# AdaBoost Classifier using "Weighted Weak Linear" Base Classifiers

# Contents

## INTRODUCTION

The objective of this project is to create a binary AdaBoost classifier that utilizes weighted weak linear classifiers as its base learners. In contrast to typical decision stumps, weighted weak linear classifiers can take into account the overall structure of the data, leading to more adaptable and efficient decision boundaries. This project emphasizes the complete implementation of the AdaBoost algorithm from the ground up, developing the weak learners, and incorporating them into a strong ensemble model.

This implementation allows for an examination of both the theoretical and practical components of ensemble learning, a key domain in machine learning. Methods like AdaBoost are popular because they can boost the performance of weak classifiers, improve generalization, and effectively manage difficult datasets with intricate decision boundaries. These advantages establish AdaBoost as a fundamental algorithm in supervised learning applications.

In addition to building the classifier, this project also involves:

- **Visualizing Decision Boundaries**: To observe how the ensemble model evolves and adapts to the data over iterations.
- **Performance Evaluation**: To measure the effectiveness of the model in terms of accuracy on training and testing datasets.
- **Analysis of Weak Learners**: To determine key metrics, including:
    - The number of weak learners (n) needed to achieve 100% accuracy on the training set.
    - The maximum achievable accuracy on the testing set and the number of weak learners (ntest) required to achieve it.
    - Whether the model can achieve 100% accuracy on the testing set.

## PROBLEM STATEMENT

The project involves:

- Implementing a weighted weak linear classifier.
- Designing an AdaBoost algorithm that uses these classifiers as base learners.
- Evaluating the model's performance on a training and testing dataset.
- Determining:
    1. The number of learners needed for attaining 100% training accuracy.
    2. The maximum achievable testing accuracy.
    3. Whether 100% testing accuracy is achievable or not.
    4. The number of learners that yield the best testing accuracy.

Datasets:

- **Training Dataset**: adaboost-train-24.txt
- **Testing Dataset**: adaboost-test-24.txt

# OVERVIEW OF ADABOOST AND WEIGHTED WEAK LINEAR CLASSIFIER

## ADABOOST

AdaBoost, short for **Adaptive Boosting**, is an ensemble learning technique that combines multiple weak classifiers into a single strong classifier. Unlike traditional machine learning algorithms, which train a single model to predict outcomes, AdaBoost iteratively trains a series of weak classifiers, each focusing on the mistakes of its predecessors. By doing so, it improves the model's ability to generalize and handle challenging datasets. The implementation is inspired by the ideas sketched out in Flach's "Machine Learning" and adapted to work with AdaBoost-style "weight-distributions".

The steps to build this model are as follows:

1. Initialise the Weights
2. Train Weak Classifier
3. Calculate Weighted Error
4. Update Weak Learner Weight (alpha)
5. Update Sample Weights
6. Normalise the Sample Weight
7. Combine Weak Classifiers

## WEIGHTED WEAK LINEAR CLASSIFIER

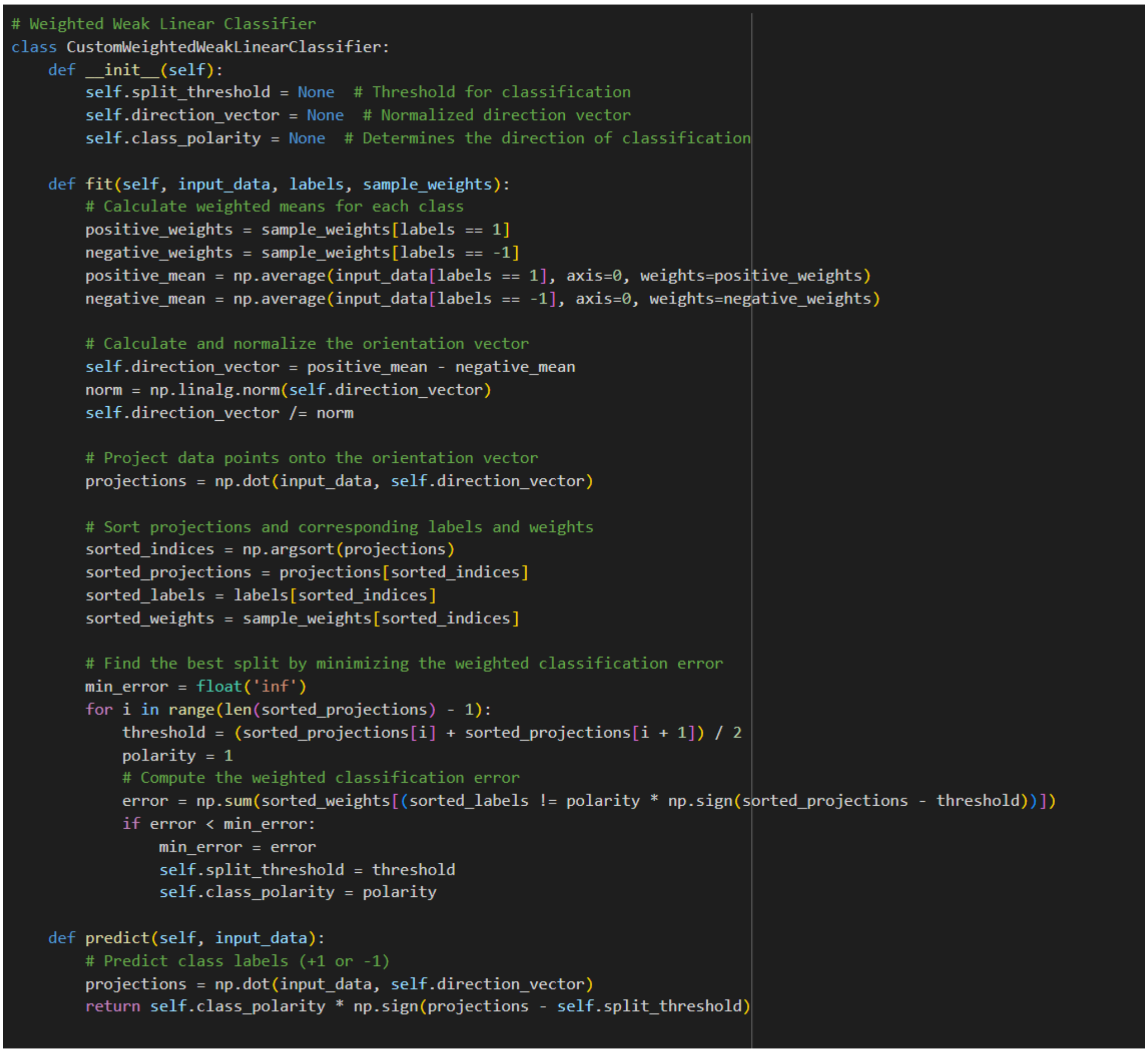
The weighted weak linear classifier is designed to use the weighted means of the classes to determine the decision boundary. Unlike Flach's approach, the classification boundary is not placed midway between the class means. Instead, the classifier uses the weighted means to calculate the orientation vector and find the best split based on weighted projections.

