def \_\_init\_\_(self, base\_estimator=None, n\_estimators=50):  
 self.estimator\_weights\_ = None  
 self.base\_estimator = base\_estimator if base\_estimator else DecisionTreeClassifier(max\_depth=1)  
 self.n\_estimators = n\_estimators  
 self.estimators = []  
 self.estimator\_weights = []

**"estimator" refers to any algorithm or model that takes input data (features) and learns patterns or relationships from that data to make predictions or estimate unknown outcomes.**

**self.estimator\_weights\_:**

* **Purpose**: This attribute is used to store the weights associated with each weak learner (base estimator) in the AdaBoost ensemble.
* **Job**: During the AdaBoost training process, the weights of the weak learners are calculated based on their performance in each iteration. The better the weak learner performs, the higher its weight in the ensemble. These weights are then used to combine the predictions of the weak learners during the final classification step.

**self.base\_estimator:**

* **Purpose**: This attribute represents the base estimator used to build the ensemble of weak learners.
* **Job**: The base\_estimator is the core weak learner that is trained in each iteration of AdaBoost. By default, if the base\_estimator argument is not provided during object creation, a decision tree with max\_depth=1 (stump) is used as the default base estimator. You can provide your own base estimator (e.g., any scikit-learn classifier) to use in the ensemble.

self.n\_estimators:

* **Purpose**: This attribute defines the number of weak learners (base estimators) in the AdaBoost ensemble.
* **Job**: The n\_estimators specifies how many iterations the AdaBoost algorithm will run, with each iteration adding a new weak learner to the ensemble. A higher number of estimators can improve the performance of the ensemble but may also increase computation time.

self.estimators:

* **Purpose**: This attribute is a list that stores the fitted weak learners (base estimators) during the training process.
* **Job**: In each iteration of AdaBoost, a new weak learner is fitted to the training data using the current sample weights. After each iteration, the newly trained weak learner is added to this list. The final ensemble of weak learners is formed by combining all the weak learners in this list.

self.estimator\_weights:

* **Purpose**: This attribute is a list that stores the weights associated with each weak learner (base estimator) during the training process.
* **Job**: In each iteration of AdaBoost, a weight (alpha) is calculated based on the performance of the weak learner on the training data. This weight represents the importance of the weak learner in the final ensemble. The weight is then used to adjust the sample weights for the next iteration. The estimator weights are stored in this list for later use during prediction.

@staticmethod  
def \_fit\_estimator(estimator, X, y, sample\_weights):  
 *"""  
 Fits the given estimator on the training data with weighted samples.  
  
 """* if hasattr(estimator, 'fit'):  
 estimator.fit(X, y, sample\_weight=sample\_weights)  
 else:  
 raise AttributeError("Base estimator does not have 'fit' method.")

**hasattr(object, attribute\_name) :** **function used to check whether an object has a particular attribute or method, returning true or false.**

* This line checks if the estimator object has a method called fit.
* hasattr(estimator, 'fit') checks whether the estimator object has an attribute named 'fit'. In Python, methods are also attributes of an object.
* If the estimator has the fit method, it means it is a valid model that can be trained.
* If it has the fit method, the function proceeds to call the fit method of the estimator with the training data X, target labels y, and the sample weights sample\_weights.

@staticmethod  
def \_predict\_estimator(estimator, X):  
 *"""  
 Predicts using the given estimator.  
  
 """* if hasattr(estimator, 'predict'):  
 return estimator.predict(X)  
 else:  
 raise AttributeError("Base estimator does not have 'predict' method.")

This line checks if the estimator object has a method called predict.

If the estimator has the predict method, it means it is a valid model that can be used for making predictions.

If it has the predict method, the function proceeds to call the predict method of the estimator with the input data X, and it returns the predictions made by the model.

def fit(self, X, y):  
 *"""  
 Fits the AdaBoost classifier on the training data.  
  
