Kemal Enes Akyüz - 22003521
Efe Tarhan - 22002840

EEE485 - Statistical Learning and Data Analytics Term Project Final Report

Flight Price Prediction Using Linear Regression, Random Forest and Deep Learning.

Introduction

This report explores the relationship between airline ticket prices and various factors like airline, destination, flight times, booking period, and flight class. The study employs custom-built machine learning models, using only fundamental Python libraries like numpy and pandas, to predict ticket prices based on these features. This report will outline the problem, describe the dataset, detail the methodologies employed, and the results obtained using these methods.

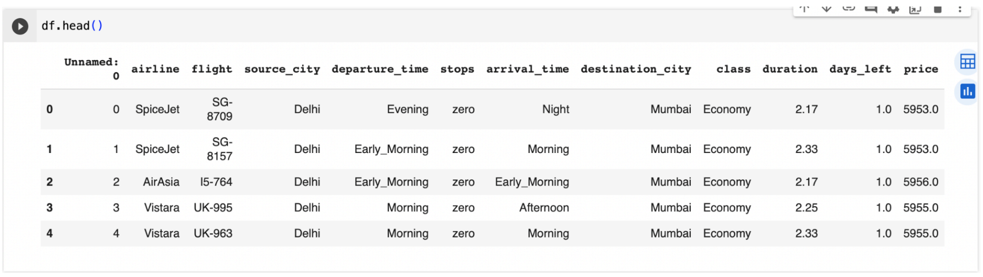
Dataset Description and Analysis

The problem that will be dealt with for this project is the prediction of ticket prices of the six biggest airline companies in India and their flights between 6 major cities. Several categorical and numerical features in the dataset will be used for the prediction task. Further, the relation between the prices and individual features will be searched, i.e., what type of correlation is there between the class of the flight and the price, or how the number of days before buying the flight affects the price. These questions and problems will be worked on throughout this project.

The dataset [1] used in this project contains the price of plane tickets of 6 Indian airline companies for the flights between 6 major cities of India. The features of the dataset will be explained in further detail.

1. **Airline**: 6 categorical values that include the names of the airline companies.
2. **Flight**: The plane's flight code, which is a categorical value.
3. **Source City**: 6 categorical values that denote the names of the cities where the flight is being departed (Bangalore, Chennai, Delhi, etc.)
4. **Departure Tim**e: 6 categorical time labels for departure time, the quantized version of the 24 hours into six intervals (Early morning, morning, afternoon, etc.)
5. **Stops**: 3 categorical values that address the number of stops of the flight (zero, one, two, or more)
6. **Arrival Time**: 6 categorical time labels for arrival time, the quantized version of the 24 hours into six intervals (Early morning, morning, afternoon, etc.).
7. **Destination City**: 6 categorical values denote the flight's landing location.
8. **Class**: 2 categorical values which indicate if the flight is "economy" or "business" class.
9. **Duration**: A continuous feature representing how long the flight is expected to last.
10. **Days Left**: how many days earlier the ticket has been bought before the flight takes off.
11. **Price**: This final feature is the price of the flight, which is a continuous variable and target of the prediction task of this dataset.

The head of the dataset can be seen in Figure 1. The data has been extracted using the DataFrame object of the pandas library.



```
df.head()
```

	Unnamed: 0	airline	flight	source_city	departure_time	stops	arrival_time	destination_city	class	duration	days_left	price
0	0	SpiceJet	SG-8709	Delhi	Evening	zero	Night	Mumbai	Economy	2.17	1.0	5953.0
1	1	SpiceJet	SG-8157	Delhi	Early_Morning	zero	Morning	Mumbai	Economy	2.33	1.0	5953.0
2	2	AirAsia	I5-764	Delhi	Early_Morning	zero	Early_Morning	Mumbai	Economy	2.17	1.0	5956.0
3	3	Vistara	UK-995	Delhi	Morning	zero	Afternoon	Mumbai	Economy	2.25	1.0	5955.0
4	4	Vistara	UK-963	Delhi	Morning	zero	Morning	Mumbai	Economy	2.33	1.0	5955.0

Fig.1 First 5 entries of the plane ticket price prediction dataset

Because the categorical values are improper for linear regression and neural network approaches, the dataset has been edited and converted into a new one. This can be considered as a preprocessing step before working on the data. The non-ordered categorical data has been converted to numerical values using one-hot encoding. Since there are many (thousands of) different flight codes, the feature has been removed from the dataset after ensuring that the naming convention does not provide much information directly about the ticket price [2]. Type conversions applied for each feature are described below.