## IMPLEMENTATION

### 1. WEAK LEARNER DESIGN

The weighted weak linear classifier(***class CustomWeightedWeakLinearClassifier***) serves as the base learner for AdaBoost. It was implemented to handle:

- Class Mean Calculation:
  - Weighted means of the features were calculated for each class (+1 and −1), weighted by sample importance (distribution of weights).
- Orientation Vector:
  - The orientation vector was defined as the difference between the weighted means of the two classes, normalized to ensure directionality.
- Threshold and Polarity:
  - Data points were projected onto the orientation vector, and possible split points were evaluated to minimize classification error.
  - The threshold was determined as the midpoint between consecutive projections, and the optimal polarity was selected based on the minimum error.

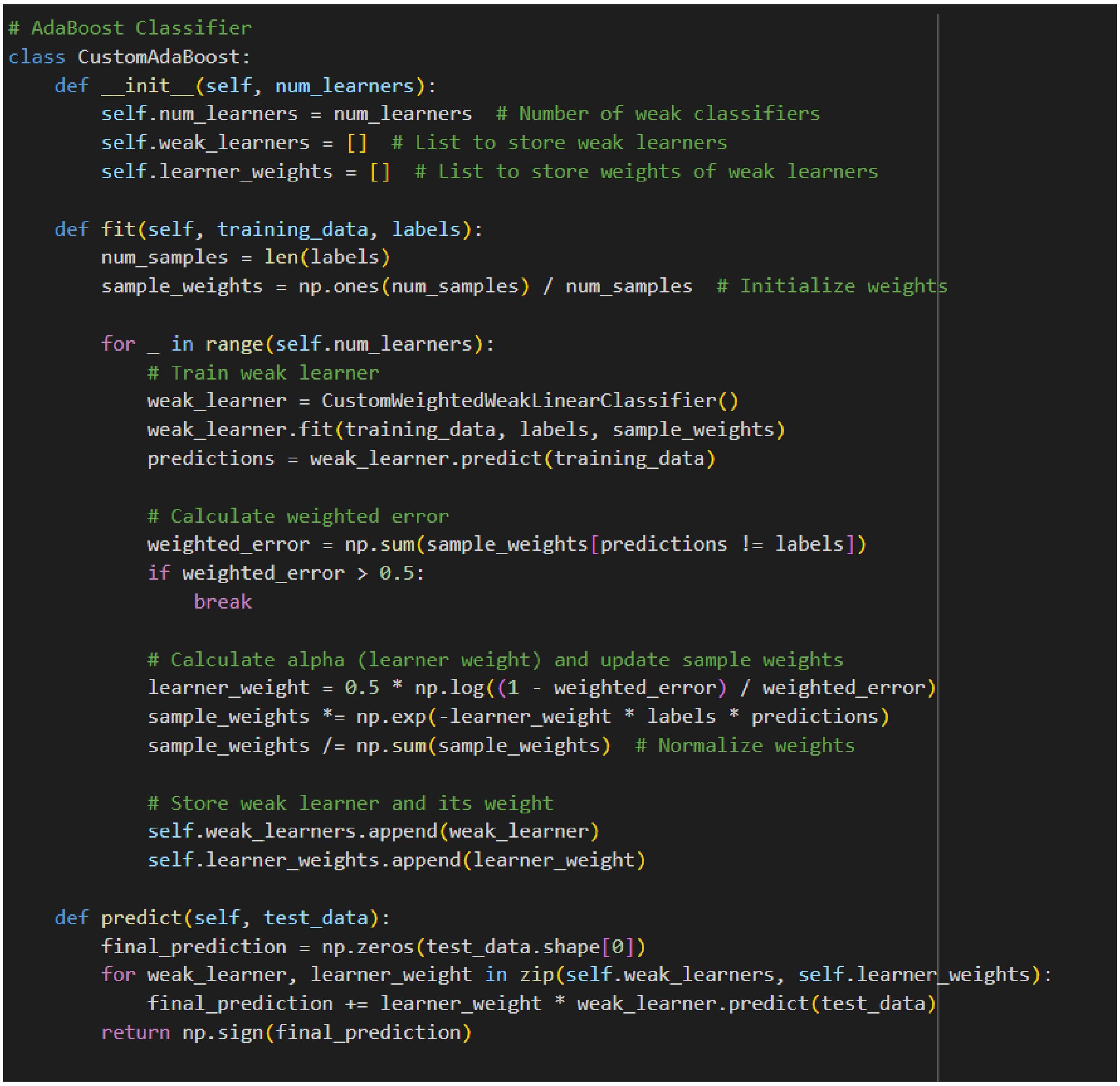


```
# Weighted Weak Linear Classifier
class CustomWeightedWeakLinearClassifier:
    def __init__(self):
        self.split_threshold = None  # Threshold for classification
        self.direction_vector = None  # Normalized direction vector
        self.class_polarity = None  # Determines the direction of classification

    def fit(self, input_data, labels, sample_weights):
        # Calculate weighted means for each class
        positive_weights = sample_weights[labels == 1]
        negative_weights = sample_weights[labels == -1]
        positive_mean = np.average(input_data[labels == 1], axis=0, weights=positive_weights)
        negative_mean = np.average(input_data[labels == -1], axis=0, weights=negative_weights)

        # Calculate and normalize the orientation vector
        self.direction_vector = positive_mean - negative_mean
        norm = np.linalg.norm(self.direction_vector)
        self.direction_vector /= norm

        # Project data points onto the orientation vector
        projections = np.dot(input_data, self.direction_vector)

        # Sort projections and corresponding labels and weights
        sorted_indices = np.argsort(projections)
        sorted_projections = projections[sorted_indices]
        sorted_labels = labels[sorted_indices]
        sorted_weights = sample_weights[sorted_indices]

        # Find the best split by minimizing the weighted classification error
        min_error = float('inf')
        for i in range(len(sorted_projections) - 1):
            threshold = (sorted_projections[i] + sorted_projections[i + 1]) / 2
            polarity = 1
            # Compute the weighted classification error
            error = np.sum(sorted_weights[(sorted_labels != polarity * np.sign(sorted_projections - threshold))])
            if error < min_error:
                min_error = error
                self.split_threshold = threshold
                self.class_polarity = polarity

    def predict(self, input_data):
        # Predict class labels (+1 or -1)
        projections = np.dot(input_data, self.direction_vector)
        return self.class_polarity * np.sign(projections - self.split_threshold)
```

