Mustafa Cankan Balcı
22101761
25 December 2024

EEE485 Statistical Learning and Data Analytics Term Project
Airline Passenger Satisfaction

Introduction

In today's globalized world, transportation is essential, and air travel stands out for its speed and convenience when compared to land or sea travel. The quality of services provided by airlines has emerged as a critical component in fostering consumer loyalty and industry expansion as the demand for air travel rises globally. Airlines rely significantly on passenger input gathered through surveys and consumer reviews to enhance their offerings. Airlines can modify their services to better suit customer needs by using this data-driven strategy to better understand changing passenger preferences and expectations. The goal of EEE 485/585 term project is to use a dataset created to record different of the flight experience in order to analyze consumer satisfaction. The project's goal in examining this dataset is to offer insights on customers. By exploring this dataset, the project seeks to provide insights that can contribute to enhancing service quality and overall customer happiness in the air transportation sector.

Problem Description

The term project aims to analyze the founded data on different machine learning algorithms. All machine learning algorithms gives result satisfied or not satisfied, which is binary classification. Machine learning algorithms planned to use in the term project are Logistic Regression, Neural Networks, Random Forest and Support Vector Machine. Logistic Regression is selected since it's goal is to predict two outcomes 0 and 1. In the training of each machine learning. model, K-fold cross-validation is used for obtaining more precise result in the training. The scores of each training set in evaluated in the validation parameters. The performance of each machine learning model is shown in the confusion matrix, which shows the result of machine learning run in test data. All chosen machine learning algorithms designed on Python. Custom build machine learning algorithm did not use in the project depending on the project description.

Revision from the First Report

After first demo, I changed the parts of machine learning models according to advice of course TA. First of all, model can be stuck in local minima in the optimization because of full batch gradient descent. I changed the optimization structure to mini batch stochastic gradient descent. Also, I implemented Random Forest and Support Vector machine algorithms for the final report. I added the desired classification output in the confusion matrix of the data set. I also added tables for increasing the investigation in the network.

Dataset

Dataset of the term project is "Airline Satisfaction Survey" on Kaagle [1]. The dataset was constructed from airline passenger satisfaction survey. The usability score of this project is 9.41 from 10. The dataset consists of 2 files such as test and training. The project train and test dataset contain information about gender, customer type, age, type of travel, class, inflight WIFI service, departure/arrival time convenient, ease of online booking, gate location, food and drink, online boarding, seat comfort, inflight entertainment, on-board service, leg room service, baggage handling, check-in service, inflight service, cleanliness, departure delay in minutes and arrival delay in minutes. These information in the dataset is used as a feature in machine learning algorithms. Both datasets set contains id and index number, which does not have meaningful to use in the satisfaction classification. As a result, these columns is deleted in both data set. The train dataset contains 103904 rows and 23 columns.

Features in the dataset are represented categorically, scores and integer values. Categorically classified features are represented by 0 and 1, so that machine learning algorithms understand these features. For instance, gender feature is classified as Male and Female, so they are converted as "Male" is represented "0" and "Female" is represented "1". In the customer type, hierarchal representation is used. Business is considered as higher priority for airline ,so it assigned highest value among other classes.

Analysis of Machine Learning Algorithms

In the project, there are 4 different machine learning algorithms is going to implement the binary classification. The algorithms are Logistic Regression, Neural Networks, Random Forest, Support Vector Machine.

Logistic Regression

Logistic regression is one of the machine learning algorithms used in the project for binary classification [2]. It enables to map input vales to probabilities between 0 and 1. The model establishes a linear relationship between input features and the log-odds of the target, creating a simple decision boundary. The advantage of logistic regression is easy to implement with high performance even if less data samples. The disadvantage of logistic regression is that it does not perform well on not linearly separable.

$$\sigma(x) = \frac{1}{1+e^{-XW}}$$
$$W = weight\ list$$
$$X = input\ data\ set$$

The loss function of the logistic regression is negative-log loss. The first gradient of logistic regression vanishes in the small parameter, and the second derivative of the logistic regression is non-convex. As a result, the negative-logloss is used for the loss calculation. It is maximum like hood of the posterior distribution of logistic regression output and weights with using scores.

$$y = \frac{1}{m}\sum_{i=1}^{m} y_i\big(\log(\sigma(x))\big) + (1-y_i)\big(\log(1-\sigma(x))\big)$$

In the optimization of logistic regression, full-batch gradient descent is used. (Describe here full batch gradient descent). In the full batch gradient descent, the gradient of parameters is represented in the below.

$$dW = \frac{1}{m}X^T\big(y-\sigma(x)\big)$$
$$db = \big(y-\sigma(x)\big)\frac{1}{m}$$

In the optimization part, mini batch stochastic gradient descent is used.

$$W = W - \propto dW$$
$$b = b - \propto db$$
$$\propto = learning\ parameter$$

Neural Networks

Neural networks make the programs to recognize patterns and solve common problems in machine learning [3]. It is able to categorize data into one of two classes even if the data set non-linear separable. The type of neural network designed in the project is called Shallow Neural Network [4]. Shallow Neural Networks consists of input layer, one hidden layer and output layer. Figure 1 represents the schematics of shallow neural network.

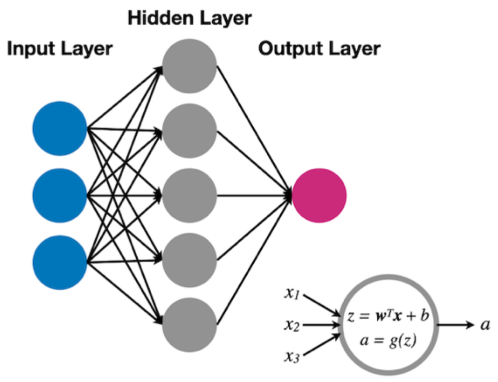

Figure 1: Shallow Neural Network Schematic

Neurons consists of 2 stage, which linear layer and activation layer. The linear layer calculates the linear relation with input and weights of layer. In the linear layer, bias term is added for making more sustainable model. The output of linear layer is passed through activation layer. There are several activations functions such as sigmoid, tanh(x) and ReLu can applied in layers of neural networks. Ta In the project, tanh is used in an activation function of hidden layer, and sigmoid is used as an activation function of output layer.

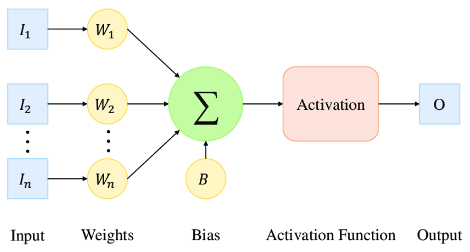

Figure 2: Neuron Schematic

All weights in the neural network layers is updated by full-batch gradient descent in the first report. The gradient of each parameter is updated by the backpropagation. The calculations of each gradient in neural network layer are shown below.

$$dZ2 = A2 - Y$$
$$dW2 = \frac{1}{m} dZ2$$
$$db2 = \frac{1}{m} dZ2$$
$$dZ1 = \frac{1}{m} dZ2$$
$$dW1 = \frac{1}{m} dZ2$$

$$db1 = \frac{1}{m} dZ$$

After obtaining the gradients of the parameters, the values are updated at each epoch. Also, mini batch stochastic gradient descent is used to updating parameters in neural network model. Begin of each epoch train data set is shuffled.

$$W = W - \propto dW$$
$$b = b - \propto db$$
$$\propto = learning\ parameter$$

Random Forest

Random Forest is an ensemble learning algorithm used for regression and classification operations [5]. It operates by constructing multiple decision trees during training. Each tree in the forest is trained on a randomly selected subset of data. This process is called bagging. Also, each split in the tree, only random subset of features is considered. As a result, it introduces diversity among the trees and reduces the risk of overfitting. The algorithm uses majority voting among the trees to determine the binary classification.