1. **Airline**: Converted into six numerical data using 6-bit one hot encoding.
2. **Flight**: The feature has been removed from the dataset
3. **Source City**: Converted into six numerical data using 6-bit one hot encoding.
4. **Departure Tim**e: Since this data is naturally ordered, the data has been converted to integers between 1 and 6.
5. **Arrival Time**: Since this data is naturally ordered, the data has been converted to integers between 1 and 6.
6. **Destination City**: Converted into six numerical data using 6-bit one hot encoding.

7. **Class**: 1 bit one hot encoding has been applied for indicating “economy” or “business” classes.

After describing the dataset, further analysis has been applied for understanding the existing relationships inside the data. First step of the analysis is finding the relation between the features of the dataset which can be achieved using a correlation matrix. The correlation matrix has been plotted as a heatmap using the Seaborn library of Python, which can be seen in Figure 2.

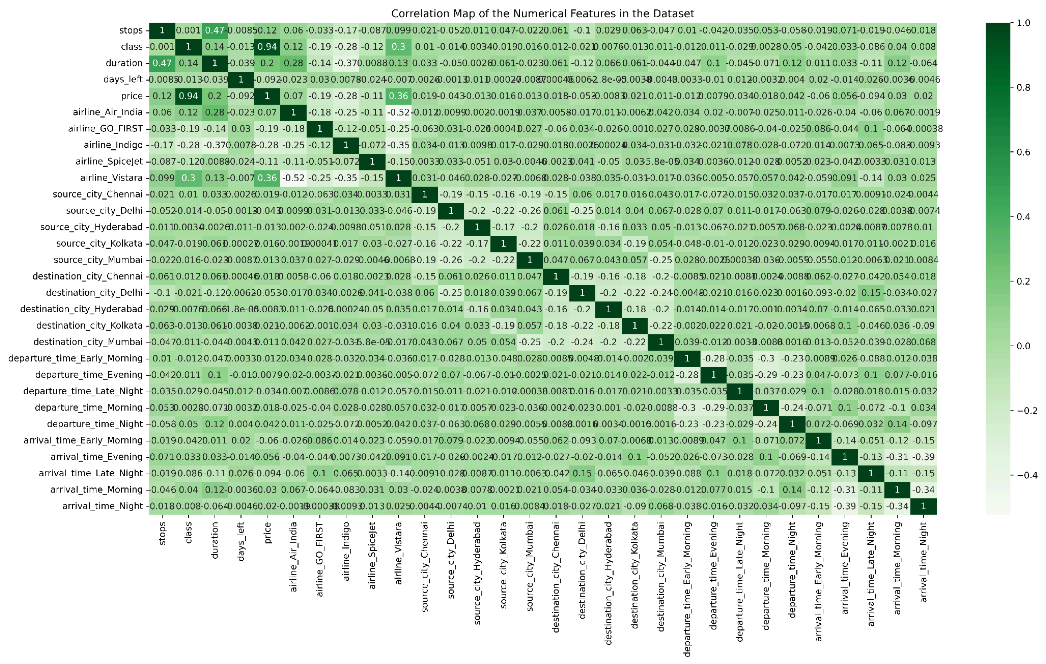
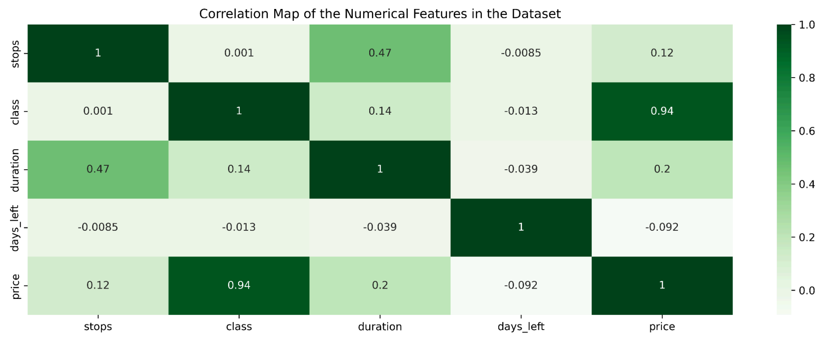


