Bilkent University

EEE 485: Statistical Learning and Data Analytics

Final Report

Burak Yetiştiren 21802608
Mustafa Hakan Kara 21703317

May 08, 2022

This report is submitted to the Department of Electrical and Electronics Engineering Department of Bilkent University in partial fulfillment of the requirements of the Statistical Learning and Data Analytics course EEE485.

1 Problem Description

There are different factors that can possibly affect the satisfaction of the customers with airline travel. According to Barrett, various airlines try to make a difference by maximizing customer satisfaction to attract customers [1]. Therefore, it is essential to determine some metrics that can decrease and increase the satisfaction level given various characteristics (i.e., age, gender, flight distance) of a given customer and flight. In our problem, we want to predict if a particular customer, given the characteristics of the customer and the level of satisfaction for the particular services (i.e., in-flight wifi, check-in services), is satisfied or not.

2 Dataset Description

For this project, we use the "Airlines Customer satisfaction" dataset we found on Kaggle [2]. This dataset is basically a survey directed to the passengers of a particular airline. There are 23 different categories falling under three different topics (we define topics) which are:

Passenger Information:

- Gender, Customer Type, Age, Type of Travel, Class

Flight Information:

- Flight Distance, Departure Delay in Minutes, Arrival Delay in Minutes

Passenger Rating Topics (Meta-ratings):

- In-flight Wifi Service, Departure/Arrival Time Convenience, Ease of Online Booking, Gate Location, Food and Drink, Online Boarding, Seat Comfort, In-flight Entertainment, On-board Service, Leg Room, Baggage Handling, Check-in Service, In-flight Service, Cleanliness

The features in the dataset are suitable for our problem definition; there is information on both the characteristics of the customers (passenger information, flight information) and the level of satisfaction for the particular services (passenger rating topics).
Our dataset consists of a total number of 129,880 samples. Out of the samples, 71087 are information for satisfied customers and the 59793 of them were dissatisfied customers. The figure for the distribution can be seen in Figure 1. In our implementation, we divide our dataset into three sets. The first set is used to train our ML model. This set consists of 90,916 samples, 70% of the total samples. The second set is used to validate our results. While training our neural network, we use this set after every certain number of epochs to validate our model. A detailed explanation for this validation operation is elaborated on in the 'Review of the ML Methods Used' part of this report. This set consists of 25,976 samples, 10% of the total samples. The last set is the test set, which is used to test the performance of our model. This set consists of 12,988 samples, 10% of the total samples.

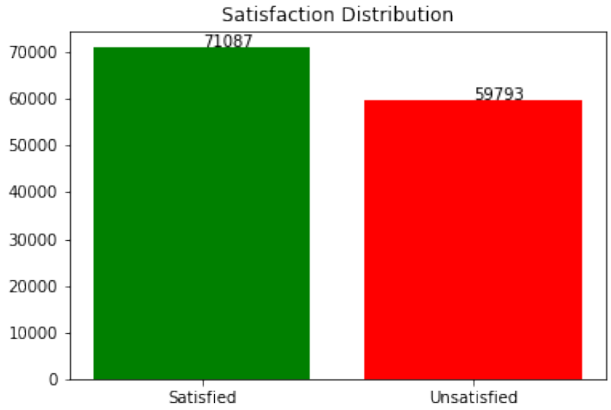

Figure 1: Satisfaction Distribution of the Dataset

	satisfaction	Gender	Customer Type	Age	Type of Travel	Class	Flight Distance	Seat comfort	Departure/Arrival time convenient	Food and drink	...	Online support	Ease of Online booking	On-board service	Leg room service	Baggage handling	Checkin service	Cleanliness	Online boarding	Departure Delay in Minutes	Arrival Delay in Minutes
0	satisfied	Female	Loyal Customer	65	Personal Travel	Eco	265	0	0	0	...	2	3	3	0	3	5	3	2	0	0.0
1	satisfied	Male	Loyal Customer	47	Personal Travel	Business	2464	0	0	0	...	2	3	4	4	4	2	3	2	310	305.0
2	satisfied	Female	Loyal Customer	15	Personal Travel	Eco	2138	0	0	0	...	2	2	3	3	4	4	4	2	0	0.0
3	satisfied	Female	Loyal Customer	60	Personal Travel	Eco	623	0	0	0	...	3	1	1	0	1	4	1	3	0	0.0
4	satisfied	Female	Loyal Customer	70	Personal Travel	Eco	354	0	0	0	...	4	2	2	0	2	4	2	5	0	0.0

Figure 2: Five Samples From our Dataset

3 Review of the ML Methods Used

1. Principal Component Analysis

   Our main purpose for using PCA in our project is to make our model more rigid. In other words, we wanted our model to make better generalizations by predicting newly encountered data more accurately.

   In Machine Learning, the curse of dimensionality is a well-known problem. It refers to the issue that the model becomes complex when the number of features is too high in a dataset. This results in an overfit to the training data, which, in turn, brings about poor predictions on the test data. By employing the PCA technique as a preprocessing step for our supervised learning tasks, we not only regularized our models but also significantly accelerated the training phase.