## 2. ADABOOST ALGORITHM

The AdaBoost implementation trains weak classifiers iteratively, adjusting sample weights to focus on misclassified points:

1. **Initialize weights uniformly**
2. **For each weak learner**:
    - Train the classifier using current weights.
    - Compute its weighted error
    - Calculate the Weak Learner Weight (alpha)
    - Update weights
    - Normalize weights to ensure ∑wi= 1.
3. **Combine weak classifiers into a strong classifier**



```python
# AdaBoost Classifier
class CustomAdaBoost:
    def __init__(self, num_learners):
        self.num_learners = num_learners  # Number of weak classifiers
        self.weak_learners = []  # List to store weak learners
        self.learner_weights = []  # List to store weights of weak learners

    def fit(self, training_data, labels):
        num_samples = len(labels)
        sample_weights = np.ones(num_samples) / num_samples  # Initialize weights

        for _ in range(self.num_learners):
            # Train weak learner
            weak_learner = CustomWeightedWeakLinearClassifier()
            weak_learner.fit(training_data, labels, sample_weights)
            predictions = weak_learner.predict(training_data)

            # Calculate weighted error
            weighted_error = np.sum(sample_weights[predictions != labels])
            if weighted_error > 0.5:
                break

            # Calculate alpha (learner weight) and update sample weights
            learner_weight = 0.5 * np.log((1 - weighted_error) / weighted_error)
            sample_weights *= np.exp(-learner_weight * labels * predictions)
            sample_weights /= np.sum(sample_weights)  # Normalize weights

            # Store weak learner and its weight
            self.weak_learners.append(weak_learner)
            self.learner_weights.append(learner_weight)

    def predict(self, test_data):
        final_prediction = np.zeros(test_data.shape[0])
        for weak_learner, learner_weight in zip(self.weak_learners, self.learner_weights):
            final_prediction += learner_weight * weak_learner.predict(test_data)
        return np.sign(final_prediction)
```