Fig.2 Heatmap of the correlation matrix of the dataset

As it can be seen most of the features in the dataset are uncorrelated and therefore a linear inference can not be directly extracted. For further observation the heatmap of the numerical features inside the dataset have been inspected separately, which can be seen in Figure 3.



	stops	class	duration	days_left	price
stops	1	0.001	0.47	-0.0085	0.12
class	0.001	1	0.14	-0.013	0.94
duration	0.47	0.14	1	-0.039	0.2
days_left	-0.0085	-0.013	-0.039	1	-0.092
price	0.12	0.94	0.2	-0.092	1

Fig.3 Heatmap of the correlation matrix of the numerical data in the dataset

From this part of the heatmap it can be seen that the class of the flight is fully correlated with its price with a score of 0.94 which was a highly expected result, also the number of stops is correlated with a score of 0.47 with the flight duration, which is also an expected result since the increased number of stops also will increase the flight duration. Also there is a notable relationship between the duration of the flight and the flight price which might be due to the increase of the price with increased fuel consumption.

The relationship between the flight prices and airlines have been inspected using a box plot, which can be examined from Figure 4. As it can be seen the airlines Vistara and Air India have a significantly higher flight price compared to the other 4 airlines where they have similar average flight prices.

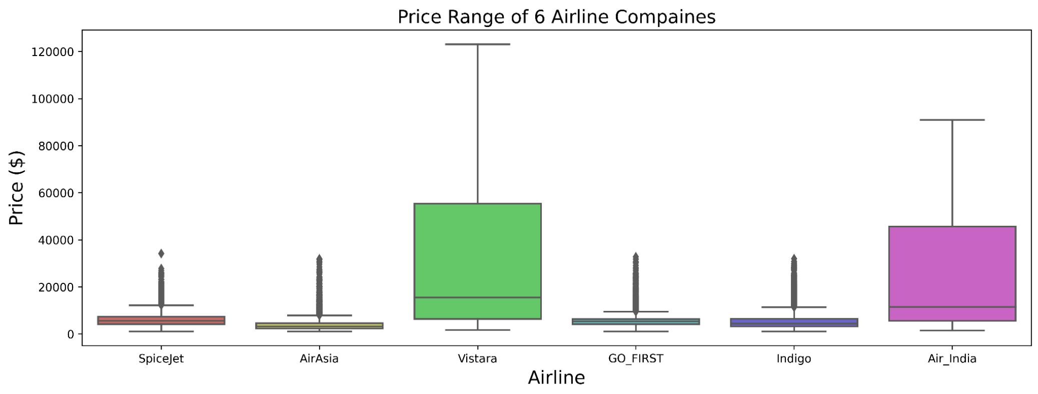


Fig.4 The plot of price ranges and 6 airlines

Figure 5 represents the relationship between the number of days before buying the ticket and ticket prices. As it can be seen the flight prices have an increasing trend as the days left until the departure decrease. But an anomaly can be seen as when one day left for the departure the prices decrease drastically which can be attributed to the last minute sales to fill the seats.