   The only hyperparameter related to the standard PCA is the specification of the number of components. In fact, we did not manually specify that value. Instead, we chose a threshold for the cumulative PVE and implemented an algorithm to automatically pick the minimal number of principal components satisfying this constraint.

2. Multi-layer Perceptron

   We implemented a neural network with a single hidden layer. We use hyperbolic tangent and sigmoid as an activation function of the hidden neurons and the output neurons. The output layer consists of only one neuron because our problem is a binary classification task. We used a mini-batch gradient descent algorithm since the full batch required a huge memory space (approximately 126 GB). We followed the OOP paradigm and implemented the network as a class. Our choice for the number of neurons in the hidden layer was arbitrary, and different choices did not considerably affect the result.

3. Random Forests

   Decision trees are used to solve the problem of binary classification. Since this coincides with our problem definition, we also chose to utilize decision trees in our implementation. To do this, we implemented a random forest that contains 10 decision trees. We have seen that the difference between utilizing 100 trees instead of 10 trees increases the performance of our model not so much, hence for performance we chose to implement 10 trees. The performance of our model is examined in Section 4.

4. Naive Bayes

   The fourth method we implemented was Naive Bayes. This method was implemented mostly to justify why the other methods should be chosen for our problem. This is because, as Naive Bayes assumes that the features given are independent. This means that if the features are not independent, then the probabilities that the method estimates are incorrect. We previously suggested that the features in our dataset are dependent on each other (i.e., customer age, gender could depend to customer ratings to In-flight Wifi Service, Departure/Arrival Time Convenience, etc.). As it will be argued in Section 4, the performance of this estimator is relatively lower than the others, hence we can say that our features are indeed not independent.

4 Performance Results

We employ some metrics to measure the performance of our methods. The usage of various metrics is important, as the results of a single metric, such as accuracy, can tell incomprehensive information about the performance of our models.
The metrics we test include:

- Accuracy
- False Positive Rate (FPR) (Type-I error)
- False Negative Rate (FNR) (Type-II error)
- Precision
- Recall (True Positive Rate, Sensitivity)
- F1
- ROC Curve

Accuracy

We test the accuracy of our models by employing our test set after the training of our model. While some of the results seemingly proves themselves to be sufficient models, testing accuracy alone is not a conclusive metric to determine the performance of our model, hence we also evaluated the following metrics.

1. Multi-layer Perceptron

   We trained our neural network for 10,000 epochs, after this training, we forwarded the test portion of our design matrix to the trained network and compared the labels for these samples. By doing this, we obtained 92.6% test accuracy for our model.

2. Random Forests

   We trained our random forest containing 10 trees with our data. The resulting test accuracy of our model was 95.1%.

3. Naive Bayes

   As argued previously in Section 3, our Naive Bayes estimator was the least successful method that we implemented in terms of accuracy. The test accuracy we obtained bu utilizing Naive Bayes was 81.7%.