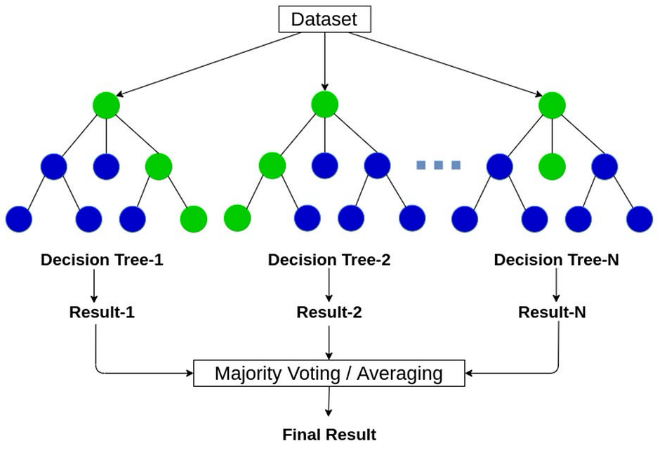

Figure 3: Random Forest Schematic

In the training process of random forest, the machine learning model minimizes "impurity" at each decision tree split, rather than a single global loss function. Gini impurity is used impurity measurement for guiding how splits are made at each node of the decision trees. Gini impurity measures the probability of incorrect classification if a random sample was labeled according to the class distribution in a node. p_k is the proportion of class belonging to class k in the node, and K is the total number of classes. Lower Gini impurity shows a more homogeneous node, which means the samples within the node predominantly belong to one class [6]. Durning training, random forest selects the feature and threshold that result in the greatest reduction in Gini impurity at each split. Also, entropy is used to measure the randomness in the class distribution. A perfectly homogeneous node has a 0 entropy.

$$Gini\ Impurirty = 1 - \sum_{k=1}^{K} p_k^2$$

Support Vector Machine

Support Vector Machine (SVM) in supervised machine learning algorithm primarily used for classification and regression tasks. In classification tsks, SVM aim to find the optimal hyperplane that best separates data points of different classes. The hyperplane with maximum margin, which is the distance between the hyperplane and the nearest data points from each class.

Support Vectors are the data points closet to the hyperplane and directly influence its position and orientation. The margin is the gap between 2 classes separated by the hyperplane. A larger margin implies a better generalizing classifier, reducing the risk of misclassification. There are 2 types of margins: soft and hard margin. Soft margin allows some misclassifications to achieve better generalizations, when data is noisy or not perfectly separable. In this project hard margin is used in SVM classification.

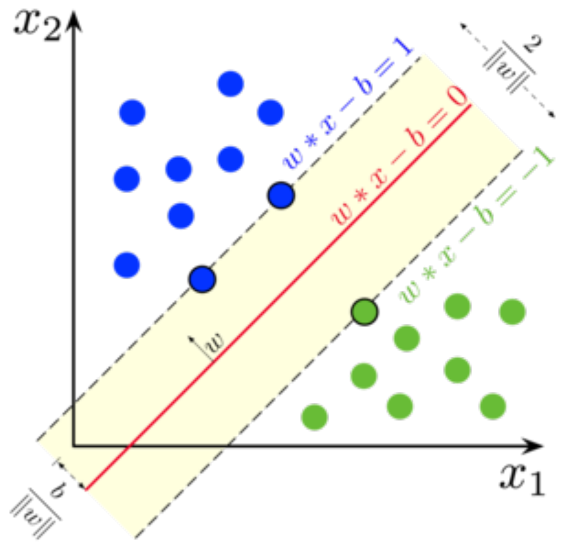

Figure 4: Support Vector Machine (SVM)

The equation of SVM classification is shown below. The classification is done by -1 and +1 parameters. However, satisfaction column is converted to 0 and 1. For solving this issue, satisfaction is converted to -1 and +1 for SVM training and remapped to 0 and 1 in the prediction.

$$\hat{y} = \begin{cases} -1, & w^T \cdot x + b < 0 \\ 1, & w^T \cdot x + b \geq 0 \end{cases}$$

In the optimization, stohasctic gradient descent algorithm is used. The loss function of SVM consists of 2 equations, which are hinge loss and L2 regularization. Hinge loss is used to maximize the margin between different classes, promoting just correct classification but also margin around the decision boundary. L2 regularization is used to balance the trade-off between maximizing the margin and minimizing classification errors. It helps to separate classes with maximum margin but also ensures that the model remains simple and generalizes well when it faces with new data.

$$L_{\text{hinge}}(y_i, f(\mathbf{x_i})) = \max(0, 1 - y_i \cdot f(\mathbf{x_i}))$$

$$L_2 = \lambda |\mathbf{w}|_2^2 = \lambda \sum_{j=1}^{d} w_j^2$$

$$L_{\text{SVM}}(y_i, f(\mathbf{x_i}) = L_2 + L_{\text{hinge}}(y_i, f(\mathbf{x_i})$$

K-fold Cross Validation

K-fold Cross-Validation is one of the methods used to estimate the performance of machine learning model [7]. When K value increases, the obtained result in the final model is more precise, but the training of the machine learning model is increased. Initially, train data shuffled and separated into number of K sets. For each iteration, one-fold selected as validation fold. Remained one is used as a training fold. Machine learning model is trained K-1 times with each training set. This process is repeated K times. At the end of K iteration, the average of performances is found. Additionally, K-fold CV applies for each hypermeter value. In the project, K value is selected as 5. Also, train and test data set have already separated, so there no need for separation.

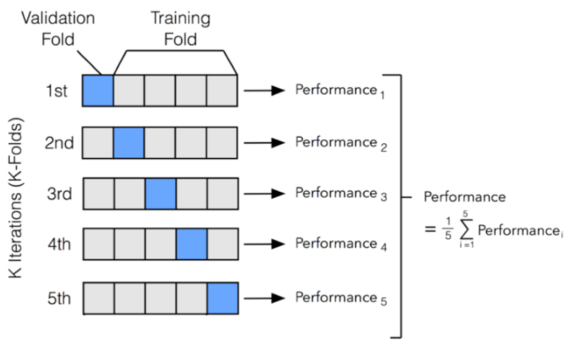

Figure 5: K-fold Cross Validation

Methods Performance Validation

The performance in each machine learning algorithms is calculated 4 different values for trained dataset. True positive is corrected predicted positive cases. True negative is correctly predicted negative cases. False positive is incorrectly predicted as positive, which is Type 1 Error. False negatives are incorrectly predicted as negative, which are Type 2 error.

$$TP = True\ Positive$$
$$TN = True\ Negative$$
$$FP = False\ Positive$$
$$FN = False\ Negative$$

Also, accuracy, precision, specify and F1 score validation metrics are used for performance checking of machine learning model. Accuracy is the proportion of all classifications that were correct, without considering positive or negative. Recall is the proportion of all actual positives that were classified correctly as positives. Precession is the ratio of correctly classified actual positives, and everything classified as positive. F1 score is a statistical measure used to evaluate the performance of a classification model, particularly during deal with imbalance dataset [8].

$$Accurarcy = \frac{TP + TN}{TP + TN + FP + FN}$$

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

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

$$F1 = 2\,x\,\frac{Precision\ x\ Recall}{TN + FP}$$

All results gathered test set output of each model is showed in the confusion matrix [9]. Confusion matrix shows the performance of a classification model by comparing the actual target values with the predicted target values.