 """* n\_samples = len(X)  
 sample\_weights = np.ones(n\_samples) / n\_samples  
  
 self.estimator\_weights\_ = [] # Initialize the estimator\_weights\_ attribute as an empty list  
  
 for \_ in range(self.n\_estimators):  
 estimator = clone(self.base\_estimator)  
 self.\_fit\_estimator(estimator, X, y, sample\_weights)  
 y\_pred = self.\_predict\_estimator(estimator, X)  
  
 err = np.sum(sample\_weights \* (y\_pred != y)) / np.sum(sample\_weights)  
  
 if err >= 0.5: # Modified condition to handle cases where err == 0.5  
 break  
  
 alpha = 0.5 \* np.log((1.0 - err) / err)  
  
 sample\_weights \*= np.exp(-alpha \* y \* y\_pred)  
 sample\_weights /= np.sum(sample\_weights)  
  
 self.estimators.append(estimator)  
 self.estimator\_weights\_.append(alpha) # Append the alpha to estimator\_weights\_  
  
 # Ensure the number of estimator weights matches the number of fitted estimators  
 num\_fitted\_estimators = len(self.estimators)  
 num\_estimator\_weights = len(self.estimator\_weights\_)  
 if num\_fitted\_estimators < num\_estimator\_weights:  
 self.estimator\_weights\_ = self.estimator\_weights\_[:num\_fitted\_estimators]  
 elif num\_fitted\_estimators > num\_estimator\_weights:  
 self.estimator\_weights\_ += [0.0] \* (num\_fitted\_estimators - num\_estimator\_weights)  
  
 # If no fitted estimators, set estimator\_weights\_ to an empty list  
 if not self.estimators:  
 self.estimator\_weights\_ = []

**n\_samples = len(X)**

**sample\_weights = np.ones(n\_samples) / n\_samples**

* This line calculates the total number of samples in the training data and stores it in the variable n\_samples.
* It initializes the sample\_weights as an array of ones (with the same length as the number of samples) and then divides each weight by the total number of samples. This step ensures that the initial weights are uniformly distributed.

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

**estimator = clone(self.base\_estimator)**

**self.\_fit\_estimator(estimator, X, y, sample\_weights)**

**y\_pred = self.\_predict\_estimator(estimator, X)**

* This part is the main loop of the AdaBoost training process. It iterates self.n\_estimators times to create an ensemble of weak learners.
* In each iteration, a new weak learner (estimator) is created by cloning (clone) the base\_estimator, which allows training the same estimator multiple times with different sample weights.
* The \_fit\_estimator method is then called to fit the weak learner (estimator) on the training data with the current sample weights.
* After fitting the estimator, the \_predict\_estimator method is used to obtain predictions (y\_pred) on the training data using the trained estimator.

**err = np.sum(sample\_weights \* (y\_pred != y)) / np.sum(sample\_weights)**

* This line calculates the weighted error of the current weak learner (estimator).
* It calculates the sum of the sample weights for the misclassified samples (where y\_pred is not equal to y) and divides it by the sum of all the sample weights.

**sample\_weights \*= np.exp(-alpha \* y \* y\_pred)**

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

These lines update the sample weights for the next iteration of the loop.

**num\_fitted\_estimators = len(self.estimators)**

**num\_estimator\_weights = len(self.estimator\_weights\_)**

**if num\_fitted\_estimators < num\_estimator\_weights:**

**self.estimator\_weights\_ = self.estimator\_weights\_[:num\_fitted\_estimators]**

**elif num\_fitted\_estimators > num\_estimator\_weights:**

**self.estimator\_weights\_ += [0.0] \* (num\_fitted\_estimators - num\_estimator\_weights)**

* After the loop, this section ensures that the self.estimator\_weights\_ list matches the number of fitted estimators (self.estimators).
* If the number of fitted estimators is less than the number of estimator weights (which can happen if the loop is terminated early due to high error), the extra estimator weights are removed from self.estimator\_weights\_.
* If the number of fitted estimators is greater than the number of estimator weights (which can happen if the loop is terminated early), additional zeros are added to self.estimator\_weights\_ to keep the lengths equal.

def predict(self, X):  
 *"""  
 Predicts the target labels using the fitted AdaBoost classifier.  
  