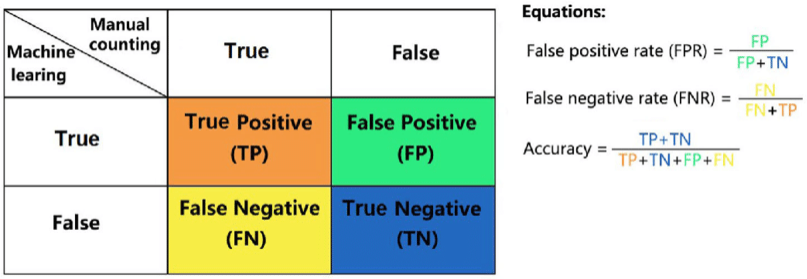

Figure 3: Confusion matrix and Performance Equations

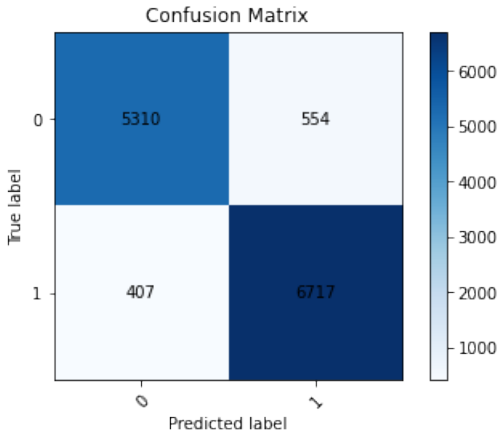

Figure 4: Our Confusion Matrix for Multi-layer Perceptron

Note: Notice that our labels are switched in comparison to the figure above

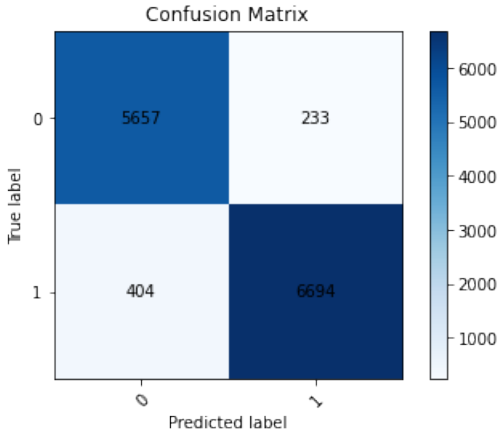

Figure 5: Our Confusion Matrix for Random Forest

Note: Notice that our labels are switched in comparison to the figure above

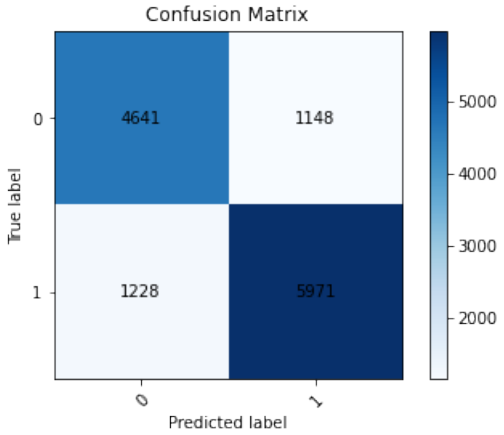

Figure 6: Our Confusion Matrix for Naive Bayes

Note: Notice that our labels are switched in comparison to the figure above

False Positive Rate (FPR) (Type-I error)

False Positive Rate (FPR) measures the ratio between the false positives and the sum of false positives and true negatives. In our models, the positives are labeled as 'satisfied', and the negatives are labeled as 'dissatisfied'. The false positive rate determines the rate of positive (satisfied) labeled samples, where their true labels are 'dissatisfied'.

1. Multi-layer Perceptron

   After 10,000 epochs, we had a total number of 554 false positive samples and 5310 true negative samples. This yielded an FPR of 9.4%.

2. Random Forests

   After we estimated our test data with our random forest, we obtained 233 false positive, and 5657 true negative samples. This yielded an an FPR of 4%.

3. Naive Bayes

   After we estimated our test data with our Naive Bayes estimator, we obtained 1148 false positive, and 4641 true negative samples. This yielded an an FPR of 19.8%.

False Negative Rate (FNR) (Type-II error)

Similar to the FPR, FNR is also a metric to determine the rate of falsely classified samples. This time, we calculate the rate of negative(dissatisfied) labeled samples, where their true labels are 'satisfied'.

1. Multi-layer Perceptron

   Again, after 10,000 epochs, we had 407 false negative samples and 6717 true positive samples. This yielded an FNR of 5.7%.

2. Random Forests

   After we estimated our test data with our random forest, we obtained 404 false negative, and 6694 true positive samples. This yielded an an FNR of 5.7%.

3. Naive Bayes

   After we estimated our test data with our Naive Bayes estimator, we obtained 1228 false negative, and 5971 true positive samples. This yielded an an FNR of 17.1%.

Precision

Precision is a metric which is a ratio of the true positive and false positive samples. The metric is calculated as:

$$Precision = \frac{TP}{TP + FP}$$

In a good model, this metric is ideally close to 1.

1. Multi-layer Perceptron

   We had 6717 true positive and 554 false positive samples in our neural network. This yielded a ratio of 92.4%.

2. Random Forests

   We had 6694 true positive and 233 false positive samples in our estimator. This yielded a ratio of 96.6%.

3. Naive Bayes

   We had 5971 true positive and 1148 false positive samples in our estimator. This yielded a ratio of 83.9%.

Recall

Recall is another metric that should be ideally close to 1. This metric is also called True Positive Rate (TPR) and Sensitivity. This metric is a ratio of the true positive and false negative samples. In the ideal case, the number of true positive samples is equal to the sum of true positive and false negative samples. The metric is calculated as:

$$Recall = \frac{TP}{TP + FN}$$

1. Multi-layer Perceptron

   In our model, we had 6717 true positive and 407 false negative samples. This yielded a ratio of 94.3%.

2. Random Forests

   We had 6694 true positive and 404 false negative samples in our estimator. This yielded a ratio of 94.3%.

3. Naive Bayes

   We had 5971 true positive and 1228 false negative samples in our estimator. This yielded a ratio of 82.9%.

F1

F1 score could be thought of as a combination of precision and recall scores. This follows that the F1 score is 1 only if both precision and recall scores are 1, similarly, the F1 score is high only when precision and recall scores are high. This score is computed by this formula:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

1. Multi-layer Perceptron

   We previously found that our precision was 92.4% and recall was 94.3%. This follows that our F1 score is 93.3%.

2. Random Forests

   We previously found that our precision was 96.6% and recall was 94.3%. This follows that our F1 score is 95.4%.

3. Naive Bayes

   We previously found that our precision was 81.7% and recall was 82.9%. This follows that our F1 score is 83.4%.