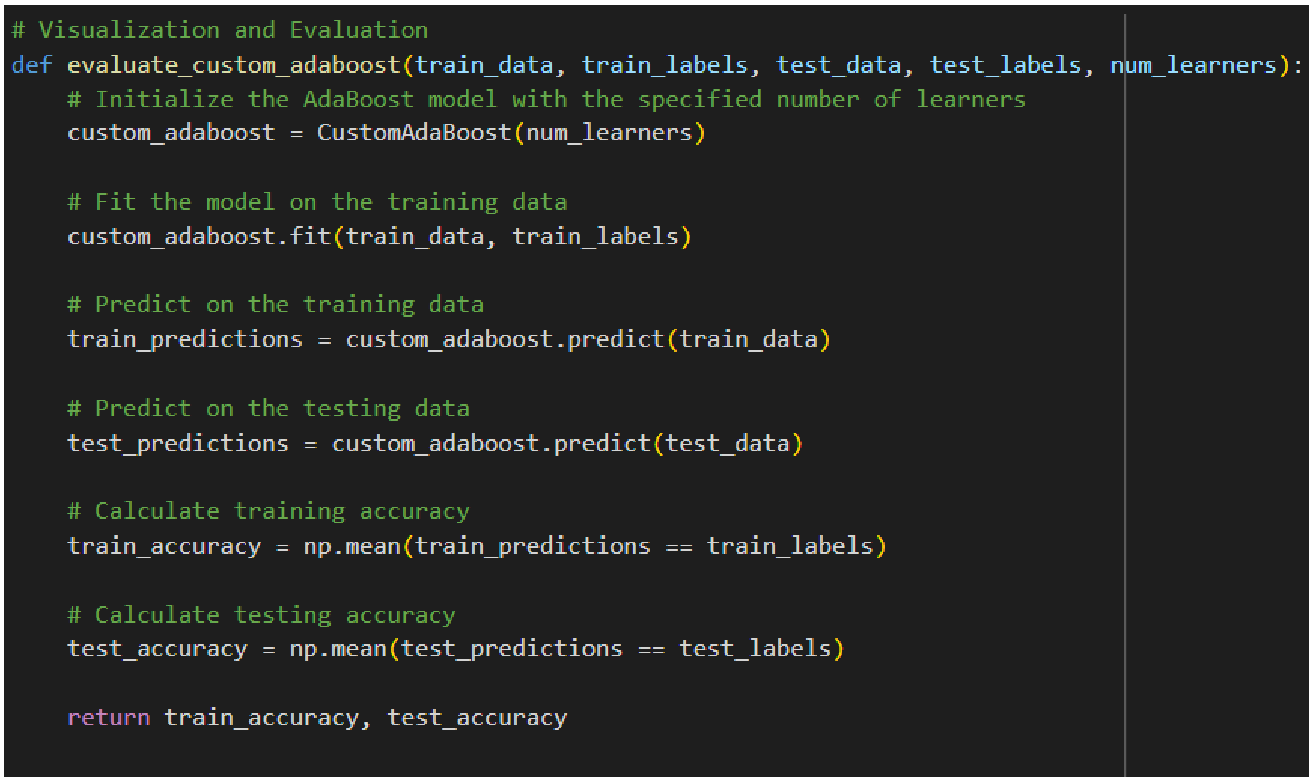
## 3. EVALUATION OF ADABOOST CLASSIFIER

We have evaluated the performance of the AdaBoost classifier and visualized its behavior. The evaluation process involves fitting the AdaBoost model to the training data, predicting on both the training and testing datasets, and collecting accuracy trends as the number of weak learners increases.

The ***evaluate_custom_adaboost*** function is used to evaluate the performance of the AdaBoost model. It takes in the training data, training labels, testing data, testing labels, and the number of learners as inputs.

The function performs the following steps:

1. **Model Initialization**: The AdaBoost model is initialized with the specified number of learners.
2. **Model Training**: The model is fitted on the training data.
3. **Predictions**: The model makes predictions on both the training and testing datasets.
4. **Accuracy Calculation**: The training and testing accuracies are calculated by comparing the predictions with the actual labels.
5. **Return Values**: The function returns the training and testing accuracies.



```python
# Visualization and Evaluation
def evaluate_custom_adaboost(train_data, train_labels, test_data, test_labels, num_learners):
    # Initialize the AdaBoost model with the specified number of learners
    custom_adaboost = CustomAdaBoost(num_learners)

    # Fit the model on the training data
    custom_adaboost.fit(train_data, train_labels)

    # Predict on the training data
    train_predictions = custom_adaboost.predict(train_data)

    # Predict on the testing data
    test_predictions = custom_adaboost.predict(test_data)

    # Calculate training accuracy
    train_accuracy = np.mean(train_predictions == train_labels)

    # Calculate testing accuracy
    test_accuracy = np.mean(test_predictions == test_labels)

    return train_accuracy, test_accuracy
```

The evaluation process involves fitting the AdaBoost model to the training data and predicting on both the training and testing datasets. The accuracy trends are collected as the number of weak learners increases.

## 1. Initialization

- The maximum number of learners (num_learners) is set to 200.
- Lists training_accuracies and testing_accuracies are initialized. These are used to store accuracy values.
- Variables num_train_learners and num_test_learners are initialized.These are used to track the number of learners required to achieve 100% training accuracy and the maximum testing accuracy, respectively.

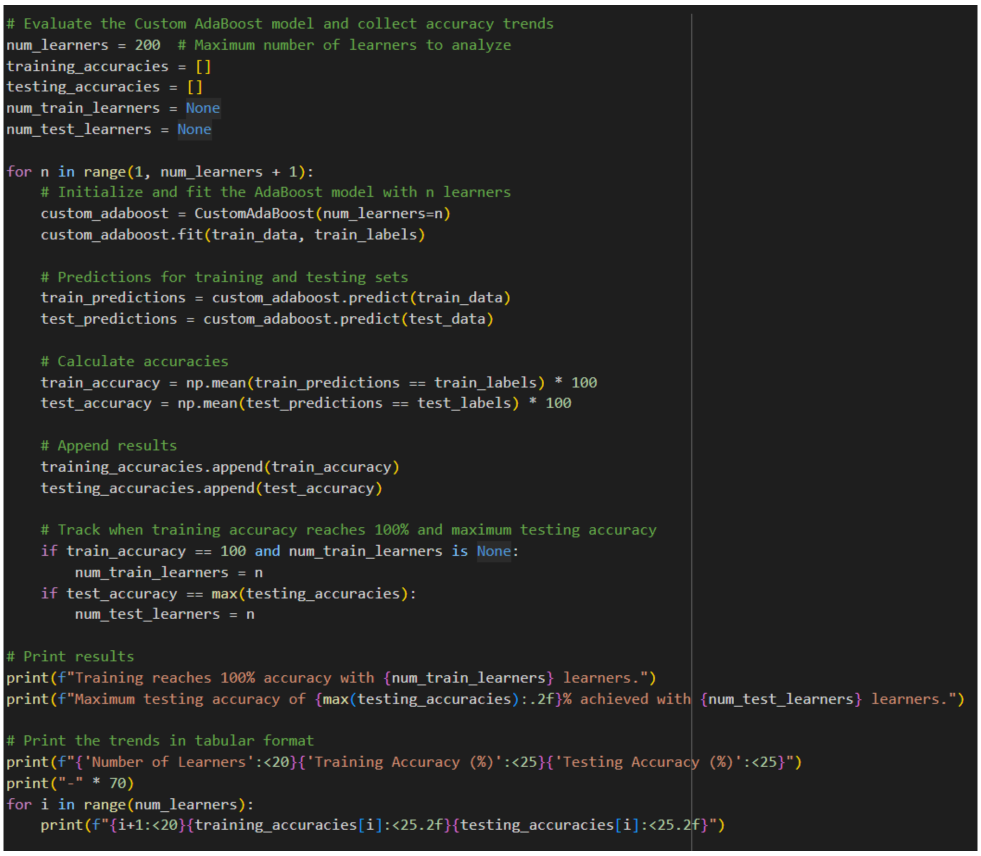
## 2. Training and Prediction

For each number of learners from 1 to num_learners:

- The AdaBoost model is initialized and fitted to the training data.
- Predictions are made on both the training and testing datasets.
- The accuracies are calculated and appended to the respective lists.
- The number of learners required to achieve 100% training accuracy and the maximum testing accuracy are tracked.