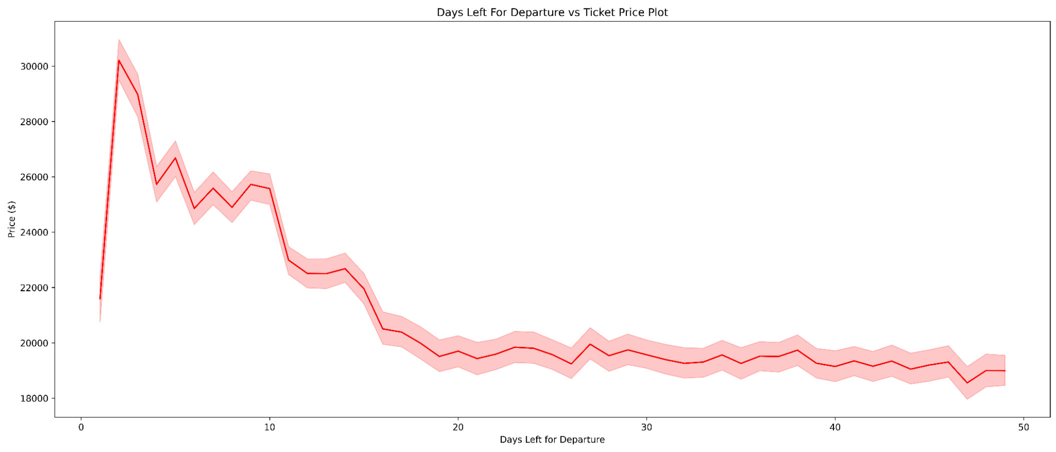


Fig.5 The plot of price vs the days left before the departure

After the ratios of the classes have been determined, the flight prices are inspected by comparing their classes. It can be seen that the business class tickets have a higher price range compared to the economy class tickets (but have low sample density compared to the economy tickets). The violin plots for comparing the ticket prices according to the classes can be seen in Figure 6.

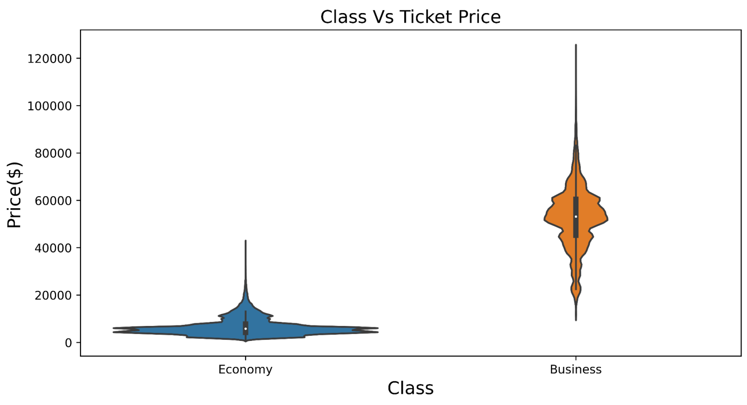


Fig.6 Violin plot of the ticket prices vs the ticket classes

Also the relationship between the departure and arrival times and the ticket prices have been inspected, the prices have seen to be decreased at the late night hours which might indicate that late night flights are not usually preferred by customers which have led the airline to decrease the prices. The box plots can be seen in Figure 7.

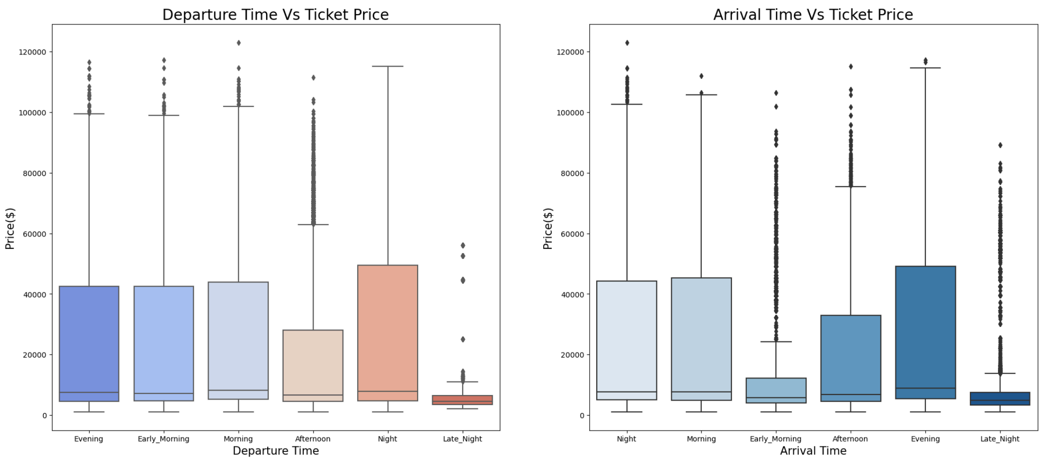


Fig.7 Box plot of the ticket prices vs the arrival time and departure times

A similar graph can be drawn to inspect the effect of selecting a destination city or a source city on the price of the ticket. In this case however the results appeared to be uncorrelated with the cities since there are not any significant differences. The result of this comparison can be seen in Figure 8.