ROC Curve

ROC Curve, more specifically the Receiver Operating Characteristic Curve is used to measure the performance of a model predicting binary labels. The method uses the probabilities of the given samples belonging to one of the classes. Different from the previous metrics, this metric is more visual, and it will be especially useful in contrasting different ML methods given the same problem. The area under this curve (AOC) gives a summary of the model, that the bigger the area the better the model, hence it is preferred that the curvature is close to the (x, y) = (0, 1) point where x = TPR, and y = FPR. If this curvature touches this point, then the area would be equal to 1, which would indicate a perfect model without errors.

1. Multi-layer Perceptron

   The ROC Curve for our model could be observed in the following. Our area is equal to 0.98, which indicates that our model is suitable to our data.

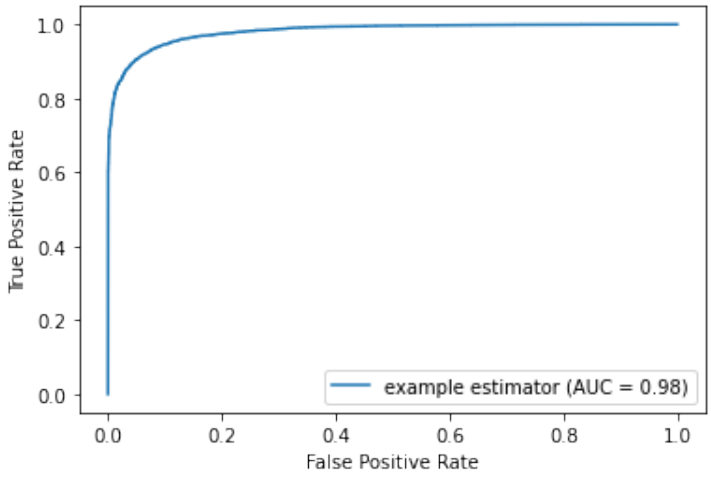

Figure 7: Our ROC Curve for our Neural Network

2. Random Forests

   The ROC Curve for our model could be observed in the following. Our area is equal to 0.99, which indicates that our model is suitable to our data.

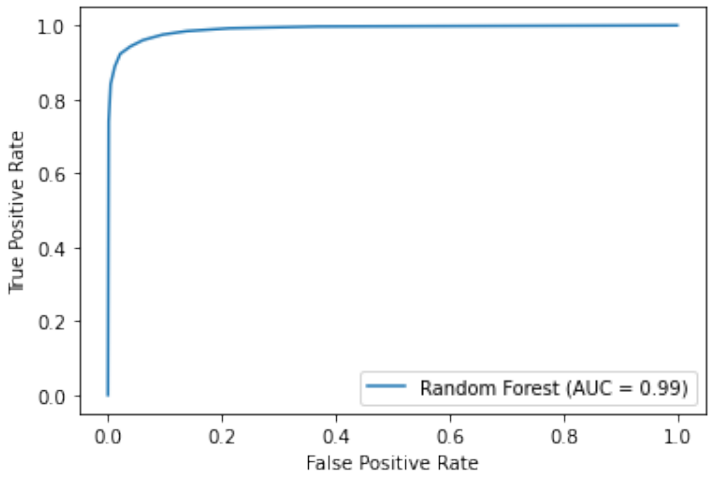

Figure 8: Our ROC Curve for our Random Forest

3. Naive Bayes

The ROC Curve for our model could be observed in the following. Our area is equal to 0.99, which indicates that our model is suitable, but less suitable than the classifiers stated above.

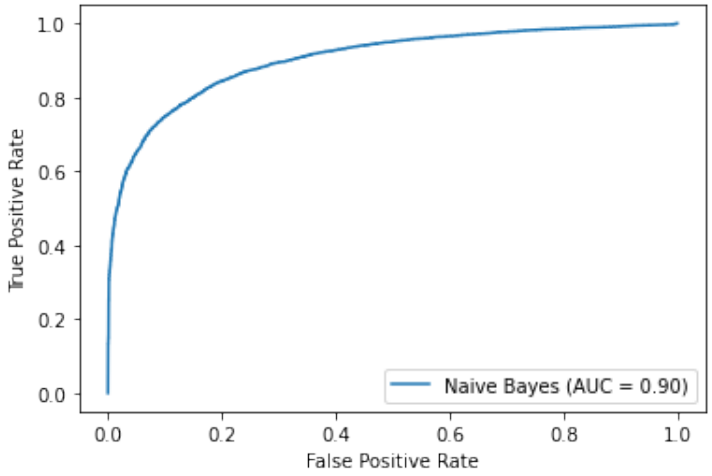

Figure 9: Our ROC Curve for our Naive Bayes model

5 Conclusion

In conclusion, we want to create a solution to the problem of predicting if a given customer is satisfied or not given the information about that customer and the flight s/he has been on, and the meta-ratings they gave for some particular aspects of their travel. Until now, we created an approach that combines the PCA and neural network methods. PCA is used as a preprocessing step to reduce the number of features we have in our dataset, which helps to decrease the computational cost we have whilst training our neural network. We calculated the performance of our results with different metrics to prove the performance of our results. So far, for each metric we calculated (the detailed explanation is given in the Performance Results section), we obtained good results, verifying the performance of our model. In the remaining part of the project, we want to implement different ML models and calculate their performance using the same metrics we explained. Then we want to contrast them with our current model to see if they could be a better choice.