Results

In the first part of project, dataset is prepared for training machine learning model. Initially, data set is checked for balanced or imbalanced. There is some missing value at "Arrival Delay in Minutes". Not fill parts filled with the median of the data set, so there is no lost observed in the train dataset. The same process applied for the test data set. Afterwards, the features contain scores, and integer values are standardized. At the end of the data-preprocessing, the covariance matrix of features is represented in the figure 5.

$$S = \frac{X - \mu}{\sigma}$$
$$\mu = mean.of\ feature$$
$$\sigma = standard\ devation\ of\ feature$$

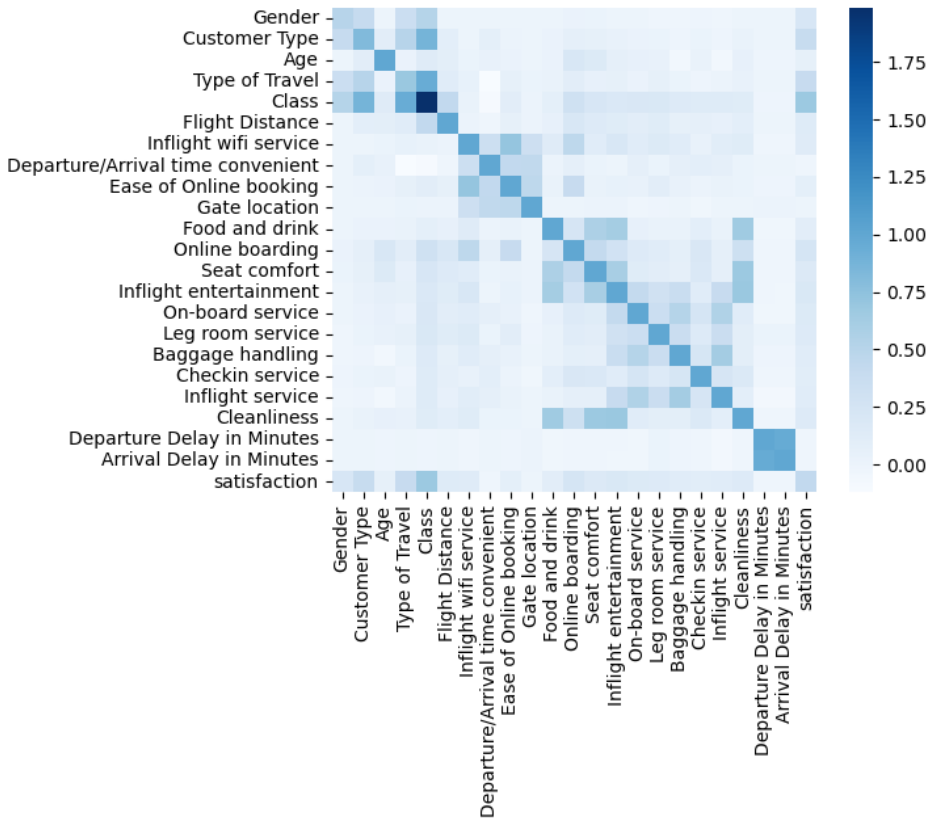

Figure 6: Covariance Matrix of Standardized Features

Moreover, the logistic regression model is implemented. The hyperparameters of logistic regression are iteration and learning rate and threshold. The highest validation parameters are selected in the hyperparameter tuning. The parameters selected for logistic regression is learning rate 0.1, iteration 500 and threshold 0.6.

Learning Rate	[0.05,0.1,0.25]
Iteration	[100,500,1000]
Threshold List	[0.4, 0.5,0.6]
Training Time	55 minutes 2.5 seconds

Table 1 Hyperparameters of Logistic Regression

Accuracy	0.87
Precision	0.91
Recall	0.79
F1 Score	0.84

Table 2 Validation Paramteres of Logistic Regression on Test Data Set

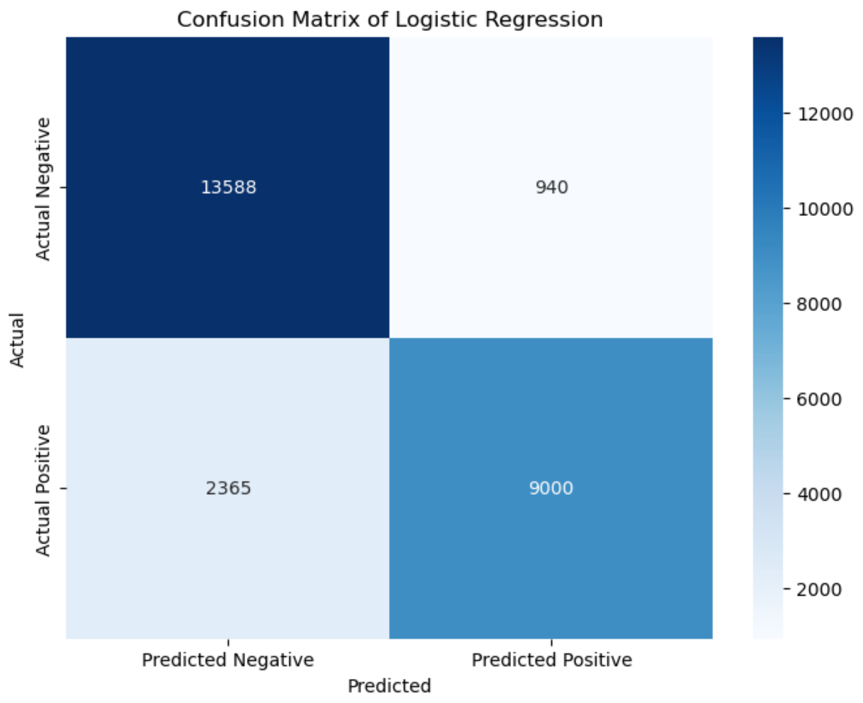

Figure 7: Confusion Matrix for Logistic Regression Run on Test Data set

Additionally, shallow neural network is implemented on code. The code for shallow neural network is designed according to equations in the analysis of machine learning part. The hyperparameters of the neural network are 8 hidden layer, 500 iteration, learning rate 0.1, and mini batch 16.

Learning Rate	[0.1, 0.25]
Iteration	[100,200,500]
Hidden Layer Number	[5,6,7,8]
Mini Batch Size	[16,32]
Training Time	302 minutes 25.0 seconds

Table 3 Hyperparameters of Neural Networks

Accuracy	0.95
Precision	0.97
Recall	0.92
F1 Score	0.94

Table 4 Validation Parameters of Neural Networks on Test Data

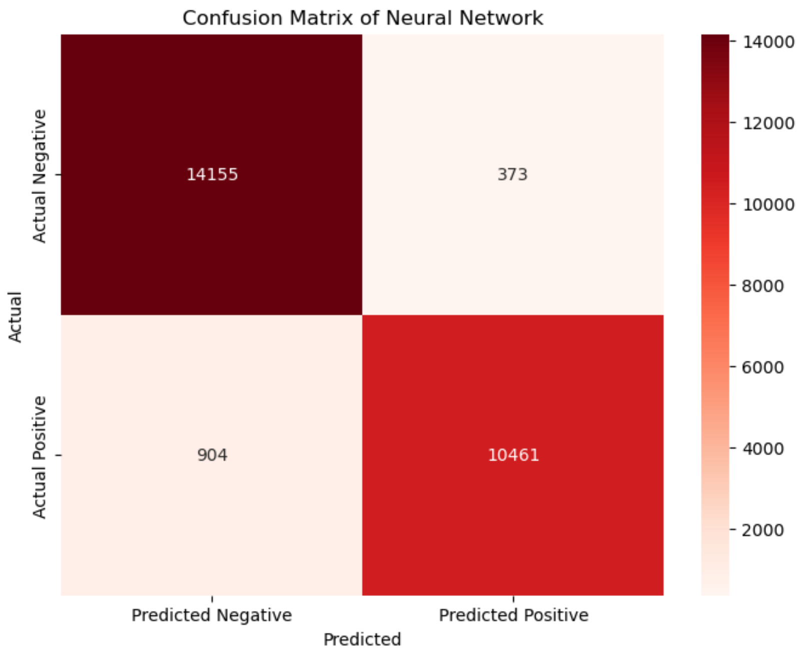

Figure 8: Confusion Matrix for Neural Network Run on Test Data set

Furthermore, Random Forest is designed according to given in the document and train preprocessed data set. The hyperparameters of random forest is in table 5. After training process, the highest. The optimal value for Random Forest is 15 features, 7 max depth and 10 trees.