 """* y\_pred = np.zeros(len(X))  
 for i, estimator in enumerate(self.estimators):  
 if i < len(self.estimator\_weights):  
 y\_pred += self.estimator\_weights[i] \* self.\_predict\_estimator(estimator, X)  
 else:  
 break  
  
 return np.sign(y\_pred)

np.sign :  
**If the input element is greater than 0, np.sign returns 1.**

**If the input element is less than 0, np.sign returns -1.**

**If the input element is equal to 0, np.sign returns 0.**

* **This part is the main loop of the prediction process.**
* **It iterates through the fitted weak learners stored in self.estimators and their corresponding weights in self.estimator\_weights\_.**
* **For each weak learner, it calculates the weighted predictions (y\_pred) on the input data X.**
* **The weighted prediction for a specific weak learner is obtained by multiplying the weight of that weak learner (self.estimator\_weights[i]) with the predictions made by that weak learner on X.**
* **The loop continues until either all weak learners have been used (i.e., i < len(self.estimator\_weights)) or until it reaches the end of the fitted weak learners.**
* **If there are fewer weights than weak learners (i.e., the loop is terminated early), it means that some weak learners were not included in the ensemble due to poor performance.**
* **After the loop, the final predicted labels are obtained by taking the sign of the sum of the weighted predictions (y\_pred). In this context, taking the sign of the sum effectively converts the continuous weighted predictions to binary labels (-1 or 1).**

def predict\_proba(self, X):  
 *"""  
 Predicts class probabilities for binary classification.  
  
 """* n\_samples = len(X)  
 proba = np.zeros((n\_samples, 2))  
 for i, estimator in enumerate(self.estimators):  
 if i < len(self.estimator\_weights):  
 if hasattr(estimator, 'predict\_proba'):  
 proba\_estimator = estimator.predict\_proba(X)  
 else:  
 raise AttributeError("Base estimator does not have 'predict\_proba' method.")  
  
 proba += self.estimator\_weights[i] \* proba\_estimator  
  
 # Check if the sum of estimator weights is not zero before dividing  
 if np.sum(self.estimator\_weights) != 0:  
 proba /= np.sum(self.estimator\_weights)  
  
 return proba

* These two lines create an array proba of shape (n\_samples, 2), where n\_samples is the number of samples in the input data X.
* proba will store the class probabilities for binary classification. For each sample, there are two elements in the array, representing the probabilities of the positive and negative classes, respectively.

**for i, estimator in enumerate(self.estimators):**

**if i < len(self.estimator\_weights):**

**if hasattr(estimator, 'predict\_proba'):**

**proba\_estimator = estimator.predict\_proba(X)**

**else:**

**raise AttributeError("Base estimator does not have 'predict\_proba' method.")**

**proba += self.estimator\_weights[i] \* proba\_estimator**

* This part is the main loop of the prediction process.
* It iterates through the fitted weak learners stored in self.estimators and their corresponding weights in self.estimator\_weights\_.
* For each weak learner, it calculates the class probabilities (proba\_estimator) on the input data X. If the weak learner has a predict\_proba method (indicating that it supports probabilistic predictions), it uses that method to get the probabilities.
* The class probabilities for the current weak learner are added to the proba array, weighted by the corresponding weight (self.estimator\_weights[i]).

**if np.sum(self.estimator\_weights) != 0:**

**proba /= np.sum(self.estimator\_weights)**

* After the loop, the class probabilities need to be normalized. This step ensures that the probabilities sum to 1 for each sample, as required for valid probability distributions.
* If the sum of the estimator weights is not zero (i.e., the ensemble contains at least one fitted weak learner), the proba array is divided element-wise by the sum of the estimator weights. This normalization is performed to adjust the contribution of each weak learner in the ensemble to ensure the final probabilities are correctly scaled.

**return proba**

* The method returns the final class probabilities stored in the proba array. Each row of the array corresponds to a sample in the input data, and the two columns represent the probabilities of the positive and negative classes, respectively.

**The next class**

def \_\_init\_\_(self, file\_path, features, target, test\_size=0.2):  
 self.file\_path = file\_path  
 self.features = features  
 self.target = target  
 self.test\_size = test\_size  
 self.data = None  
 self.X = None  
 self.y = None  
 self.X\_train = None  
 self.X\_test = None  
 self.y\_train = None  
 self.y\_test = None  
 self.target\_labels\_encoder = None

**self.file\_path:**

* **Purpose**: This attribute stores the file path to the CSV file from which the dataset will be loaded.
* **Usage**: It is used to read the dataset from the specified CSV file during the load\_data method.