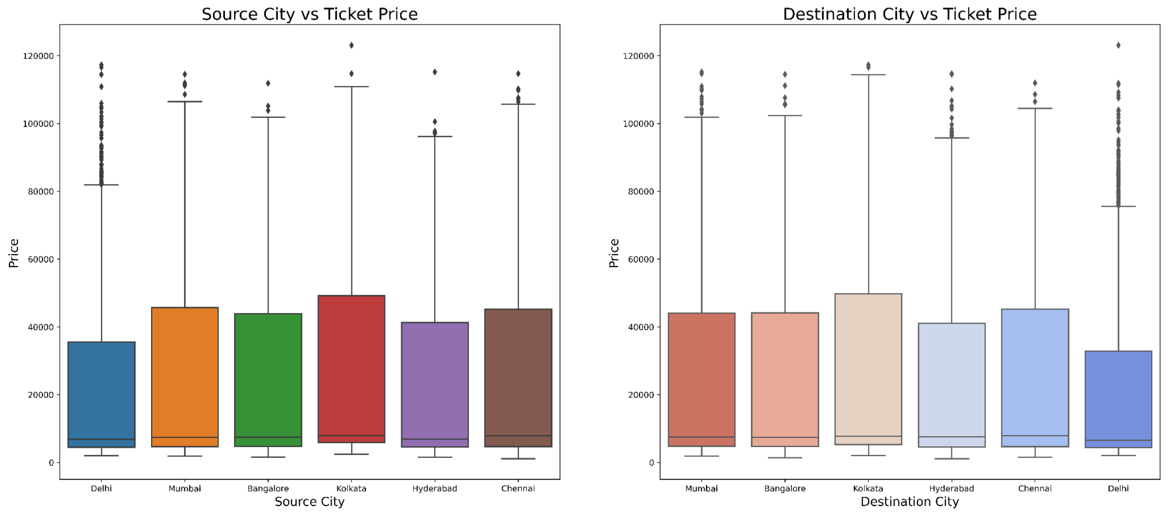


Fig.8 Box plot of the ticket prices vs the source city and destination city

Lastly the ticket prices have been inspected by comparing them with the days left for departure when the flight has been bought, as expected the airline companies Vistara and Air India have the maximum value for the ticket prices and others seem to be lower. The result of this graph can be seen in Figure 9.

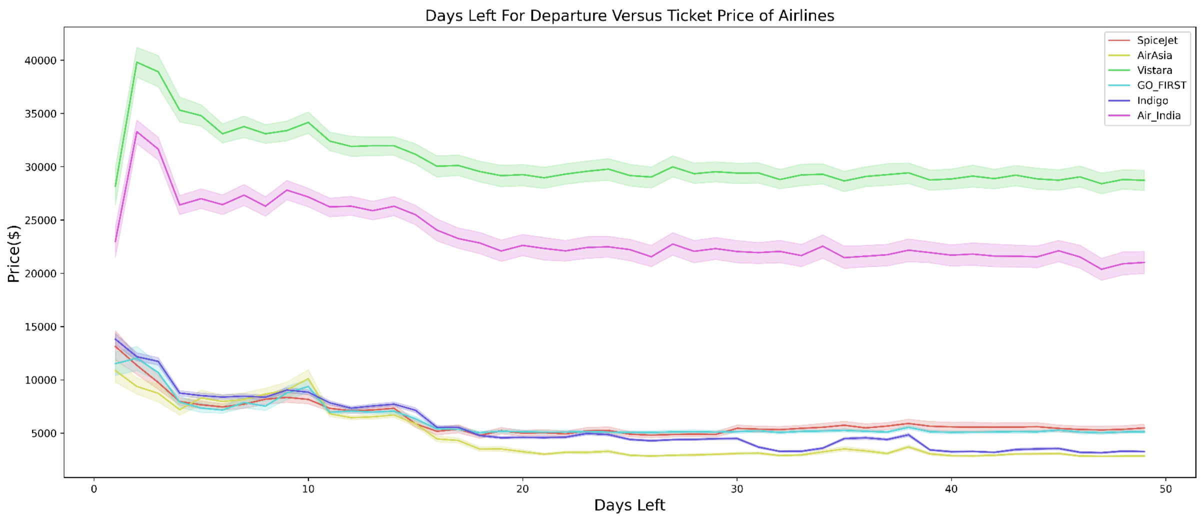


Fig.9 Plot of ticket prices compared according to the plane companies and days lefts until departure

This concludes the data analysis that provides the necessary insight for the operations made for prediction. The next section will be about the description and operations that have been done to perform the selected machine learning methods.