References

[1] S. Jana, "Airlines customer satisfaction," Mar 2020. [Online]. Available: https://www.kaggle.com/datasets/sjleshrac/airlines-customer-satisfaction

[2] T. Barnett, "What factors affect airline customer satisfaction?" Mar 2022. [Online]. Available: https://www.wise-geek.com/what-factors-affect-airline-customer-satisfaction.htm

Appendices

```
import pandas as pd
from math import isqrt
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import *
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA

def split_data(data):
    X = data[:, 1:]
    y = data[:, 0]
    return X, y

def ordinal_encode(X, y):
    oe = OrdinalEncoder()
    le = LabelEncoder()
    oe.fit(X)
    le.fit(y)
    X = oe.transform(X)
    y = le.transform(y)
    return X, y

def compute_eigens(X):
    covMatrix = np.matmul(X.T, X)
    eigenValues, eigenVectors = np.linalg.eigh(covMatrix)
    return eigenValues, eigenVectors

def principal_components(X, min_pve=0.9):
    eigenValues, eigenVectors = compute_eigens(X)
    indices = np.argsort(eigenValues)[::-1]
    eigenValues = np.sort(eigenValues)[::-1]
    k = 1
    pve = np.sum(eigenValues[:k]) / np.sum(eigenValues)
    while(pve < min_pve):
        k += 1
        pve = np.sum(eigenValues[:k]) / np.sum(eigenValues)
    print(k, pve)
    firstK = np.array([eigenVectors[:, indices[i]] for i in range(k)])
    X_new = np.matmul(X, firstK.T)
    return X_new

data = pd.read_csv('./datasets/Invistico_Airline.csv')
data.fillna(0, inplace=True)
data = data.to_numpy()
```

```
X,y = split_data(data)
X,y = ordinal_encode(X, y)
print(y)
X = StandardScaler().fit_transform(X)
X = principal_components(X, min_pve=0.89)

# 70% train, 20% validation, 10% test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, train_size=0.9)

X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train, train_size=7/9)

from sklearn.metrics import *
class NeuralNetwork:
    def __init__(self):
        self.gradients = dict()

    def __initialize_parameters(self, n_x, n_h, n_y):
        np.random.seed(2)

        W1 = np.random.randn(n_h, n_x) * 0.01
        b1 = np.zeros((n_h, 1))
        W2 = np.random.randn(n_y, n_h) * 0.01
        b2 = np.zeros((n_y,1))

        self.weights = {"W1": W1,
                        "b1": b1,
                        "W2": W2,
                    "b2": b2}

    def __sigmoid(self, X):
        return 1 / (1 + np.exp(-X))

    def fit(self, X, y):
        self.X = X
        self.y = y
        self.num_samples = self.y.shape[0]
        n_x = X.shape[1]
        n_h = 22
        n_y = 1
        self.__initialize_parameters(n_x, n_h, n_y)

    def __forward(self, X):
```

```
        W1 = self.weights["W1"]
        b1 = self.weights["b1"]
        W2 = self.weights["W2"]
        b2 = self.weights["b2"]

        Z1 = np.dot(W1, X.T) + b1
        A1 = np.tanh(Z1)
        Z2 = np.dot(W2, A1) + b2
        A2 = self.__sigmoid(Z2)

        self.cache = {"Z1": Z1,
                      "A1": A1,
                      "Z2": Z2,
                      "A2": A2}

        return A2

    def __get_loss(self, y_hat, y):
        m = y.shape[0]
        log_probs = np.multiply(np.log(y_hat),y) +
np.multiply(np.log(1-y_hat), (1-y))
        loss = (-1./m) * np.sum(log_probs)

        loss = float(np.squeeze(loss))

        return loss

    def __get_grads(self, X, y):
        m = X.shape[0]

        W2 = self.weights["W2"]
        A1 = self.cache["A1"]
        A2 = self.cache["A2"]

        dZ2 = A2-y
        dW2 = (1./m)*np.dot(dZ2, A1.T)
        db2 = (1./m)*np.sum(dZ2, axis = 1, keepdims=True)
        dZ1 = np.dot(W2.T, dZ2) * (1 - np.power(A1, 2))
        dW1 = (1./m)*np.dot(dZ1, X)
        db1 = (1./m)*np.sum(dZ1, axis = 1, keepdims=True)

        grads = {"dW1": dW1,
                 "db1": db1,
                 "dW2": dW2,
                 "db2": db2}
```

```
        return grads

    def __update_parameters(self):
        W1 = self.weights["W1"]
        W2 = self.weights["W2"]
        b1 = self.weights["b1"]
        b2 = self.weights["b2"]

        dW1 = self.gradients["dW1"]
        db1 = self.gradients["db1"]
        dW2 = self.gradients["dW2"]
        db2 = self.gradients["db2"]

        W1 -= self.learning_rate * dW1
        b1 -= self.learning_rate * db1
        W2 -= self.learning_rate * dW2
        b2 -= self.learning_rate * db2

        self.weights = {"W1": W1,
                        "b1": b1,
                        "W2": W2,
                        "b2": b2}

    def train(self, batch_size=5000, num_iterations=1000,
learning_rate=0.1):
        self.learning_rate = 0.1
        num_batches = self.num_samples // batch_size