**self.features:**

* **Purpose**: This attribute holds a list of feature column names that will be used as input features for the machine learning models.
* **Usage**: It determines which columns from the dataset will be considered as input features (X) during data manipulation and model training.

**self.target:**

* **Purpose**: This attribute stores the name of the target column (dependent variable) in the dataset.
* **Usage**: It indicates the column in the dataset that contains the target labels (y) for the classification problem.

**self.test\_size:**

* **Purpose**: This attribute determines the proportion of the dataset that will be used as the test set during the train-test split.
* **Usage**: It is used in the split\_data method when splitting the dataset into training and testing sets.

**self.data:**

* **Purpose**: This attribute holds the pandas DataFrame that represents the loaded dataset.
* **Usage**: It is used to manipulate and preprocess the data, including dropping missing values, encoding categorical variables, and scaling the features.

**self.X:**

* **Purpose**: This attribute stores the feature matrix (input data) for the machine learning models.
* **Usage**: It holds the feature data after preprocessing, which will be used for training the models.

**self.y:**

* **Purpose**: This attribute holds the target vector (output labels) for the machine learning models.
* **Usage**: It stores the target data after preprocessing, which corresponds to the labels of the feature data (self.X) used during training.

**self.X\_train:**

* **Purpose**: This attribute stores the feature matrix of the training set.
* **Usage**: It holds the feature data that will be used to train the machine learning models.

**self.X\_test:**

* **Purpose**: This attribute stores the feature matrix of the test set.
* **Usage**: It holds the feature data that will be used to evaluate the machine learning models' performance.

**self.y\_train:**

* **Purpose**: This attribute stores the target vector of the training set.
* **Usage**: It holds the target data that corresponds to the training feature data (self.X\_train) used during model training.

**self.y\_test:**

* **Purpose**: This attribute stores the target vector of the test set.
* **Usage**: It holds the target data that corresponds to the test feature data (self.X\_test) used for evaluating the model's performance.

**self.target\_labels\_encoder:**

* **Purpose**: This attribute stores the instance of the LabelEncoder used to encode the target labels.
* **Usage**: It is used to transform the target labels (self.y) into numeric form during the encode\_target method.

@staticmethod  
def evaluate\_models(y\_true, \*preds):  
 *"""  
 Evaluate the models by calculating and printing relevant metrics.  
 """* for idx, pred in enumerate(preds):  
 if isinstance(pred[0], (int, np.integer)): # Classification models  
 accuracy = accuracy\_score(y\_true, pred)  
 print(f"Model {idx + 1} Accuracy: {accuracy:.4f}")  
  
 # Additional metrics for classification models  
 if np.unique(pred).size == 2: # Check if it's a binary classification  
 pred = np.where(pred == -1, 0, 1) # Convert -1 to 0 for binary evaluation  
 cm = confusion\_matrix(y\_true, pred)  
 tn, fp, fn, tp = cm.ravel()  
 precision = tp / (tp + fp)  
 recall = tp / (tp + fn)  
 f1\_score = 2 \* (precision \* recall) / (precision + recall)  
 roc\_auc = roc\_auc\_score(y\_true, pred)  
 print(f"Model {idx + 1} Confusion Matrix:")  
 print(cm)  
 print(f"Model {idx + 1} Precision: {precision:.4f}")  
 print(f"Model {idx + 1} Recall: {recall:.4f}")  
 print(f"Model {idx + 1} F1-Score: {f1\_score:.4f}")  
 print(f"Model {idx + 1} ROC AUC: {roc\_auc:.4f}")  
 else:  
 pass

isinstance() function in used to check if an object is an instance of a specific class or a subclass of that class. It returns True | false.

In our case :

* pred[0]: This is the first element of the pred array, representing the predicted target labels for a specific model.
* (int, np.integer): This is a tuple containing two classes: the built-in

**Np.unique works like that :**   
import numpy as np

arr = np.array([2, 1, 3, 2, 1, 4, 5, 4, 5])

unique\_elements = np.unique(arr)

print(unique\_elements) # Output: [1 2 3 4 5]

and np.where (pred == -1,0,1) convert 0 to 1