores suggest the model has difficulty predicting age categories, which can be a complex   
regression-like classification problem.

**Balloons (0.75 Accuracy, 0.73 Precision, 0.72 Recall, 0.72 F1-Score):**

The model has moderate performance; the task seems neither particularly easy nor hard for the model.

**Car\_evaluation (0.85 Accuracy, 0.83 Precision, 0.84 Recall, 0.83 F1-Score):**

Good performance, the model likely captures the features well that determine car evaluation classes.

From an engineering perspective, the use of stumps helps to prevent overfitting, especially in datasets with high variance or less-clearly defined decision boundaries. However, the trade-off might be a loss in model complexity and potentially reduced performance in capturing more nuanced patterns within the data. Overall, the decision stump approach seems to work exceptionally well for datasets like Iris and Breast-Cancer, which may have well-defined and separable classes, but less so for datasets like Haberman and Abalone, which may benefit from a more complex model capable of capturing subtleties in the data.

**In terms of timing and memory utilization**, we note that use of stumps decrease the time utilization by a magnitude of 10x whioe the memory utilization remains the same. This can be attributed to the memory utilization used in importing the libraries for the open source frameworks.

**Effect of varying number of base estimators:**

A graph with green lines

Description automatically generated

Here we see the result of varying the number of stumps for the Iris dataset. Here are some analysis we did:  
  
**Initial Increase**: As the number of estimators increases from a small number up to around 20-40, there's a sharp increase in all performance metrics. This indicates that adding more weak learners (stumps) initially improves the model's ability to capture the underlying patterns in the data.

**Peak Performance**: Between approximately 20-40 estimators, the model achieves peak performance. This is where the balance between bias and variance is optimal, leading to the best generalization on unseen data.

**Performance Decline**: After reaching the peak, there's a noticeable decline in performance as more estimators are added. This suggests that the model begins to overfit the training data, paying too much attention to the noise or outliers, which negatively affects its generalization capability.

**Convergence and Plateau**: Beyond approximately 100 estimators, the performance metrics seem to converge and plateau. The decline stabilizes, and further increases in the number of estimators do not lead to significant changes in performance. This indicates that the model has reached its capacity to learn from the data with the given features and hyperparameters.

**Performance Metrics Behavior**: It's also worth noting how the different metrics behave relative to each other:

**Precision**: Appears to be the most stable metric, indicating that the proportion of true positives over all positive predictions remains relatively constant despite the number of estimators.

**Recall and F1-Score**: These metrics experience a more pronounced decline after the peak, which might suggest that as the model overfits, it becomes less capable of detecting all positive cases (true positives), impacting the balance between precision and recall (harmony in F1-Score).

Increasing the number of stumps in an AdaBoost classifier typically improves model performance initially, as the diversity of weak learners allows for a better capture of underlying patterns. However, there's an optimal range of stump counts where the model performs best, often observed in the range of 20-40 estimators for this particular dataset. Beyond this range, the benefits of adding more stumps diminish, and the performance starts to decline due to overfitting, where the model becomes too tailored to the training data, impairing its ability to generalize. Eventually, the model's performance metrics stabilize, indicating that adding more stumps doesn't contribute to learning further from the data, and the model's predictive power plateaus regardless of the increased complexity.

VERIFICATION:  
  
We can see that both of our Adaboost methods provide similar results, which can be seen in the confusion matrix created. For credit\_card dataset:  
  
Without stumps:  
  
Confusion Matrix:

[[2132 190]

[ 552 126]]  
  
With Stumps:  
  
Confusion Matrix:

[[2277 45]

[ 639 39]]  
  
  
This leads us to believe that the implementation is correct. Testing AdaBoost by hand is not practically feasible since there will be multiple classifiers to train by hand.   
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
APPENDIX  
  
**main.py:**  
  
import numpy as np

import tensorflow as tf

from sklearn import svm

from sklearn.tree import DecisionTreeClassifier

from adaboost import AdaBoostClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import pandas as pd

import time, resource

from decision\_tree import DecisionTreeClassifier\_2

from sklearn.metrics import confusion\_matrix, accuracy\_score, precision\_score, recall\_score, f1\_score

def accuracy\_multi\_class(confusion\_matrix):