Applied Methods

Four methods have been applied for training a model that can be used for predicting plane ticket prices. These methods are linear regression, k-th nearest neighbor, random forest regression, and deep learning.

1- Linear Regression

First method applied on the dataset is linear regression. It is one of the most fundamental methods to develop a model that carries the underlying statistical information. Parameters of the n dimensional linear equation that models the data can be found by applying the following formula.

$$\beta_{predicted} = \left(X^T X\right)^{-1} X^T y_{train}$$

This formula can be derived using both statistical methods or linear algebra where its solution minimizes the residual sum of squares between the predicted and real values of the dataset:

$$Cost = \sum_{i=1}^{N} \left(y_{i,predict} - y_{i,true}\right)^2$$

Linear regression has been implemented using linear algebra functionalities of numpy to be able to find the values of $\beta_{predicted}$. Several performance metrics have been deployed to compare this method with other methods used in this project. The performance metrics and their explanations can be seen below

RMSE (Root Mean Squared Error):
This metric explains the average squared error the predictor makes between the real and the predicted test data.

$$RMSE\left(y_{predict}, y_{true}\right) = \sqrt{\frac{\sum_{i=1}^{N}\left(y_{i,predict} - y_{i,true}\right)^2}{N}}$$

R^2 (Coefficient of Determination) Score [3]:
The R-squared value is a measure of the goodness of fit for a model, indicating the proportion of variance in the dependent variable that is predictable from the independent variables.

$$R^2\left(y_{predict}, y_{true}\right) = 1 - \frac{\Sigma\left(y_{test} - y_{predict}\right)^2}{\Sigma\left(y_{test} - \bar{y}_{test}\right)^2}$$

Also the data has been normalized by mapping the values between 0 and 1, this has been done to prevent possible problems which can result from scales of the features. Overall %20 of the data in the dataset has been allocated for only testing and the other %80 has been used for training. After training OLS on the training data, it was applied to the test data and the resulting RMSE($) was found to be 21973.28$ and its R-squared value was 0.061.

The ordinary linear regression model has a drawback since the model has a high tendency to overfit training data. Two common regularization methods (ridge and lasso) have been deployed for an attempt to decrease the model's overfit on the training data. Ridge regression, also known as L2 regularized linear regression, has the following close-form expression as its solution for a centralized dataset.

$$\beta_{predicted} = \left(X^T X + \lambda I\right)^{-1} X^T y_{train}$$

Where the training data is centralizing the data, the lambda parameter has been adjusted using the 10-fold cross-validation. The results for 10-fold cross validation in terms of the average RMSE and R-squared over ten validation sets can be seen in Figures 10 and 11.

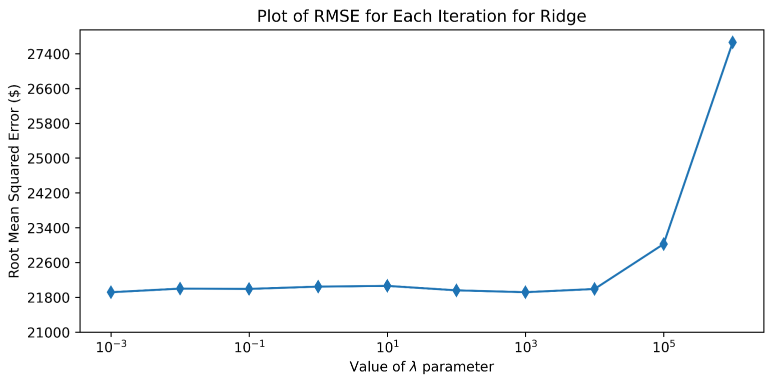


Fig.10 RMSE values for each iteration of 10-fold validation of ridge regression

After completing the training phase the best value for the ridge parameter has selected as 10^3. The resulting test RMSE appeared to be 21961.37$ and the value of R^2 0.0609.