        for i in range(num_iterations):
            start = 0
            loss = 0

            self.gradients["dW1"] = 0
            self.gradients["db1"] = 0
            self.gradients["dW2"] = 0
            self.gradients["db2"] = 0
            for j in range(num_batches):
                end = start + batch_size

                X_miniBatch = self.X[start:end, :]
                y_miniBatch = self.y[start:end]

                A2 = self.__forward(X_miniBatch)
                loss += self.__get_loss(A2, y_miniBatch)
                mini_grads = self.__get_grads(X_miniBatch, y_miniBatch)
```

```
                self.gradients["dW1"] += mini_grads["dW1"]
                self.gradients["db1"] += mini_grads["db1"]
                self.gradients["dW2"] += mini_grads["dW2"]
                self.gradients["db2"] += mini_grads["db2"]
                start = end
            self.__update_parameters()
            loss /= batch_size
            if i % (num_iterations // 50) == 0:
                self.print_accuracy(X_val[5000:10000, :],
y_val[5000:10000])
                print ("Cost after iteration %i: %f" %(i, loss))

    def __predict(self, X_test):
        A2 = self.__forward(X_test)
        predictions = A2 > 0.5
        return predictions

    def print_accuracy(self, X_test, y_test):
        y_pred = self.__predict(X_test)
        print ('Accuracy: %d' % float((np.dot(y_test,y_pred.T) +
np.dot(1-y_test,1-y_pred.T))/float(y_test.size)*100) + '%')

    def print_confusion(self, X_test, y_test):
        y_predicted = self.__predict(X_test)[0, :]
        conf_matrix = confusion_matrix(y_test, y_predicted)
        acc = accuracy_score(y_test, y_predicted)
        pre = precision_score(y_test, y_predicted)
        recall = recall_score(y_test, y_predicted)
        f1 = f1_score(y_test, y_predicted)
        # print metrics
        print("\nMean Acc:", acc,"\nMean Macro Precision:",
              pre, "\nMean Macro Recall:", recall, "\nMean Macro F1
Score:", f1)
        # plot confusion matrix
        fig, ax = plt.subplots()

        for i in range(conf_matrix.shape[0]):
            for j in range(conf_matrix.shape[1]):
                ax.text(j, i, conf_matrix[i, j], ha="center",
va="center")

        im = ax.imshow(conf_matrix, interpolation='nearest',
cmap=plt.cm.Blues)
        ax.figure.colorbar(im, ax=ax)
        #show the text in the plot
        ax.set(xticks=np.arange(conf_matrix.shape[1]),
```

```
                yticks=np.arange(conf_matrix.shape[0]),
                title="Confusion Matrix",
                ylabel="True label",
                xlabel="Predicted label")
        plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
                rotation_mode="anchor")
        # show the plot
        plt.show()


        #im = ax.imshow(conf_matrix)
        #We want to show all ticks...
        #ax.set_xticks(np.arange(2))
        #ax.set_yticks(np.arange(2))
        #fig.tight_layout()
        #plt.show()

    def print_roc(self, X_test, y_test):
        pred = self.__forward(X_test)[0, :]
        fpr, tpr, thresholds = roc_curve(y_test, pred)
        roc_auc = auc(fpr, tpr)
        display = RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc,
                                  estimator_name='example
estimator')
        display.plot()
        plt.show()

    def calculate_rates(self, X_test, y_test):
        pred = self.__predict(X_test)[0, :]
        false_positives = np.sum(pred * (1 - y_test))
        false_negatives = np.sum((1 - pred) * y_test)
        true_positives = np.sum(pred * y_test)
        true_negatives = np.sum((1 - pred) * (1 - y_test))

        fpr = false_positives / (false_positives + true_negatives)
        fnr = false_negatives / (false_negatives + true_positives)

        print("False Positives:", false_positives)
        print("False Negatives:", false_negatives)
        print("True Positives:", true_positives)
        print("True Negatives:", true_negatives)
        print('')
        print("False Positive Rate:", fpr)
        print("False Negative Rate:", fnr)

NN = NeuralNetwork()
```

```
NN.fit(X_train, y_train)
NN.train(batch_size=5000,num_iterations=10000)

NN.print_roc(X_test, y_test)

NN.print_confusion(X_test, y_test)

NN.calculate_rates(X_test, y_test)
```

```
class NaiveBayes:
    def __init__(self, X, y):
        self.X = X
        self.y = y
        self.num_samples = X.shape[0]
        self.num_features = X.shape[1]
        self.num_classes = np.unique(y).shape[0]
        self.priors = np.zeros(self.num_classes)
        self.conditional_probabilities = np.zeros((self.num_classes,
self.num_features))