# Calculate accuracy for multi-class classification

correct = sum(confusion\_matrix[i][i] for i in range(len(confusion\_matrix)))

total = sum(sum(row) for row in confusion\_matrix)

return correct / total if total != 0 else 0

def precision\_multi\_class(confusion\_matrix):

# Calculate precision for each class and return a list of precisions

precisions = []

for i in range(len(confusion\_matrix)):

true\_positive = confusion\_matrix[i][i]

false\_positive = sum(confusion\_matrix[row][i] for row in range(len(confusion\_matrix))) - true\_positive

precision = true\_positive / (true\_positive + false\_positive) if (true\_positive + false\_positive) != 0 else 0

precisions.append(precision)

return precisions

def recall\_multi\_class(confusion\_matrix):

# Calculate recall for each class and return a list of recalls

recalls = []

for i in range(len(confusion\_matrix)):

true\_positive = confusion\_matrix[i][i]

false\_negative = sum(confusion\_matrix[i]) - true\_positive

recall = true\_positive / (true\_positive + false\_negative) if (true\_positive + false\_negative) != 0 else 0

recalls.append(recall)

return recalls

def f1\_score\_multi\_class(precisions, recalls):

# Calculate F1-score for each class and return a list of F1-scores

f1\_scores = []

for i in range(len(precisions)):

precision = precisions[i]

recall = recalls[i]

f1\_score = 2 \* (precision \* recall) / (precision + recall) if (precision + recall) != 0 else 0

f1\_scores.append(f1\_score)

return f1\_scores

# Read in data

def read\_csv\_without\_libraries(filename, delimiter=","):

with open(filename, "r") as f:

headers = f.readline().strip().split(delimiter)

data = []

for line in f:

row = [(int(float(entry.strip()))) for entry in line.split(delimiter)]

data.append(row)

return headers, data

# Initial weight of 1/d to all

# mod\_data -> List[Tuple(List[],List[])]

# Tuple: [i:0-len(data)][0][0]

# Data: [i:0-len(data][1][k:len(row) {this is for accessing and changing a row in the data}]

def initial\_weights(data):

weight = 1 / len(data)

mod\_data = [([weight], row) for row in data]

return mod\_data

def resample(data, num\_sample=None):

num\_sample = num\_sample or len(data)

# Get weight for each training example and append into 1 list

offset = 1e-9

weights = []

for row in data:

if row[0][0] == 0:

weights.append(offset)

else:

weights.append(row[0][0])

# Sample indices with replacement with the weight of each training set being its probability of being picked

indices = np.random.choice(len(data), size=num\_sample, replace=True, p=weights / np.sum(weights))

data\_resampled = []

for i in indices:

data\_resampled.append(data[i])

return data\_resampled, weights

def generate\_training\_sets(data):

x\_train = []

y\_train = []

for element in data:

x\_train.append(element[1][:-1])

y\_train.append(element[1][-1])

return x\_train, y\_train

def train\_test\_split(data, split = 0.1):

np.random.shuffle(data)

length = len(data)

split\_index = int(length\*split)

test\_data = data[:split\_index]

train\_data = data[split\_index:]

x\_test = []

y\_test = []

for row in test\_data:

x\_test.append(row[:-1])

y\_test.append(row[-1])

return train\_data, x\_test, y\_test

def MLP(x\_train, y\_train):

x\_train = np.array(x\_train)

input\_shape = x\_train.shape

output\_shape = np.max(y\_train) + 1

y\_train = np.array(y\_train)

model = tf.keras.Sequential([

tf.keras.layers.Dense(128, activation='relu', input\_shape=(input\_shape[1],)),

tf.keras.layers.Dropout(0.1),

tf.keras.layers.Dense(64, activation='relu'),

tf.keras.layers.Dropout(0.1),

tf.keras.layers.Dense(32, activation='relu'),

tf.keras.layers.Dropout(0.1),

tf.keras.layers.Dense(output\_shape, activation='softmax')