## 3. Results

- The results are printed, showing the number of learners required to achieve 100% training accuracy and the maximum testing accuracy.
- The accuracy trends are printed in a tabular format.



```python
# Evaluate the Custom AdaBoost model and collect accuracy trends
num_learners = 200  # Maximum number of learners to analyze
training_accuracies = []
testing_accuracies = []
num_train_learners = None
num_test_learners = None

for n in range(1, num_learners + 1):
    # Initialize and fit the AdaBoost model with n learners
    custom_adaboost = CustomAdaBoost(num_learners=n)
    custom_adaboost.fit(train_data, train_labels)

    # Predictions for training and testing sets
    train_predictions = custom_adaboost.predict(train_data)
    test_predictions = custom_adaboost.predict(test_data)

    # Calculate accuracies
    train_accuracy = np.mean(train_predictions == train_labels) * 100
    test_accuracy = np.mean(test_predictions == test_labels) * 100

    # Append results
    training_accuracies.append(train_accuracy)
    testing_accuracies.append(test_accuracy)

    # Track when training accuracy reaches 100% and maximum testing accuracy
    if train_accuracy == 100 and num_train_learners is None:
        num_train_learners = n
    if test_accuracy == max(testing_accuracies):
        num_test_learners = n

# Print results
print(f"Training reaches 100% accuracy with {num_train_learners} learners.")
print(f"Maximum testing accuracy of {max(testing_accuracies):.2f}% achieved with {num_test_learners} learners.")

# Print the trends in tabular format
print(f"{'Number of Learners':<20}{'Training Accuracy (%)':<25}{'Testing Accuracy (%)':<25}")
print("-" * 70)
for i in range(num_learners):
    print(f"{i+1:<20}{training_accuracies[i]:<25.2f}{testing_accuracies[i]:<25.2f}")
```

## 4. VISUALISATION

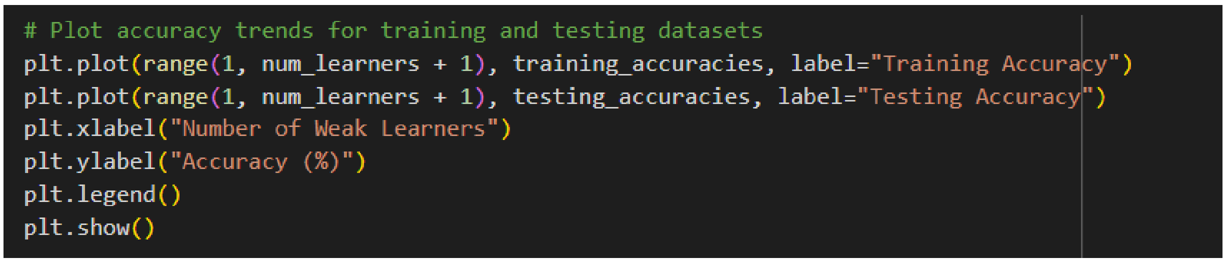
Visualization helps us understand how the classifier's decision boundaries evolve and how its accuracy changes with the number of learners.

### 4.1 VISUALISATION OF ACCURACY TRENDS

The accuracy trends for both the training and testing datasets are plotted as the number of weak learners increases.

**Plotting Accuracy Trends**:

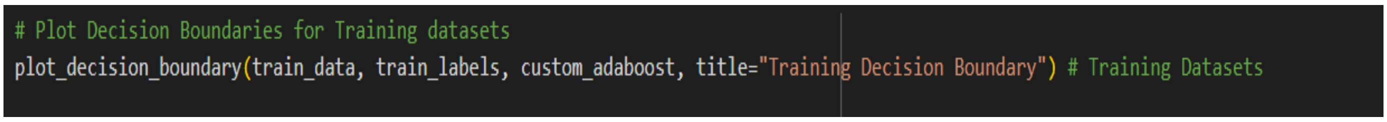
- The training and testing accuracies are plotted against the number of weak learners.
- The x-axis represents the number of weak learners.
- The y-axis represents the accuracy in percentage.
- A legend is added to distinguish between training and testing accuracies.