        self.__fit()

    def __fit(self):
        for i in range(self.num_classes):
            self.priors[i] = np.sum(self.y == i) / self.num_samples
            for j in range(self.num_features):
                self.conditional_probabilities[i, j] =
np.sum(self.X[self.y == i, j]) / np.sum(self.y == i)

    def predict(self, X_test):
        predictions = np.zeros(X_test.shape[0])
        for i in range(X_test.shape[0]):
            probabilities = np.zeros(self.num_classes)
            for j in range(self.num_classes):
                probabilities[j] = self.priors[j]
                for k in range(self.num_features):
                    probabilities[j] *=
self.conditional_probabilities[j, k] ** X_test[i, k] * (1 -
self.conditional_probabilities[j, k]) ** (1 - X_test[i, k])
            predictions[i] = np.argmax(probabilities)
        return predictions

    def print_accuracy(self, X_test, y_test):
        y_pred = self.predict(X_test)
        print ('Accuracy: %d' % float((np.dot(y_test,y_pred.T) +
np.dot(1-y_test,1-y_pred.T))/float(y_test.size)*100) + '%')

    def print_confusion(self, X_test, y_test):
        y_predicted = self.predict(X_test)[0, :]
        conf_matrix = confusion_matrix(y_test, y_predicted)
        acc = accuracy_score(y_test, y_predicted)
        pre = precision_score(y_test, y_predicted)
        recall = recall_score(y_test, y_predicted)
        f1 = f1_score(y_test, y_predicted)
        # print metrics
```

```
        print("\nMean Acc:", acc,"\nMean Macro Precision:",
                pre, "\nMean Macro Recall:", recall, "\nMean Macro F1
Score:", f1)
        # plot confusion matrix
        fig, ax = plt.subplots()

        for i in range(conf_matrix.shape[0]):
            for j in range(conf_matrix.shape[1]):
                ax.text(j, i, conf_matrix[i, j], ha="center",
va="center")

        im = ax.imshow(conf_matrix, interpolation='nearest',
cmap=plt.cm.Blues)
        ax.figure.colorbar(im, ax=ax)
        #show the text in the plot
        ax.set(xticks=np.arange(conf_matrix.shape[1]),
                yticks=np.arange(conf_matrix.shape[0]),
                title="Confusion Matrix",
                ylabel="True label",
                xlabel="Predicted label")
        plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
                rotation_mode="anchor")
        # show the plot
        plt.show()
```

```
class RandomForest:
    def __init__(self, X, y, num_trees):
        self.X = X
        self.y = y
        self.num_trees = num_trees
        self.num_samples = X.shape[0]
        self.num_features = X.shape[1]
        self.num_classes = np.unique(y).shape[0]
        self.trees = []
        self.__fit()

    def __fit(self):
        for i in range(self.num_trees):
            indices = np.random.choice(self.num_samples, self.num_samples,
replace=True)
            X_train = self.X[indices, :]
            y_train = self.y[indices]
            tree = DecisionTree(X_train, y_train)
            self.trees.append(tree)
```

```
    def predict(self, X_test):
        predictions = np.zeros(X_test.shape[0])
        for i in range(X_test.shape[0]):
            probabilities = np.zeros(self.num_classes)
            for j in range(self.num_trees):
                probabilities[self.trees[j].predict(X_test[i, :])] += 1
            predictions[i] = np.argmax(probabilities)
        return predictions

    def print_accuracy(self, X_test, y_test):
        y_pred = self.predict(X_test)
        print ('Accuracy: %d' % float((np.dot(y_test,y_pred.T) +
np.dot(1-y_test,1-y_pred.T))/float(y_test.size)*100) + '%')

    def print_confusion(self, X_test, y_test):
        y_predicted = self.predict(X_test)[0, :]
        conf_matrix = confusion_matrix(y_test, y_predicted)
        acc = accuracy_score(y_test, y_predicted)
        pre = precision_score(y_test, y_predicted)
        recall = recall_score(y_test, y_predicted)
        f1 = f1_score(y_test, y_predicted)
        # print metrics
        print("\nMean Acc:", acc,"\nMean Macro Precision:",
                pre, "\nMean Macro Recall:", recall, "\nMean Macro F1
Score:", f1)
        # plot confusion matrix
        fig, ax = plt.subplots()

        for i in range(conf_matrix.shape[0]):
            for j in range(conf_matrix.shape[1]):
                ax.text(j, i, conf_matrix[i, j], ha="center", va="center")

        im = ax.imshow(conf_matrix, interpolation='nearest',
cmap=plt.cm.Blues)
        ax.figure.colorbar(im, ax=ax)
        #show the text in the plot
        ax.set(xticks=np.arange(conf_matrix.shape[1]),
                yticks=np.arange(conf_matrix.shape[0]),