Number of Trees	[3,5,10]
Max Depth	[5,6,7]
Max Features	[10,15,22]
Training Time	400 minutes

Table 5 Hyperparameters of Random Forest

Accuracy	0.84
Precision	0.78
Recall	0.90
F1 Score	0.84

Table 6 Validation Parameters of Random Forest on Test Data

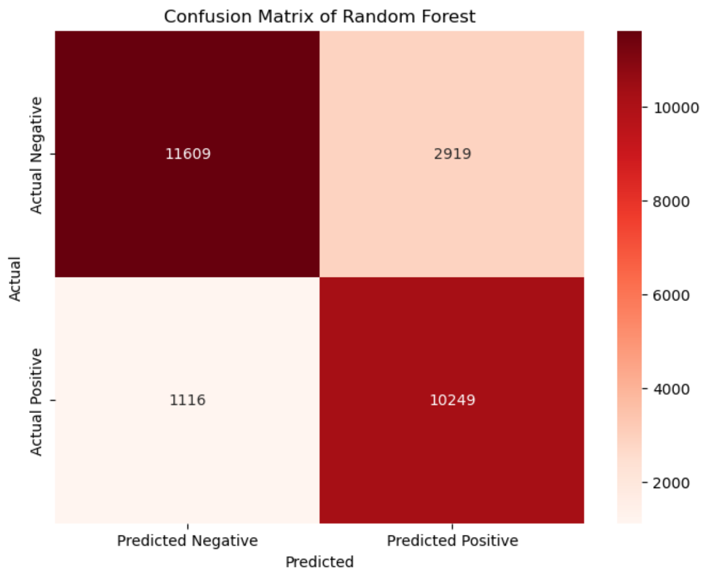

Figure 9: Confusion Matrix for Random Forest Run on Test Data

Moreover, Support Vector Machine is trained on the test data. The SVM is designed according to given formulas in the document. The hyperparameters for SVM are shown in table 7. After the training of SVM, the highest validation accuracy is selected between other hyperparameters. The selected parameters are 0.001 learning rate, 0.001 learning rate and 400 iteratipn. The accuracy of optimal parameter values on test set is shown in the table.

Iteration	[100,200,400]
Lambda	[0.001,0.01]
Learning Rate	[0.1,0.001]
Training Time	169 minutes 16.1 second

Table 7 Hyperparameters of SVM

Accuracy	0.87
Precision	0.87
Recall	0.84
F1 Score	0.85

Table 8 Validation Parameters of SVM on Test Data

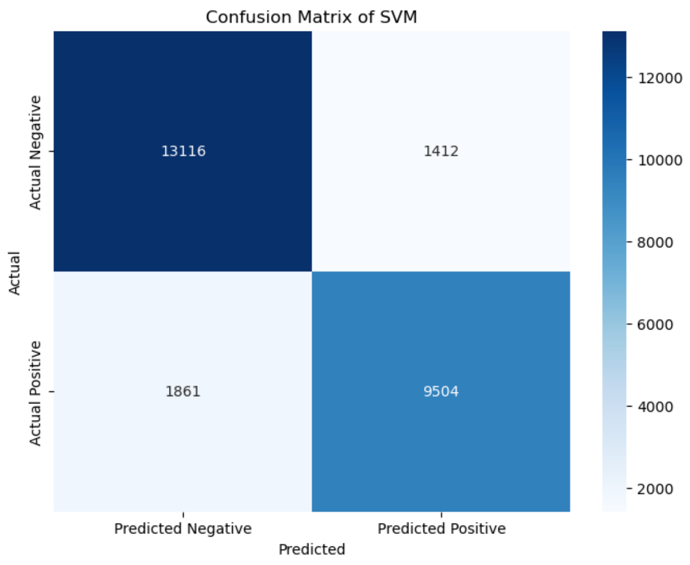

Figure 10: Confusion Matrix for SVM Run on Test Data

	Logistic Regression	Neural Network	Random Forest	SVM
Accuracy	0.87	0.95	0.84	0.87
Precision	0.91	0.97	0.78	0.87
Recall	0.79	0.92	0.90	0.84
F1 Score	0.84	0.94	0.84	0.85

Table 9 Validation Parameters of Machine Learning on Test Data

According to result obtained in the test set, all models have accuracy higher than 80 %, which shows all models work well on new data. Among machine learning models, Neural Network has highest validation parameters among other models.

Conclusion

The objective of the EEE 485/ 585 term project is to analyze consumer satisfaction by examining a data set records various aspects of flight experiences. In the term project, 4 different machine learning algorithms designed from scratch to perform binary classification categorizing outcomes as either "satisfied" or "not satisfied". The selected algorithms are Logistic Regression, Neural Networks Random Forest, and Support Vector Machine. During the training phase of each machine learning model, K-fold cross-validation is employed to achieve more accurate and reliable results. The performance of each machine learning model is assessed using validation metrics, and the effectiveness of each algorithm is further illustrated through confusion matrix. After models are trained, the highest validation parameters is selected for each machine learning model. Finally, Neural Network model worked the highest accuracy between implemented machine learning models. To sum up, all process done in the term project can be considered as a success since all algorithms are work well and give high accuracy result on test result.

References

[1] teejmahal20, 'Airline Passenger Satisfaction,' Kaggle, [Online].
Available: https://www.kaggle.com/datasets/teejmahal20/airline-passenger-satisfaction/data. [Accessed: 12-Nov-2023].
[2] Spiceworks, 'What is Logistic Regression?,' [Online].
Available: https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-logistic-regression/. [Accessed: 12-Nov-2023].
[3] IBM, 'Neural Networks,' [Online]. Available: https://www.ibm.com/topics/neural-networks. [Accessed: 12-Nov-2023].
[4] MRI Questions, 'Shallow Neural Networks,' [Online]. Available: https://mriquestions.com/shallow-networks.html. [Accessed: 12-Nov-2023].
[5] IBM, 'Random Forest,' [Online]. Available: https://www.ibm.com/topics/random-forest. [Accessed: 12-Nov-2023].
[6] V. Zhou, "Gini Impurity," [Online]. Available: https://victorzhou.com/blog/gini-impurity/. [Accessed: 12-Nov-2023].
[7] GeeksforGeeks, 'Cross Validation in Machine Learning,' [Online].
Available: https://www.geeksforgeeks.org/cross-validation-machine-learning/. [Accessed: 12-Nov-2023].
[8] Google Developers, 'Accuracy, Precision, and Recall,' [Online].
Available: https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall. [Accessed: 12-Nov-2023].
[9] IBM, 'Confusion Matrix,' [Online]. Available: https://www.ibm.com/topics/confusion-matrix. [Accessed: 12-Nov-2023].

Appendix

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import time

# Obtaining Training Data Set
training_set = pd.read_csv("train.csv")
print(training_set.head())
print(training_set.info())

print(training_set.isna().sum())

training_set['Arrival Delay in Minutes'].fillna(training_set['Arrival Delay in Minutes'].median(axis = 0),
inplace = True)

print(training_set.isna().sum())

# Dropping out unnecessarily columns
training_set = training_set.drop(columns = ['Unnamed: 0','id'])

weights_name_list = training_set.columns
training_set

# Standardization
def standardization(weights_name_list,t_set):
    t_set_l = t_set
    for i in weights_name_list :
        if t_set[i].dtype != object:
            t_set_l[i] = (t_set_l[i] - t_set_l[i].mean()) / t_set_l[i].std()
    return t_set_l

training_set = standardization(weights_name_list,training_set)

# Hierarchal Encoding
training_set['Gender'].replace({'Male': 0, 'Female': 1}, inplace=True)
training_set['Customer Type'].replace({'Loyal Customer': 1, 'disloyal Customer': 0}, inplace=True)
training_set['Type of Travel'].replace({'Personal Travel': 0, 'Business travel': 1}, inplace=True)
training_set['Class'].replace({'Eco': 0, 'Eco Plus': 1, 'Business': 2}, inplace=True)
training_set['satisfaction'].replace({'neutral or dissatisfied': 0, 'satisfied': 1}, inplace=True)

def covariance(x):
    return np.dot(x.T,x)/(x.shape[0]-1)

y = training_set['satisfaction']
training_set_encoded = training_set.drop(columns=(['satisfaction']))
weights_list = training_set.columns
weights_name_list_pre= training_set_encoded.columns
```

```
cov_mat = covariance(training_set)
sns.heatmap(cov_mat, square = True, cmap = 'Blues',xticklabels=weights_list,yticklabels=weights_list)