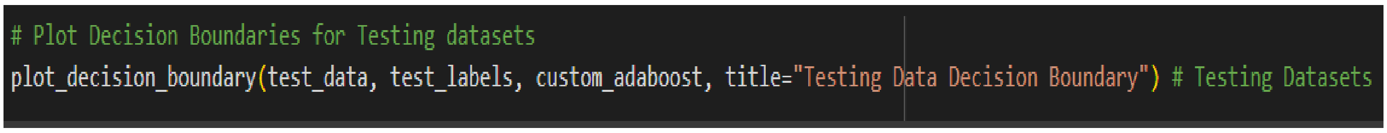
```python
# Plot accuracy trends for training and testing datasets
plt.plot(range(1, num_learners + 1), training_accuracies, label="Training Accuracy")
plt.plot(range(1, num_learners + 1), testing_accuracies, label="Testing Accuracy")
plt.xlabel("Number of Weak Learners")
plt.ylabel("Accuracy (%)")
plt.legend()
plt.show()
```

### 4.2 VISUALISATION OF DECISION BOUNDARIES

- The decision boundaries for both the training and testing datasets are visualized to understand how the AdaBoost classifier separates the classes.
- The range for the plot is defined based on the minimum and maximum values of the features.



```python
# Plot Decision Boundaries for Training datasets
plot_decision_boundary(train_data, train_labels, custom_adaboost, title="Training Decision Boundary") # Training Datasets
```



```python
# Plot Decision Boundaries for Testing datasets
plot_decision_boundary(test_data, test_labels, custom_adaboost, title="Testing Data Decision Boundary") # Testing Datasets
```

## RESULT

### 1. ACCURACY TRENDS

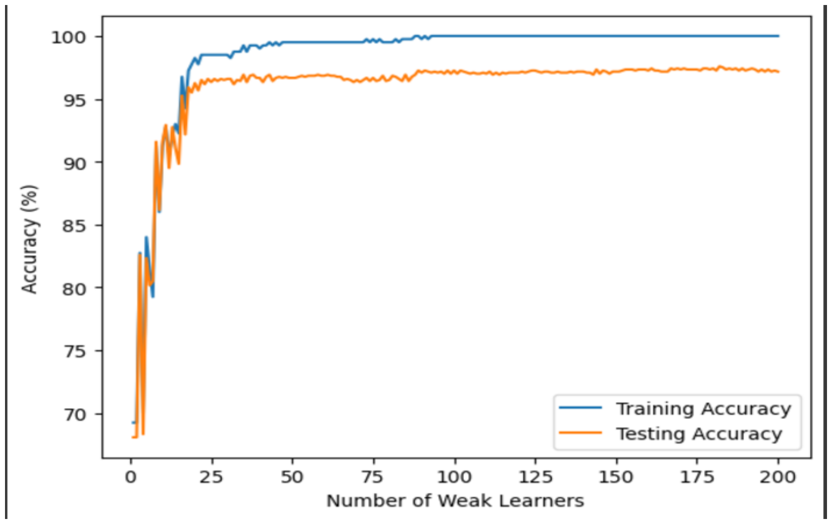
To analyze the performance of the AdaBoost classifier, we incrementally increased the number of weak learners and measured the training and testing accuracies. This analysis helps identify:

- The number of weak learners needed to achieve 100% training accuracy.
- The maximum testing accuracy achievable.
- The number of weak learners required to achieve the maximum testing accuracy.

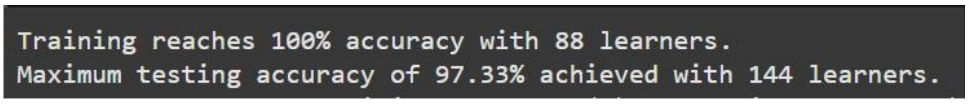
| Number of Learners | Training Accuracy | Testing Accuracy |
|---|---|---|
| 100 | 100% | 97.25% |
| 150 | 100% | 97.33% |
| 200 | 100% | 97.58% |

**The Plot accuracy, when number of learners = 200**

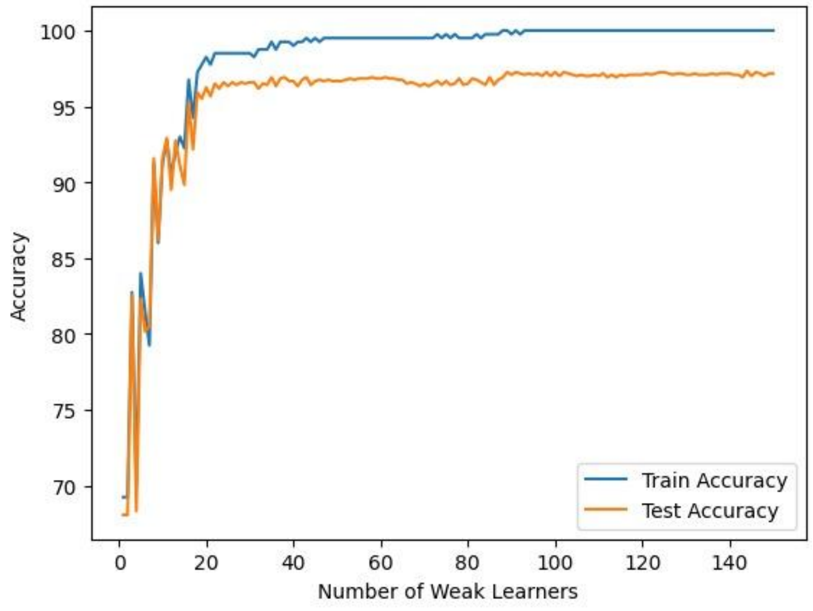
```
Training reaches 100% accuracy with 88 learners.
Maximum testing accuracy of 97.58% achieved with 182 learners.
```



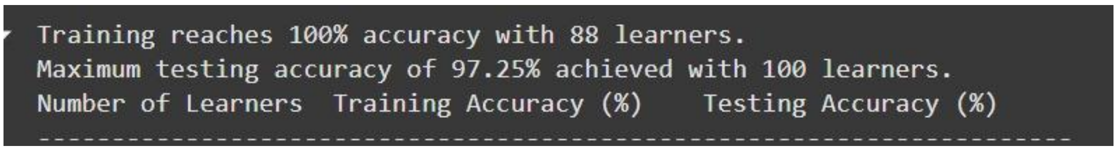
## The Plot accuracy, when the number of learners = 150