])

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=15)

accuracy = model.evaluate(x\_train, y\_train)[1]

print(f'Neural Network Accuracy: {accuracy}')

predictions = [np.argmax(prob) for prob in model.predict(x\_train,verbose=0)]

return predictions, model

def svm\_classifier(x\_train, y\_train):

x\_train = np.array(x\_train)

y\_train = np.array(y\_train)

model = svm.SVC()

model.fit(x\_train,y\_train)

accuracy = model.score(x\_train,y\_train)

predictions = model.predict(x\_train)

print(f'Svm accuracy: {accuracy}')

return predictions, model

def decision\_tree(x\_train,y\_train,criterion = 'gini',max\_depth=None):

model = DecisionTreeClassifier()

model.fit(x\_train,y\_train)

accuracy = model.score(x\_train,y\_train)

print(f'Sklearn Decision Tree Accuracy: {accuracy}')

predictions = model.predict(x\_train)

return predictions,model

def our\_decision\_tree(x\_train, y\_train, max\_depth = 3):

model = DecisionTreeClassifier\_2()

x\_train = pd.DataFrame(x\_train)

y\_train = pd.DataFrame(y\_train)

model.fit(x\_train, y\_train)

y\_pred = model.predict(x\_train)

prediction = [val[0] for val in y\_pred]

print(f'Our Decision Tree Accuracy: {accuracy\_score(y\_pred, y\_train)}')

return prediction, model

def custom\_adaboost(x\_train,y\_train, n\_estimators=100, min\_samples\_split=2):

x\_train = pd.DataFrame(x\_train)

y\_train = pd.Series(y\_train)

model = AdaBoostClassifier(n\_estimators, min\_samples\_split)

model.fit(x\_train, y\_train)

prediction = model.predict(x\_train)

# print(f'Custom Stumped Adaboost Accuracy: {accuracy\_score(prediction, y\_train):}')

return prediction, model

def was\_missclass(predictions, y\_train, x\_train, weighted\_original\_data, weights):

is\_misclassified = [False] \* len(predictions)

error = 0.0

for i, predict\_val in enumerate(predictions):

if y\_train[i] != predict\_val:

index = return\_index(weighted\_original\_data, x\_train[i])

error += weights[index]

is\_misclassified[i] = True

return is\_misclassified, error

def return\_index(sampled\_data, x\_train):

for index, val in enumerate(sampled\_data):

if x\_train == val[1][:-1]:

return index

print('Fail')

return 0

def update\_weights(was\_misclassified, error, weights, weighted\_original\_data, x\_train):

bias = error / (1 - error)

old\_weights\_sum = np.sum(weights)

for i, val in enumerate(was\_misclassified):

if not val:

index = return\_index(weighted\_original\_data, x\_train[i])

weighted\_original\_data[i][0][0] \*= bias

weights[index] \*= bias

new\_weights\_sum = np.sum(weights)

norm\_term = old\_weights\_sum / new\_weights\_sum

new\_weights = []

for row in weighted\_original\_data:

row[0][0] \*= norm\_term

new\_weights.append(row[0][0])

return new\_weights, weighted\_original\_data

def main():

time\_start = time.perf\_counter()

file = 'final\_datasets/credit\_card\_binned.csv'

num\_classifiers = 4

header, original\_data = read\_csv\_without\_libraries(file)

num\_classes = np.max([val[-1] for val in original\_data]) + 1

train\_data, x\_test, y\_test = train\_test\_split(original\_data)

ada\_x\_train = [val[:-1] for val in train\_data]

ada\_y\_train = [val[-1] for val in train\_data]