# Arraning Test Set

test_set = pd.read_csv("test.csv")
test_set = test_set.drop(columns = ['Unnamed: 0','id'])

test_set.isna().sum()

test_set.dropna(inplace=True)
print(test_set.shape)

test_set['Gender'].replace({'Male': 0, 'Female': 1}, inplace=True)
test_set['Customer Type'].replace({'Loyal Customer': 1, 'disloyal Customer': 0}, inplace=True)
test_set['Type of Travel'].replace({'Personal Travel': 0, 'Business travel': 1}, inplace=True)
test_set['Class'].replace({'Eco': 0, 'Eco Plus': 1, 'Business': 2}, inplace=True)
test_set['satisfaction'].replace({'neutral or dissatisfied': 0, 'satisfied': 1}, inplace=True)

print(test_set.info())

test_set_y = test_set['satisfaction'].values
test_set_x = test_set.drop(columns=["satisfaction"])
test_set_x = test_set_x.values

# Logistic Regression

class LogisticRegression():
  def __init__(self, learning_rate, W, X,b, y, epochs):
    np.random.seed(38)
    self.learning_rate = learning_rate
    self.W = W
    self.X = X
    self.b = b
    self.y = y
    self.epochs = epochs

  def sigmoid(self,input_x):
    result = 1/(1+ np.exp(-(np.dot(input_x,self.W)+self.b)))
    return result

  def log_loss(self,res,y_temp):

    cost = y_temp* np.log(res+ 1e-9) +(1-y_temp)* np.log(1-res+ 1e-9)
    return -cost.mean()

  def update_weights(self):

    m = self.y.size
    cost_list = []
```

```
        for _ in range(1,self.epochs):
            np.random.seed(42)
            start = 0
            end = 32

            # Shuffle dataset
            indices = np.arange(self.X.shape[0])
            np.random.shuffle(indices)

            X_shuffle = self.X[indices]
            y_shuffle = self.y.values[indices]

            temp_cost = 0

            for _ in range(self.X.shape[0] // 32):
                X_temp = X_shuffle[start:end]
                y_temp = y_shuffle[start:end]

                res = self.sigmoid(X_temp)

                cost = self.log_loss(res,y_temp)
                temp_cost += cost


                #compute gradient w.r.t 'W'
                dW = np.dot(X_temp.T,(res-y_temp))/m
                db = np.sum(res-y_temp)/m


                self.W = self.W - self.learning_rate * dW
                self.b = self.b - self.learning_rate * db

                start = end
                end += 32

            temp_cost = temp_cost / (self.X.shape[0] // 32)
            cost_list.append(temp_cost)

        #plt.plot(range(1,len(cost_list)+1),cost_list)
        #plt.show()
        return cost_list

    def get_parameters(self):
        return self.W, self.b

def initliaze_weights_logistic(n):
    W = np.random.randn(len(n)-1)*0.01
    b = 0
    return W,b
```

```
def K_fold_splitting_dataset(k,training_set_encoded):
    #split the data set
    training_set_shuffled = training_set_encoded.sample(frac = 1)
    splited_data_set = []

    m,n = training_set_shuffled.shape

    boundary = m // k
    for i in range(0,k):
        splited_data_set.append(training_set_shuffled.iloc[i*boundary:(i+1)*boundary])
    return splited_data_set

def confusion_matrix(actual,predicted):
    TP = 0  # True Positive
    TN = 0  # True Negative
    FP = 0  # False Positive
    FN = 0  # False Negative


    for a, p in zip(actual, predicted):
        if a == 1 and p == 1:
            TP += 1
        elif a == 0 and p == 0:
            TN += 1
        elif a == 0 and p == 1:
            FP += 1
        elif a == 1 and p == 0:
            FN += 1


    return {"TP": TP,"TN": TN,"FP": FP,"FN": FN}

def calculate_metrics(conf_matrix):
    TP = conf_matrix['TP']
    TN = conf_matrix['TN']
    FP = conf_matrix['FP']
    FN = conf_matrix['FN']

    accuracy = (TP + TN) / (TP + TN + FP + FN)
    precision = TP / (TP + FP) if (TP + FP) != 0 else 0
    recall = TP / (TP + FN) if (TP + FN) != 0 else 0
    f1_score = (2 * precision * recall) / (precision + recall) if (precision + recall) != 0 else 0


    return {
        "Accuracy": accuracy,
        "Precission": precision,
        "Recall": recall,
        "F1 Score ": f1_score
    }
```

```
def print_calculated_metrcis(cm):

    accuracy = cm["Accuracy"]
    precision = cm["Precission"]
    recall = cm["Recall"]
    f1_score = cm["F1 Score "]

    print("\nMetrics:")
    print(f"Accuracy: {accuracy:.2f}")
    print(f"Precision: {precision:.2f}")
    print(f"Recall: {recall:.2f}")
    print(f"F1 Score: {f1_score:.2f}")

    return


# Determine Hyperparameters
learning_rate = [0.05,0.1,0.25]
iteration = [100,500,1000]
threshold_list = [0.4,0.5,0.6]

splited_data_set= K_fold_splitting_dataset(5,training_set)

record_list = []
temp = []
temp_mean =[]
result_k_flod = []
weights_list = []
bias_list = []

# Selecting Hyperparameters
for threshold in threshold_list:
    for l_i in learning_rate:
        for each_iteration in iteration:
            #Initliaze weights
            # Apply Algorithms on K-1
            temp_store_values = []
            W,b = initliaze_weights_logistic(weights_name_list)
            start_time = time.time()
            print(f"Learning parameter {l_i} and iteration {each_iteration} and threshold {threshold} ")
            for each_split in range(0,len(splited_data_set)):
                # Taking out validation set
                for t_i in range(0,len(splited_data_set)):
                    if t_i != each_split:
                        temp.append(splited_data_set[t_i])
                    else:
                        val_set = splited_data_set[each_split]
                for each_temp_split in range(0,len(temp)):
                    ##Train the data set on remained sets
                    predicted_values = temp[each_temp_split]["satisfaction"]
                    traning_set_final = temp[each_temp_split].drop(columns=["satisfaction"])
                    traning_set_final = traning_set_final.values

                    model =
LogisticRegression(learning_rate=l_i,W=W,X=traning_set_final,y=predicted_values,b=b,epochs=each_it
eration)
                    model.update_weights()
                    W,b = model.get_parameters()

                # Check the validation score
                val_set_predicted_values = val_set["satisfaction"].values
                val_set_final = val_set.drop(columns=["satisfaction"])
                val_set_final = val_set_final.values

                predicted_val = 1/(1+np.exp(-(np.dot(val_set_final,W)+b)))
                predicted_val = (predicted_val>threshold).astype(int)

                cm = confusion_matrix(val_set_predicted_values,predicted_val)
                validation_score = calculate_metrics(cm)
                temp_store_values.append(validation_score)


                temp =[]

            epoch_value = {
            "Accuracy": 0,
            "Precission": 0,
            "Recall": 0,
            "F1 Score ": 0
        }

            # Taking average of all parameteres
            for val_value in range(len(temp_store_values)):
                epoch_value["Accuracy"] += temp_store_values[val_value]["Accuracy"]
                epoch_value["Precission"] += temp_store_values[val_value]["Precission"]
                epoch_value["Recall"] += temp_store_values[val_value]["Recall"]
                epoch_value["F1 Score "] += temp_store_values[val_value]["F1 Score "]

                if val_value == len(temp_store_values) - 1:
                    epoch_value["Accuracy"] = epoch_value["Accuracy"]/(len(temp_store_values))
                    epoch_value["Precission"] = epoch_value["Precission"]/(len(temp_store_values))
                    epoch_value["Recall"] = epoch_value["Recall"]/(len(temp_store_values))
                    epoch_value["F1 Score "] = epoch_value["F1 Score "]/(len(temp_store_values))
            end_time = time.time()
            elasped_time = end_time-start_time
            print_calculated_metrcis(epoch_value)
            print(f"{elasped_time:.2f}")
            print(" ")
            # record the result on list
            record_list.append([l_i,each_iteration,W,b,epoch_value,threshold])
            temp_mean =[]
            weights_list.append(W)
```