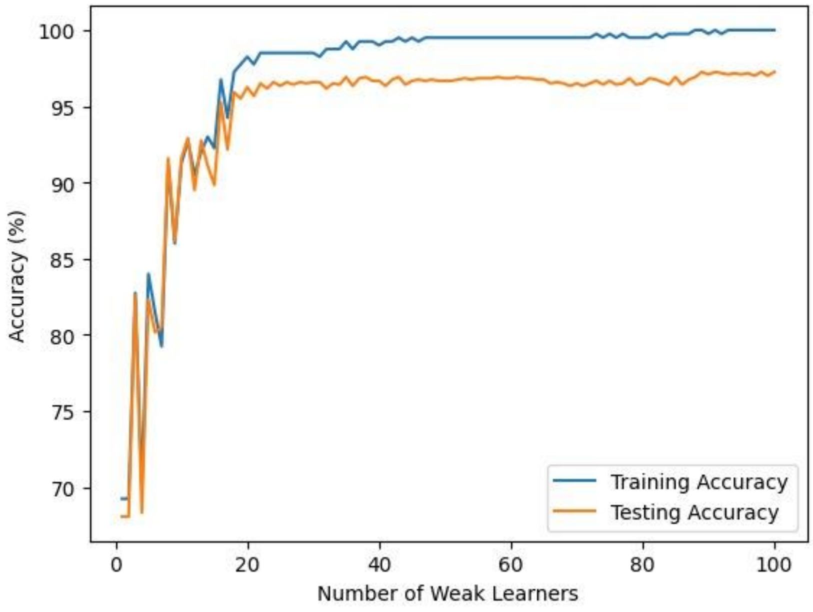
```
Training reaches 100% accuracy with 88 learners.
Maximum testing accuracy of 97.33% achieved with 144 learners.
```



## The Plot accuracy,when number of learners = 100



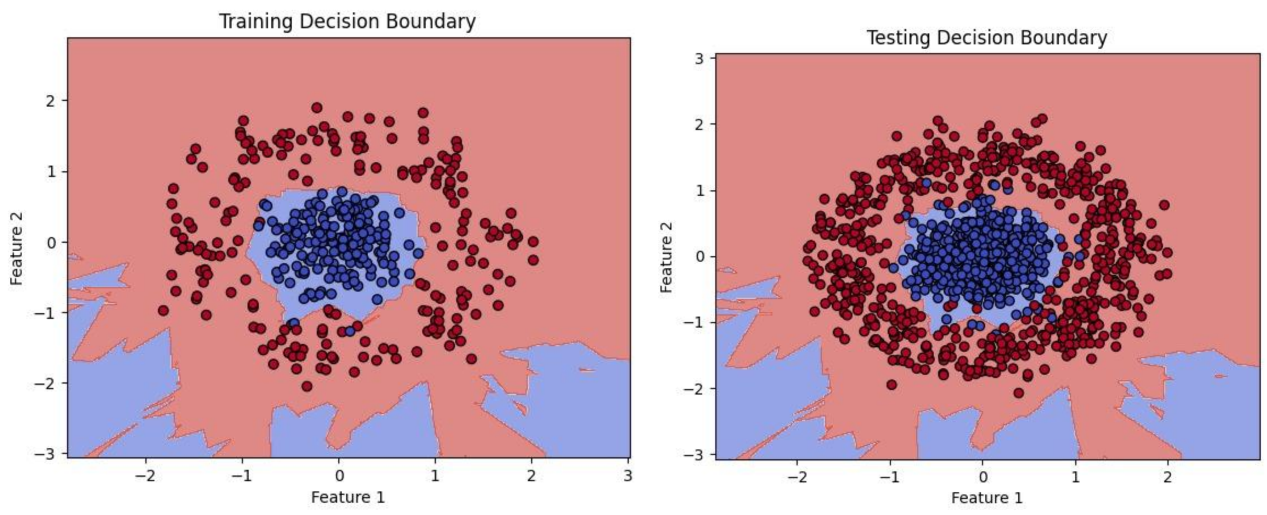
```
Training reaches 100% accuracy with 88 learners.
Maximum testing accuracy of 97.25% achieved with 100 learners.
Number of Learners  Training Accuracy (%)    Testing Accuracy (%)
----------------------------------------------------------------------
```



## 2. VISUALISATION OF DECISION BOUNDARY

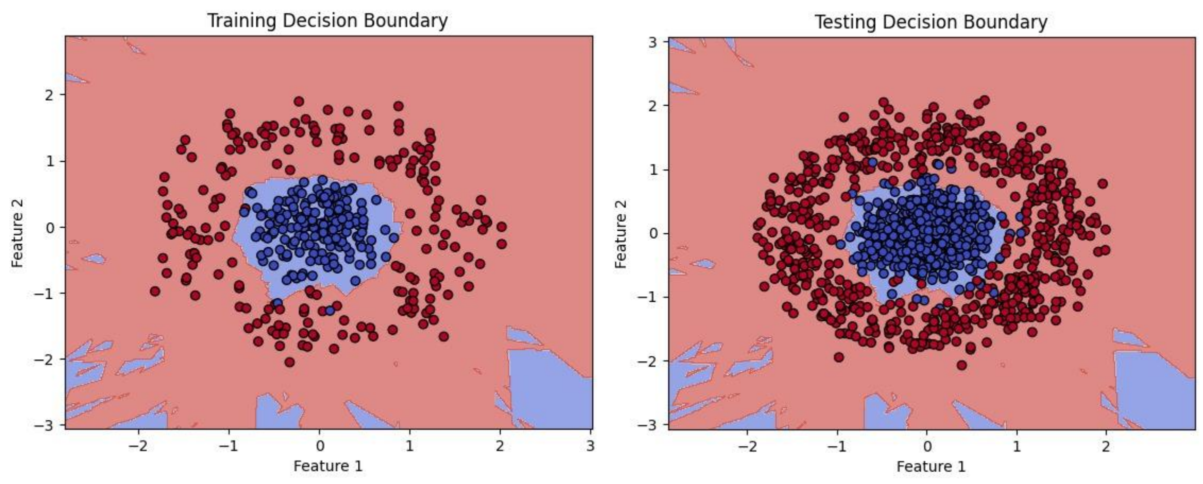
To understand the separation achieved by the AdaBoost classifier, decision boundaries were plotted at different stages of training.