# Add initial weight to data

weighted\_train\_data = initial\_weights(train\_data)

classifier\_errors = [-1]\*num\_classifiers

classifier\_models = [0]\*num\_classifiers

for i in range(0, num\_classifiers):

resampled\_data, weights = resample(weighted\_train\_data)

x\_train, y\_train = generate\_training\_sets(resampled\_data)

if i == 0:

predictions, mlp\_model = MLP(x\_train,y\_train)

classifier\_models[i] = mlp\_model

if i == 1:

predictions, svm\_model = svm\_classifier(x\_train, y\_train)

classifier\_models[i] = svm\_model

if i == 2:

predictions, decision\_tree\_model = decision\_tree(x\_train,y\_train)

classifier\_models[i] = decision\_tree\_model

if i == 3:

predictions, our\_decision\_tree\_model = our\_decision\_tree(x\_train,y\_train)

classifier\_models[i] = our\_decision\_tree\_model

was\_misclassified, error = was\_missclass(predictions, y\_train, weights=weights, x\_train=x\_train,

weighted\_original\_data=weighted\_train\_data)

if error == 0:

error = 1e-6

classifier\_errors[i] = error

weights, weighted\_train\_data = update\_weights(was\_misclassified, error, weights, weighted\_train\_data,

x\_train)

classifier\_weights = [0]\*num\_classifiers

length\_test = len(x\_test)

predict\_array = []

#num\_classes = np.max(y\_test) + 1

for j in range(0, length\_test):

classifier\_prediction = [0] \* num\_classes

for i in range(0, num\_classifiers):

classifier\_weights[i] = np.log((1-classifier\_errors[i])/(classifier\_errors[i]))

if i == 0:

prediction = np.argmax(classifier\_models[i].predict(np.array(x\_test[j]).reshape(1,-1),verbose=0))

elif i == 3:

prediction = classifier\_models[i].predict(np.array(x\_test[j]).reshape(1, -1))[0][0]

else:

prediction = classifier\_models[i].predict(np.array(x\_test[j]).reshape(1,-1))[0]

classifier\_prediction[prediction] += classifier\_weights[i]

# print(f'The predicted class is: {np.argmax(classifier\_prediction)} and the actual class is: {y\_test[j]}')

predict\_array.append(np.argmax(classifier\_prediction))

predict\_array = np.array(predict\_array)

y\_test = np.array(y\_test)

# print(predict\_array)

# print(y\_test)

incorrect\_sum = 0

for i in range(0, length\_test):

if predict\_array[i] != y\_test[i]:

incorrect\_sum += 1

print(f'{incorrect\_sum} out of {length\_test} values misclassified')

conf\_matrix = confusion\_matrix(y\_test, predict\_array)

# Extract true positives, true negatives, false positives, and false negatives

# Calculate metrics

accuracy = accuracy\_multi\_class(conf\_matrix)

precision = precision\_multi\_class(conf\_matrix)

recall = recall\_multi\_class(conf\_matrix)

f1 = f1\_score\_multi\_class(precision, recall)

# Print the results

print("Confusion Matrix:")

print(conf\_matrix)

print("\nMetrics:")

print(f"Accuracy: {accuracy:.2f}")

print(f"Precisions: {sum(precision)/len(precision):.2f}")

print(f"Recalls: {sum(recall)/len(recall):.2f}")

print(f"F1 Scores: {sum(f1)/len(f1):.2f}")

time\_elapsed = (time.perf\_counter() - time\_start)

memMb = resource.getrusage(resource.RUSAGE\_SELF).ru\_maxrss / 1024.0 / 1024.0

print(f"Adaboost without stumping took %5.4f secs %5.4f MByte" % (time\_elapsed, memMb))

print(f'Stumped Adaboost implementation:\n')

time\_start = time.perf\_counter()

\_, ada\_model = custom\_adaboost(ada\_x\_train,ada\_y\_train)

ada\_prediction = ada\_model.predict(pd.DataFrame(x\_test))

# print(f'Predicted Class: {ada\_prediction}')

incorrect\_sum = 0

for i in range(0, length\_test):

if ada\_prediction[i] != y\_test[i]:

incorrect\_sum += 1

print(f'{incorrect\_sum} out of {length\_test} values misclassified')

conf\_matrix = confusion\_matrix(y\_test, ada\_prediction)