```
            bias_list.append(b)

predicted = 1/(1+ np.exp(-(np.dot(test_set_x,weights_list[26])+bias_list[26])))
predicted = (predicted>record_list[26][5]).astype(int)
cm = confusion_matrix(test_set_y,predicted)
validation_score = calculate_metrics(cm)
print_calculated_metrcis(validation_score)
confusion_matrix_list= [[cm["TN"],cm["FP"]],[cm["FN"],cm["TP"]]]

plt.figure(figsize=(8, 6))

sns.heatmap(confusion_matrix_list, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted Negative',
'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])

plt.title('Confusion Matrix of Logistic Regression')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

# Neural Network

class NeuralNetworks():

    def __init__(self,parameters,X_data,Y_data,epochs,learning_rate,mini_batch):
        self.parameters = parameters
        self.X_data= X_data
        self.Y_data = Y_data
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.prediction = 0
        self.mini_batch = mini_batch


    def sigmoid(self,z):
        return 1/(1+ np.exp(-z))

    def tanh(self,x):
        return np.tanh(x)


    def predict_nn(self,activation,x):
        L = len (self.parameters) // 2
        A = x
        result= {}

        for l in range(1,L+1):

            Z = np.dot(A,self.parameters["W"+str(l)]) + self.parameters["b"+str(l)]

            if  activation[l - 1]== "tanh":
```

```
            A = self.tanh(Z)
         elif activation[l - 1] == "sigmoid":
            A = self.sigmoid(Z)
         result["Z"+str(l)]= Z
         result["A"+str(l)] = A

      return result

   def get_parameters(self):
      return self.parameters

   def get_prediction(self,x):
      L = len(self.parameters) // 2
      activation = ["tanh","sigmoid"]
      predicted = self.predict_nn(activation,x)
      self.prediction = predicted["A"+str(L)]
      return self.prediction

   def cost_function_nn(self,predicted_values,real_result):
      cost = -(np.dot(real_result,np.log(predicted_values + 1e-9))+np.dot(1-real_result,np.log(1-
predicted_values+1e-9)))/ real_result.size
      cost = np.squeeze(cost)
      return cost

   def model_back(self,Y,result,parameters,X):
      W2 = parameters["W2"]

      Z1 = result["Z1"]
      A1 = result["A1"]
      Z2 =result["Z2"]
      A2 =result["A2"]

      Y = np.array([Y]).T
      m = Y.size

      dZ2 = A2 - Y
      dW2 = np.dot(A1.T,dZ2)/m
      db2 = np.sum(dZ2, axis=0,keepdims=True)/m
      dZ1 = np.dot(dZ2,W2.T)*(1-np.power(A1,2))
      dW1 = np.dot(X.T,dZ1)/m
      db1 = np.sum(dZ1,axis=0,keepdims=True)/m


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


      return grads

   def update_parameters(self, grads):
      self.parameters["W1"] = self.parameters["W1"] - self.learning_rate*grads["dW1"]
      self.parameters["b1"] = self.parameters["b1"] - self.learning_rate*grads["db1"]
```

```
        self.parameters["W2"] = self.parameters["W2"] - self.learning_rate*grads["dW2"]
        self.parameters["b2"] = self.parameters["b2"] - self.learning_rate*grads["db2"]

        return self.parameters

    def train_nn(self):
        cost_list = []
        activation = ["tanh","sigmoid"]


        for epoch in range(0,self.epochs):

            np.random.seed(42)

            # Shuffle dataset
            indices = np.arange(self.Y_data.shape[0])
            np.random.shuffle(indices)

            X_shuffle = self.X_data[indices]
            y_shuffle = self.Y_data.values[indices]


            # Adding mini batch
            start = 0
            end = self.mini_batch



            num_batches = self.Y_data.shape[0] // self.mini_batch


            for _ in range(num_batches):

                x_batch = X_shuffle[start:end]
                y_batch = y_shuffle[start:end]

                # Run forward
                predicted = self.predict_nn(activation,x_batch)

                # Find cost
                cost = self.cost_function_nn(predicted["A2"],y_batch)
                cost_list.append(cost)

                # Backpropagate
                grads = self.model_back(y_batch,predicted,self.parameters,x_batch)

                # Update Parameters
                self.parameters = self.update_parameters(grads)

                start = end
                end += self.mini_batch
        return self.parameters

    def init_parameters_nn(s_input,s_hidden,s_output):

    ## Initliaze parameters in the network
    W1 = np.random.randn(s_input,s_hidden)*0.01
    b1 = np.zeros((1,s_hidden))
    W2 = np.random.randn(s_hidden,s_output)*0.01
    b2 = np.zeros((1,s_output))

    parameters = {"W1": W1,"b1": b1,"W2": W2,"b2": b2}

    return parameters

# Determine Hyperparameters
number_hidden_list = list(range(5,9))
iteration_list  = [200,500,1000]
learning_rate_list = [0.1,0.25]
mini_batch_list = [16,32]
splited_data_set= K_fold_splitting_dataset(5,training_set)
parameters_list = []
record_list = []


for mini_batch in mini_batch_list:
    for number_hidden in number_hidden_list:
        for iteration in iteration_list:
            for learning_rate in learning_rate_list:
                #Initliaze weights
                # Apply Algorithms on K-1
                parameters = init_parameters_nn(s_input = len(weights_name_list)-1,s_hidden=number_hidden
,s_output = 1)
                print(f"Learning parameter {learning_rate} and iteration {iteration} hidden {number_hidden}
mini batch: {mini_batch}")
                temp_store_values = []
                for each_split in range(0,len(splited_data_set)):
                    # Taking out validation set
                    for t_i in range(0,len(splited_data_set)):
                        if t_i != each_split:
                            temp.append(splited_data_set[t_i])
                        else:
                            val_set = splited_data_set[each_split]
                    for each_temp_split in range(0,len(temp)):
                    ##Train the data set on remained sets
                        traning_set_y = temp[each_temp_split]["satisfaction"]

                        traning_set_x = temp[each_temp_split].drop(columns=["satisfaction"])
                        traning_set_x = traning_set_x.values
                        model =
NeuralNetworks(parameters=parameters,epochs=iteration,learning_rate=learning_rate,X_data=traning_se
t_x,Y_data=traning_set_y,mini_batch=mini_batch)
                        model.train_nn()
                        parameters = model.get_parameters()

                    # Check the validation score
                    val_set_y = val_set["satisfaction"].values
                    val_set_x = val_set.drop(columns=["satisfaction"])
                    val_set_x = val_set_x.values

                    model_val =
NeuralNetworks(parameters=parameters,epochs=iteration,learning_rate=learning_rate,X_data=val_set_x,
Y_data=val_set_y,mini_batch=mini_batch)

                    predicted_nn_value = model_val.get_prediction(val_set_x)
                    predicted_nn = (predicted_nn_value>0.6).astype(int)
                    cm_nn = confusion_matrix(val_set_y,predicted_nn)
                    validation_score_nn = calculate_metrics(cm_nn)
                    temp_store_values.append(validation_score_nn)

                    temp =[]

                epoch_value = {
                    "Accuracy": 0,
                    "Precission": 0,
                    "Recall": 0,
                    "F1 Score ": 0
                    }

                # Taking average of all parameteres
                for val_value in range(len(temp_store_values)):
                    epoch_value["Accuracy"] += temp_store_values[val_value]["Accuracy"]
                    epoch_value["Precission"] += temp_store_values[val_value]["Precission"]
                    epoch_value["Recall"] += temp_store_values[val_value]["Recall"]
                    epoch_value["F1 Score "] += temp_store_values[val_value]["F1 Score "]

                    if val_value == len(temp_store_values) - 1:
                        epoch_value["Accuracy"] = epoch_value["Accuracy"]/(len(temp_store_values))
                        epoch_value["Precission"] = epoch_value["Precission"]/(len(temp_store_values))
                        epoch_value["Recall"] = epoch_value["Recall"]/(len(temp_store_values))
                        epoch_value["F1 Score "] = epoch_value["F1 Score "]/(len(temp_store_values))

                print_calculated_metrcis(epoch_value)
                print(" ")