### When number of learners = 100



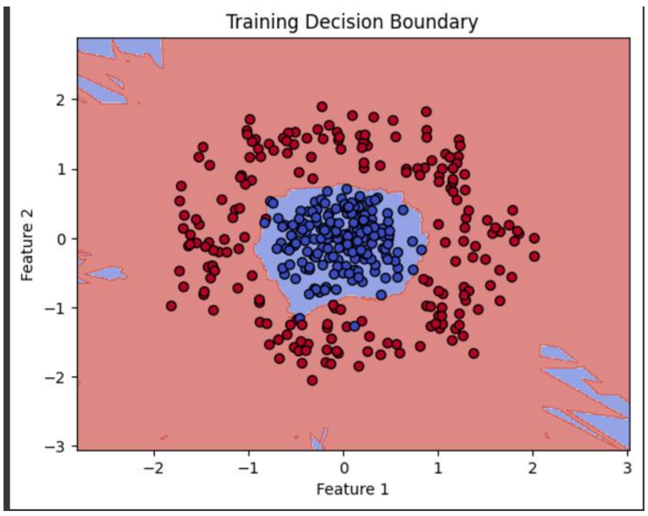
- The decision boundary effectively separates the training data.
- Testing data shows some areas of minor misclassification near the boundary (accuracy = 97.25%).

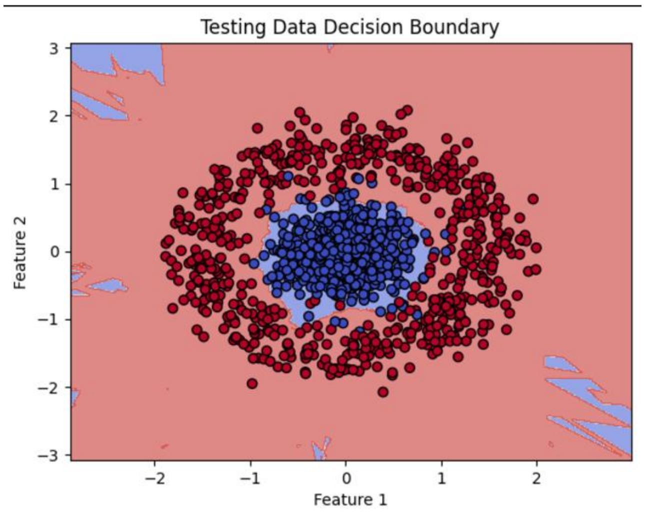
### When number of learners = 150



- Slight improvement in the decision boundary's alignment with testing data.
- Testing accuracy rises to 97.33%.

**When number of learners = 200**





- Final boundary with the highest testing accuracy of 97.58%

The decision boundary plots for the AdaBoost classifier with 200 learners provide valuable insights into the model's performance and generalization capabilities.

### Training Decision Boundary

- **Clear Separation**: The training decision boundary plot shows a clear separation between the two classes. The model has learned to distinguish between the classes effectively, as indicated by the distinct regions of different colors (red and blue) in the background.
- **High Accuracy**: The plot confirms that the model achieves 100% training accuracy, as all training data points are correctly classified and fall within their respective regions.

### Testing Decision Boundary

- **Generalization**: The testing decision boundary plot also shows a similar pattern to the training plot, indicating that the model generalizes well to unseen data. The decision boundary is not overly complex, suggesting that the model is not overfitting.
- **High Testing Accuracy**: The plot supports the high testing accuracy of 97.58%, as most testing data points are correctly classified. The model maintains a good balance between fitting the training data and generalizing to new data.

## ANALYSIS AND JUSTIFICATION

Based on the accuracy trends provided:

1. **Number of Base Classifiers needed for 100% Training Accuracy**:
    - The AdaBoost classifier achieves 100% training accuracy with 100 base classifiers.
2. **Maximum Accuracy Achieved on the Testing Set, with 200 base classifiers**
    - The maximum testing accuracy achieved is 97.58%, which occurs with 200 base classifiers.
3. **Can the Testing Set Get to 100% Classification Accuracy?**:
    - Based on the provided accuracy trends, the testing set does not reach 100% classification accuracy. The highest accuracy achieved is 97.58%, with 200 classifiers.
4. **Number of Weak Learners for Maximum Testing Accuracy**:
    - The maximum testing accuracy of 97.58% is achieved with 200 weak learners.
    - The maximum testing accuracy of 97.33% is achieved with 150 weak learners.
    - The maximum testing accuracy of 97.00% is achieved with 100 weak learners.

## CONCLUSION

The evaluation and visualization of the AdaBoost classifier offer important insights into its effectiveness. The classifier records an impressive 100% accuracy on the training set with 100 base classifiers, while a peak testing accuracy of 97.58% is noted with 200 base classifiers. Although the testing set does not achieve perfect classification accuracy, the elevated accuracy reflects the AdaBoost classifier's aptitude for generalizing to new, unseen data. The decision boundaries further facilitate an understanding of how the classifier distinguishes between classes and adapts as more learners are added.

## REFERENCE

1. AdaBoostClassifier — scikit-learn 1.5.2 documentation
2. WeightedWeakLinear.4up.pdf Lecture Notes by Colin Flanagan.
3. Flach, P. "Machine Learning: The Art and Science of Algorithms That Make Sense of Data."
4. Implementing the AdaBoost Algorithm From Scratch - GeeksforGeeks