# Extract true positives, true negatives, false positives, and false negatives

accuracy = accuracy\_multi\_class(conf\_matrix)

precision = precision\_multi\_class(conf\_matrix)

recall = recall\_multi\_class(conf\_matrix)

f1 = f1\_score\_multi\_class(precision, recall)

# Print the results

print("Confusion Matrix:")

print(conf\_matrix)

print("\nMetrics:")

print(f"Accuracy: {accuracy:.2f}")

print(f"Precisions: {sum(precision)/len(precision):.2f}")

print(f"Recalls: {sum(recall)/len(recall):.2f}")

print(f"F1 Scores: {sum(f1)/len(f1):.2f}")

time\_elapsed = (time.perf\_counter() - time\_start)

memMb = resource.getrusage(resource.RUSAGE\_SELF).ru\_maxrss / 1024.0 / 1024.0

print(f"Adaboost with stumping took %5.4f secs %5.4f MByte" % (time\_elapsed, memMb))

if \_\_name\_\_ == "\_\_main\_\_":

main()

**adaboost.py:**  
import math

import numpy as np

import os

import sys

sys.path.append(os.path.join(os.path.dirname(sys.path[0]), 'decision-tree'))

from decision\_tree import DecisionTreeClassifier\_2 # type: ignore

class Stump(DecisionTreeClassifier\_2):

def \_\_init\_\_(self, min\_samples\_split):

super().\_\_init\_\_(max\_depth=1, min\_samples\_split=min\_samples\_split)

self.weight = None

class AdaBoostClassifier:

def \_\_init\_\_(self, n\_estimators=100, min\_samples\_split=2):

self.n\_estimators = n\_estimators

self.min\_samples\_split = min\_samples\_split

self.stumps = []

self.sample\_weights = []

def \_build\_forest\_stump(self, X, y):

stump = Stump(self.min\_samples\_split)

stump.fit(X, y)

return stump

def \_build\_forest(self, X, y):

self.sample\_weights = np.ones(len(X)) / len(X)

for \_ in range(self.n\_estimators):

random\_indices = np.random.choice(len(X), len(X), p=self.sample\_weights)

stump = self.\_build\_forest\_stump(X[random\_indices], y[random\_indices])

y\_pred = stump.predict(X)

misclassified = np.array([int(i) for i in (y\_pred != y)])

error = np.sum(self.sample\_weights \* misclassified) / np.sum(self.sample\_weights)

stump.weight = 0.5 \* math.log((1 - error) / error)

self.sample\_weights[misclassified == 0] \*= np.exp(-stump.weight)

self.sample\_weights[misclassified == 1] \*= np.exp(stump.weight)

self.sample\_weights /= np.sum(self.sample\_weights)

self.stumps.append(stump)

def fit(self, X, y):

self.\_build\_forest(np.array(X), np.array(y))

def predict(self, X):

return np.array([self.\_make\_prediction(np.array(x)) for x in np.array(X)])

def \_make\_prediction(self, x):

stump\_preds = {}

for stump in self.stumps:

pred = stump.predict([x])[0]

if pred in stump\_preds:

stump\_preds[pred] += stump.weight

else:

stump\_preds[pred] = stump.weight

return max(stump\_preds, key=stump\_preds.get)

**decision\_tree.py:**

import numpy as np

# Node class

class Node:

# default arguments set to none since the node can be decision or leaf

def \_\_init\_\_(self, feature\_index=None, threshold=None, left=None, right=None, value=None):

# decision node

self.feature\_index = feature\_index

self.threshold = threshold

self.left = left

self.right = right

# leaf node

self.value = value

class DecisionTreeClassifier\_2:

def \_\_init\_\_(self, min\_samples\_split=2, max\_depth=float('inf'), criterion='gini'):

# will be set as the tree when fit function is run

self.root = None

self.min\_samples\_split = min\_samples\_split

self.max\_depth = max\_depth

self.criterion = criterion

self.tree\_depth = 0

def \_build\_tree(self, X, y, depth=1):

num\_samples = np.shape(X)[0]