                # record the result on list
                record_list.append({"learning_rate":
learning_rate,"iteration":iteration,"number_hidden":number_hidden,"parameters":parameters,"epoch":epo
ch_value,"mini batch":mini_batch})
                temp_mean =[]
        parameters_list.append(parameters)


model_test =
NeuralNetworks(parameters=record_list[20]["parameters"],epochs=record_list[20]["iteration"],learning_r
ate=record_list[20]["learning_rate"],X_data=test_set_x,Y_data=test_set_y,mini_batch=record_list[20]["m
ini batch"])

predicted_test_nn_value = model_test.get_prediction(test_set_x)
predicted_nn = (predicted_test_nn_value>0.6).astype(int)

cm_nn = confusion_matrix(test_set_y,predicted_nn)
validation_score_nn = calculate_metrics(cm_nn)
print_calculated_metrcis(validation_score_nn)
confusion_matrix_list_nn= [[cm_nn["TN"],cm_nn["FP"]],[cm_nn["FN"],cm_nn["TP"]]]


plt.figure(figsize=(8, 6))

sns.heatmap(confusion_matrix_list_nn, annot=True, fmt='d', cmap='Reds', xticklabels=['Predicted
Negative', 'Predicted Positive'],
      yticklabels=['Actual Negative', 'Actual Positive'])

plt.title('Confusion Matrix of Neural Network')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()


# SVM

class SVM:

  def __init__(self, learning_rate, lambda_param, n_iters, w, b):
    self.lr = learning_rate
    self.lambda_param = lambda_param
    self.n_iters = n_iters
    self.w = w
    self.b = b


  def cost(self,output_of_svm):
    # Hinge loss per sample: max(0, 1 - margin)
    loss_values = np.maximum(0, 1 - output_of_svm)

    # Compute total loss = regularization + average hinge loss
    loss = self.lambda_param * np.sum(self.w**2) + np.mean(loss_values)

    return loss

  def get_parameters(self):
        return self.w , self.b


    def fit(self, X, y):
        n_samples, n_features = X.shape

        y_ = np.where(y <= 0, -1, 1)

        total_cost = []
        for i in range(self.n_iters):
            cost_temp = []
            for idx, x_i in enumerate(X):
                forward_run = y_[idx] * (np.dot(x_i, self.w) - self.b)
                cost = self.cost(forward_run)
                cost_temp.append(cost)
                condition = forward_run >= 1
                if condition:
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.lr * (2 * self.lambda_param * self.w - np.dot(x_i, y_[idx]))
                    self.b -= self.lr * y_[idx]
            cost_temp = np.array(cost_temp)
            total_cost.append(float(np.mean(cost_temp)))

        return total_cost

    def predict(self, X):
        approx = np.dot(X, self.w) - self.b
        approx = np.sign(approx)
        approx = np.where(approx <= -1, 0, 1)
        return approx


def init_weights_svm(number_of_features):
    # init weights
    W = np.zeros(number_of_features)
    b = 0

    parameteres = {
        "W": W,
        "b": b
    }

    return parameteres


iterations_list_svm = [100,200,400]
lambda_parameteres_svm  = [0.001,0.01]
lr_parameteres_svm = [0.1,0.001]

splited_data_set= K_fold_splitting_dataset(5,training_set)
```

```
counter = 0
record_list_svm = []
n_samples, n_features = training_set.shape
parameters = init_weights_svm(training_set.shape[1]-1)

np.random.seed(42)

# Applying K- fold Cross Validation

for iteration in iterations_list_svm :
    for lr in lr_parameteres_svm:
        for lambda_svm in lambda_parameteres_svm:

                print(f"Learning parameter:{lr} and iteration:{iteration} lambda:{lambda_svm} list
number:{counter} ")
                temp_store_values = []
                temp =[]
                start_time = time.time()
                parameters = init_weights_svm(training_set.shape[1]-1)
                for each_split in range(0,len(splited_data_set)):
                    print(each_split)
                    # Taking out validation set
                    for t_i in range(0,len(splited_data_set)):
                        if t_i != each_split:
                            temp.append(splited_data_set[t_i])
                        else:
                            val_set = splited_data_set[each_split]
                    for each_temp_split in range(0,len(temp)):
                    ##Train the data set on remained sets

                        traning_set_y = temp[each_temp_split]["satisfaction"]

                        traning_set_x = temp[each_temp_split].drop(columns=["satisfaction"])
                        traning_set_x = traning_set_x.values

                        w = parameters['W']
                        b = parameters['b']

                        model = SVM(n_iters=iteration,lambda_param=lambda_svm,learning_rate=lr,w=w,b= b,)
                        cost = model.fit(X=traning_set_x,y=traning_set_y)
                        parameters["W"], parameters["b"] = model.get_parameters()

                    # Check the validation score
                    val_set_y = val_set["satisfaction"].values
                    val_set_x = val_set.drop(columns=["satisfaction"])
                    val_set_x = val_set_x.values

                    model_val =
SVM(n_iters=iteration,lambda_param=lambda_svm,learning_rate=lr,w=parameters["W"],b=
parameters["b"])
```

```
                predicted_svm_value = model_val.predict(val_set_x)
                cm_svm = confusion_matrix(val_set_y,predicted_svm_value)
                validation_score_svm = calculate_metrics(cm_svm)

                temp_store_values.append(validation_score_svm)

                temp =[]

            epoch_value = {
            "Accuracy": 0,
            "Precission": 0,
            "Recall": 0,
            "F1 Score ": 0
            }

            # Taking average of all parameteres
            for val_value in range(len(temp_store_values)):
                epoch_value["Accuracy"] += temp_store_values[val_value]["Accuracy"]
                epoch_value["Precission"] += temp_store_values[val_value]["Precission"]
                epoch_value["Recall"] += temp_store_values[val_value]["Recall"]
                epoch_value["F1 Score "] += temp_store_values[val_value]["F1 Score "]

            if val_value == len(temp_store_values) - 1:
                epoch_value["Accuracy"] = epoch_value["Accuracy"]/(len(temp_store_values))
                epoch_value["Precission"] = epoch_value["Precission"]/(len(temp_store_values))
                epoch_value["Recall"] = epoch_value["Recall"]/(len(temp_store_values))
                epoch_value["F1 Score "] = epoch_value["F1 Score "]/(len(temp_store_values))

            end_time = time.time()
            elasped_time = end_time-start_time
            print_calculated_metrcis(epoch_value)
            print(f"{elasped_time:.2f}")
            print(" ")
            counter += 1

            # record the result on list
            record_list_svm.append({"learning_rate":
lr,"iteration":iteration,"lambda":lambda_svm,"parameters":parameters,"epoch":epoch_value})
            temp_mean =[]


modeltest = SVM(n_iters=record_list_svm[2]["iteration"],
            lambda_param=record_list_svm[2]["lambda"],
            learning_rate=record_list_svm[2]["learning_rate"],
            w=record_list_svm[2]["parameters"]["W"],
            b= record_list_svm[2]["parameters"]["b"],
            c=record_list_svm[2]["c"] )

result = modeltest.predict(test_set_x)

cm_svm = confusion_matrix(test_set_y,result)
```

```
validation_score_svm = calculate_metrics(cm_svm)
print_calculated_metrcis(validation_score_svm)


confusion_matrix_list_svm= [[cm_svm["TN"],cm_svm["FP"]],[cm_svm["FN"],cm_svm["TP"]]]


plt.figure(figsize=(8, 6))

sns.heatmap(confusion_matrix_list_svm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted
Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])

plt.title('Confusion Matrix of SVM')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

from collections import Counter

class Node:
  def __init__(self,feature_index= None, threshold=None,left=None,right= None,*, value = None):
    self.feature_index = feature_index
    self.threshold = threshold
    self.left = left
    self.right = right
    self.value = value

  def is_leaf_node(self):
    return self.value is not None

def split_dataset(X, y, feature_idx, threshold):
    """Split dataset into left and right subsets based on a feature threshold."""
    left_mask = X[:, feature_idx] <= threshold
    right_mask = ~left_mask
    return X[left_mask], y[left_mask], X[right_mask], y[right_mask]


def most_common_label(y):
  """Return the most common class label in y."""
  counter = Counter(y)
  return counter.most_common(1)[0][0]

class DecisionTree:
  def __init__(self,max_depth= 10, min_samples_split = 2, n_features = None):
    np.random.seed(38)
    self.min_samples_split= min_samples_split
    self.max_depth = max_depth
    self.n_features = n_features
    self.root = None
```