                title="Confusion Matrix",
                ylabel="True label",
                xlabel="Predicted label")
        plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
                rotation_mode="anchor")
        # show the plot
        plt.show()
```

```
class DecisionTree:
    def __init__(self, X, y):
        self.X = X
        self.y = y
        self.num_samples = X.shape[0]
        self.num_features = X.shape[1]
        self.num_classes = np.unique(y).shape[0]
        self.__fit()

    def __fit(self):
        self.tree = self.__build_tree(self.X, self.y)

    def __build_tree(self, X, y):
        if np.unique(y).shape[0] == 1:
            return y[0]
        if X.shape[0] == 0:
            return np.argmax(np.bincount(y))
        if X.shape[1] == 0:
            return np.argmax(np.bincount(y))
        best_feature = self.__get_best_feature(X, y)
        best_feature_index = best_feature[0]
        best_feature_value = best_feature[1]
        X_left = X[X[:, best_feature_index] <= best_feature_value, :]
        y_left = y[X[:, best_feature_index] <= best_feature_value]
        X_right = X[X[:, best_feature_index] > best_feature_value, :]
        y_right = y[X[:, best_feature_index] > best_feature_value]
        tree = {'feature': best_feature_index, 'value':
best_feature_value,
                'left': self.__build_tree(X_left, y_left),
                'right': self.__build_tree(X_right, y_right)}
        return tree

    def __get_best_feature(self, X, y):
        best_feature_index = 0
        best_feature_value = 0
        best_feature_gain = 0
        for i in range(X.shape[1]):
            values = np.unique(X[:, i])
            for j in range(values.shape[0]):
                X_left = X[X[:, i] <= values[j], :]
                y_left = y[X[:, i] <= values[j]]
                X_right = X[X[:, i] > values[j], :]
                y_right = y[X[:, i] > values[j]]
                gain = self.__get_gain(X, y, i, values[j])
                if gain > best_feature_gain:
                    best_feature_index = i
```

```
                    best_feature_value = values[j]
                    best_feature_gain = gain
        return best_feature_index, best_feature_value

    def __get_gain(self, X, y, feature_index, feature_value):
        X_left = X[X[:, feature_index] <= feature_value, :]
        y_left = y[X[:, feature_index] <= feature_value]
        X_right = X[X[:, feature_index] > feature_value, :]
        y_right = y[X[:, feature_index] > feature_value]
        gain = self.__get_entropy(y) - \
                (X_left.shape[0]/X.shape[0]) * self.__get_entropy(y_left)
- \
                (X_right.shape[0]/X.shape[0]) *
self.__get_entropy(y_right)
        return gain

    def __get_entropy(self, y):
        unique_y = np.unique(y)
        entropy = 0
        for i in range(unique_y.shape[0]):
            p = np.sum(y == unique_y[i])/y.shape[0]
            entropy += -p * np.log2(p)
        return entropy

    def predict(self, X):
        predictions = np.zeros(X.shape[0])
        for i in range(X.shape[0]):
            predictions[i] = self.__predict(X[i, :])
        return predictions

    def __predict(self, X):
        node = self.tree
        while True:
            if X[node['feature']] <= node['value']:
                node = node['left']
            else:
                node = node['right']
            if type(node) == int:
                return node

    def print_tree(self):
        self.__print_tree(self.tree)

    def __print_tree(self, node, depth=0):
        if type(node) == int:
            print(" "*depth, "Leaf:", node)
```

```
            return
        print(" "*depth, "Feature:", node['feature'], "Value:",
node['value'])
        self.__print_tree(node['left'], depth+1)
        self.__print_tree(node['right'], depth+1)

    def __get_accuracy(self, y_predicted, y_test):
        correct = 0
        for i in range(y_predicted.shape[0]):
            if y_predicted[i] == y_test[i]:
                correct += 1
        return correct/y_predicted.shape[0]

    def __get_confusion_matrix(self, y_predicted, y_test):
        confusion_matrix = np.zeros((self.num_classes, self.num_classes))
        for i in range(y_predicted.shape[0]):
            confusion_matrix[y_predicted[i], y_test[i]] += 1
        return confusion_matrix
```

```
nb = NaiveBayes(X_train, y_train)
nb.print_accuracy(X_test, y_test)
nb.print_confusion(X_test, y_test)
nb.print_roc(X_test, y_test)
```

```
rf = RandomForest(X_train, y_train, 10)
rf.print_accuracy(X_test, y_test)
rf.print_confusion(X_test, y_test)
rf.print_roc(X_test, y_test)
```