# used later for print\_tree function

if depth > self.tree\_depth:

self.tree\_depth = depth

# if the current node is a decision node

if num\_samples >= self.min\_samples\_split and self.max\_depth >= depth:

# get the best split with the maximum information gain

best\_split = self.\_get\_best\_split(X, y)

if best\_split['information\_gain'] > 0:

# recursively generate the the subtrees

left\_subtree = self.\_build\_tree(depth=depth+1,\*\*best\_split['left'])

right\_subtree = self.\_build\_tree(depth=depth+1,\*\*best\_split['right'])

return Node(best\_split['feature\_index'], best\_split['threshold'], left\_subtree, right\_subtree)

Y = list(y)

value = max(Y, key=Y.count)

return Node(value=value)

def \_get\_best\_split(self, X, y):

num\_features = np.shape(X)[1]

max\_gain = -float('inf')

best\_split = {}

# iterates through each possible feature and threshold combination

for feature\_index in range(num\_features):

for threshold in np.unique(X[:, feature\_index]):

# condition for dividing the samples

left\_c = X[:, feature\_index] <= threshold

right\_c = X[:, feature\_index] > threshold

# divides all the samples into two nodes based on the feature\_index and threshold

left\_X, right\_X = X[left\_c], X[right\_c]

left\_y, right\_y = y[left\_c], y[right\_c]

# calculate the information gained by a unique split

information\_gain = self.\_get\_information\_gain(y, left\_y, right\_y)

if information\_gain > max\_gain:

best\_split['feature\_index'] = feature\_index

best\_split['threshold'] = threshold

best\_split['left'] = {'X': left\_X, 'y': left\_y}

best\_split['right'] = {'X': right\_X, 'y': right\_y}

best\_split['information\_gain'] = information\_gain

max\_gain = information\_gain

return best\_split

def \_get\_information\_gain(self, y, left, right):

l\_weight = len(left) / (len(left) + len(right))

r\_weight = 1 - l\_weight

if self.criterion == 'entropy':

return self.\_calculate\_entropy(y) - (l\_weight \* self.\_calculate\_entropy(left) + r\_weight \* self.\_calculate\_entropy(right))

elif self.criterion == 'gini':

return self.\_calculate\_gini(y) - (l\_weight \* self.\_calculate\_gini(left) + r\_weight \* self.\_calculate\_gini(right))

else:

raise ValueError('criterion can only be "entropy" or "gini"')

def \_calculate\_entropy(self, y):

classes = np.unique(y)

entropy = 0

for cls in classes:

p\_cls = len(y[y == cls]) / len(y)

entropy += -p\_cls \* np.log2(p\_cls)

return entropy

def \_calculate\_gini(self, y):

classes = np.unique(y)

gini = 0

for cls in classes:

p\_cls = len(y[y == cls]) / len(y)

gini += p\_cls \*\* 2

return 1 - gini

def fit(self, X, y):

self.column\_names = list(X.columns) if not isinstance(X, np.ndarray) else None

self.root = self.\_build\_tree(np.array(X), np.array(y))

def predict(self, X):

return [self.\_make\_prediction(x, self.root) for x in np.array(X)]

def \_make\_prediction(self, x, tree):

if tree.value != None:

return tree.value

if x[tree.feature\_index] <= tree.threshold:

return self.\_make\_prediction(x, tree.left)

else:

return self.\_make\_prediction(x, tree.right)

def \_print\_tree(self, tree, i=0):

if self.root == None:

print(f'Decision Tree has not been trained yet')

return

if tree == None:

return ''

if tree.value is not None: return tree.value

feature\_name = self.column\_names[tree.feature\_index]

return f'{feature\_name} <= {tree.threshold}\n' + \

f'{"|" \* i}left: {self.\_print\_tree(tree.left, i+1)}\n' + \

f'{"|" \* i}right: {self.\_print\_tree(tree.right, i+1)}'

def print\_tree(self):

print(self.\_print\_tree(self.root))