```
def predict(self,X):
    return np.array([self.predict_one(row) for row in X])

def predict_one(self,X):
    current = self.root
    while current.value is None:
        if X[current.feature_index] <= current.threshold:
            current = current.left
        else:
            current = current.right
    return current.value

def gini_impurity(self,input):
    classes, counts = np.unique(input, return_counts = True)
    p = counts /counts.sum()
    return 1.0 - np.sum(p**2)


def fit(self,train_X, train_Y):
    if self.n_features is None:
        self.n_features = train_X.shape[1]
    self.root = self.grow_tree(train_X,train_Y,depth=0)

def grow_tree(self, train_X, train_Y, depth):
    n_sample, n_feats = train_X.shape
    n_labels = len(np.unique(train_Y))

    # check the stopping criteria
    if (depth >= self.max_depth or n_labels == 1 or n_samples<self.min_samples_split) :
        leaf_value = most_common_label(train_Y)
        return Node(value = leaf_value)

    feat_idxs = np.random.choice(n_feats,self.n_features,replace=False)

    # find the best split
    best_feature, best_thresh = self.best_split(train_X,train_Y,feat_idxs)

    # If no improvement, return leaf
    if best_feature is None:
        leaf_value = most_common_label(y)
        return Node(value=leaf_value)

    # create child nodes
    X_left, y_left, X_right, y_right = split_dataset(train_X, train_Y, best_feature, best_thresh)
    left = self.grow_tree(X_left,y_left,depth+1)
    right = self.grow_tree(X_right,y_right,depth+1)
    return Node(feature_index=best_feature, threshold=best_thresh,left=left,right=right)

def best_split(self, train_X, train_Y, feat_idxs):
    best_gain = 0.0
    best_feature, best_threshold = None, None
```

```
        current_impurity = self.gini_impurity(y)
        n_samples = train_X.shape[0]

        for feat in feat_idxs:
            values = np.unique(train_X[:, feat])
            for threshold in values :
                X_left, y_left, X_right, y_right = split_dataset(train_X, train_Y, feat, threshold)
                p_left = len(y_left) / n_samples
                p_right = 1 - p_left
                gain = current_impurity - (p_left * self.gini_impurity(y_left) + p_right *
self.gini_impurity(y_right))
                if gain > best_gain:
                        best_gain = gain
                        best_feature = feat
                        best_threshold = threshold

        return best_feature, best_threshold

class RandomForest:
    """A simple random forest classifier for binary classification."""
    def __init__(self, n_trees=10, max_depth=5, min_samples_split=2, max_features=8,trees=[]):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.max_features = max_features
        self.trees = trees


    def bootstrap_sample(self, X, y):
        """Create a bootstrap sample from X and y."""
        n = len(X)
        indices = np.random.randint(0, n, n)
        return X[indices], y[indices]

    def fit(self, X, y):
        n_features_total = X.shape[1]
        n_features_sub = self.max_features

        self.trees = []
        for j in range(self.n_trees):
            # Bootstrap sample

            X_sample, y_sample = self.bootstrap_sample(X, y)
            # Train a decision tree
            tree = DecisionTree(max_depth=self.max_depth,
                          min_samples_split=self.min_samples_split,
                          n_features=n_features_sub)
            tree.fit(X_sample, y_sample)
            self.trees.append(tree)

n_trees_list = [3,5,10]
```

```
max_depth_list = [5,6,7]
max_feature_list = [10,15,22]
splited_data_set= K_fold_splitting_dataset(5,training_set)
counter = 0
record_list_rm = []
n_samples, n_features = training_set.shape


for ma_feature in max_feature_list:
    for max_depth in max_depth_list:
        for n_trees in n_trees_list:
            trees = []
            print(f" Features {ma_feature}  Max Depth {max_depth}  Number of Trees : {n_trees} ")
            temp_store_values = []
            temp =[]
            start_time = time.time()


            for each_split in range(0,len(splited_data_set)):

                # Taking out validation set
                for t_i in range(0,len(splited_data_set)):
                    if t_i != each_split:
                        temp.append(splited_data_set[t_i])
                    else:
                        val_set = splited_data_set[each_split]
                for each_temp_split in range(0,len(temp)):
                    ##Train the data set on remained sets

                    traning_set_y = temp[each_temp_split]["satisfaction"].values
                    traning_set_x = temp[each_temp_split].drop(columns=["satisfaction"])
                    traning_set_x = traning_set_x.values

                    model = RandomForest(n_trees=n_trees,
max_depth=max_depth,max_features=ma_feature,trees=trees)
                    model.fit(traning_set_x,traning_set_y)
                    trees = model.trees

                # Check the validation score
                val_set_y = val_set["satisfaction"]
                val_set_x = val_set.drop(columns=["satisfaction"])
                val_set_x = val_set_x.values

                model_val = RandomForest(n_trees=n_trees,
max_depth=max_depth,max_features=ma_feature,trees=trees)

                predicted_rm_value = model_val.predict(val_set_x)
                cm_rm = confusion_matrix(val_set_y,predicted_rm_value)
                validation_score_rm = calculate_metrics(cm_rm)

                temp_store_values.append(validation_score_rm)
                temp =[]

            epoch_value = {
                "Accuracy": 0,
                "Precission": 0,
                "Recall": 0,
                "F1 Score ": 0
                }

                # Taking average of all parameteres
            for val_value in range(len(temp_store_values)):
                epoch_value["Accuracy"] += temp_store_values[val_value]["Accuracy"]
                epoch_value["Precission"] += temp_store_values[val_value]["Precission"]
                epoch_value["Recall"] += temp_store_values[val_value]["Recall"]
                epoch_value["F1 Score "] += temp_store_values[val_value]["F1 Score "]

            if val_value == len(temp_store_values) - 1:
                epoch_value["Accuracy"] = epoch_value["Accuracy"]/(len(temp_store_values))
                epoch_value["Precission"] = epoch_value["Precission"]/(len(temp_store_values))
                epoch_value["Recall"] = epoch_value["Recall"]/(len(temp_store_values))
                epoch_value["F1 Score "] = epoch_value["F1 Score "]/(len(temp_store_values))

            end_time = time.time()
            elasped_time = end_time-start_time
            print_calculated_metrcis(epoch_value)
            print(f"{elasped_time:.2f}")
            print(" ")
            counter += 1

            # record the result on list
            record_list_rm.append({"trees":trees,"n trees":n_trees,"Max Depth":max_depth,"Max
Features":ma_feature,"epoch":epoch_value})
            temp_mean =[]



    def predict(self, X):
        # Get predictions from all trees
        tree_preds = [tree.predict(X) for tree in self.trees]
        # Convert to array (n_trees, n_samples)
        tree_preds = np.array(tree_preds)
        # Majority vote: sum along trees, if > half trees say 1, predict 1 else 0
        summed_predictions = np.sum(tree_preds, axis=0)
        # Majority vote threshold
        final_preds = (summed_predictions > self.n_trees / 2).astype(int)
        return final_preds

    def get_trees(self):
        return self.trees
```

```
# Test a random forest
rf = RandomForest(n_trees=record_list_rm[17]["n trees"], max_depth=record_list_rm[17]["Max
Depth"],max_features=record_list_rm[17]["Max Features"],trees=record_list_rm[17]["trees"])

# Compute accuracy
result = rf.predict(test_set_x)
print(result)

cm_randomforest = confusion_matrix(test_set_y,result)
validation_score_randomforest = calculate_metrics(cm_randomforest)
print_calculated_metrcis(validation_score_randomforest)


confusion_matrix_list_randomforest =
[[cm_randomforest["TN"],cm_randomforest["FP"]],[cm_randomforest["FN"],cm_randomforest["TP"]]]


plt.figure(figsize=(8, 6))

sns.heatmap(confusion_matrix_list_randomforest, annot=True, fmt='d', cmap='Reds',
xticklabels=['Predicted Negative', 'Predicted Positive'],
        yticklabels=['Actual Negative', 'Actual Positive'])

plt.title('Confusion Matrix of Random Forest')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```