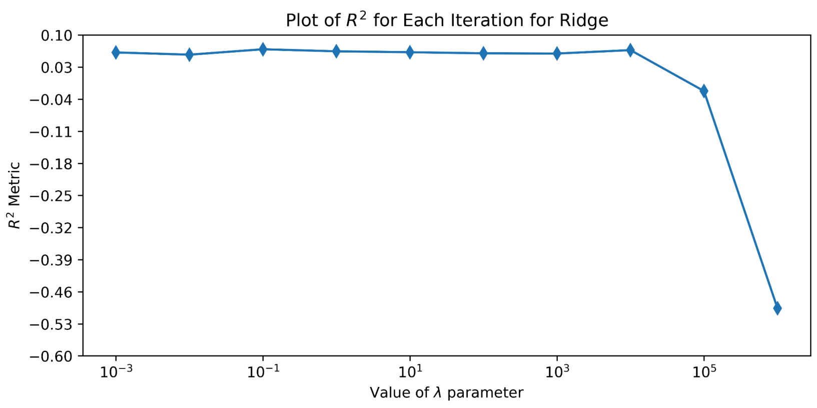


Fig.11 R^2 values for each iteration of 10-fold validation of ridge regression

As can be seen, increasing the lambda value decreased the model's performance. This happened because the linear regression model suffers from underfitting rather than overfitting (caused by large number of data points and small number of predictors) and additional regularization decreases the model's fit further to the given dataset. The minimum value for the average RMSE was seen for λ
After completing the ridge, lasso regression has also been tried to see if any parameter can be eliminated that creates a sparser parameter set. The cost function of the lasso can be seen below:

$$C = \frac{1}{N} \left\| y_{predicted} - y_{train} \right\|^2 + \lambda|\beta|$$

To optimize the parameters using this cost function (λ), an iterative numerical approach has been used. Results of 10-fold cross validation using lasso regression can be seen from Figures 12 and 13.

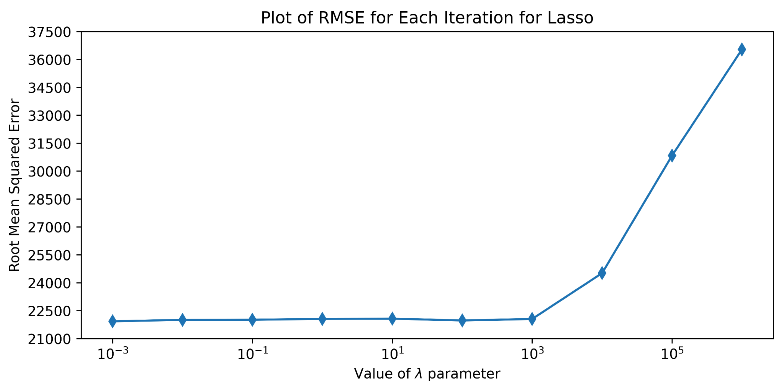


Fig.12 RMSE values for each iteration of 10-fold validation of lasso regression

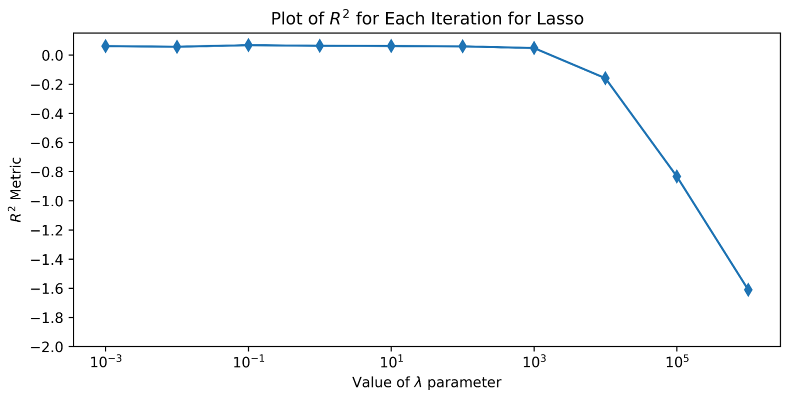


Fig.13 R^2 values for each iteration of 10-fold validation of lasso regression

As can be seen, the underfitting problem occurs when using the lasso regression, too. Therefore, regularization for linear regression is not a strategic application for this dataset. The overall results of using linear regression to normalized data can be seen in Table 1.

	RMSE ($)	R^2
OLS	21973.28	0.061
Ridge Regression	21961.37	0.0609
Lasso Regression	21965.48	0.0605

Table 1. RMSE and R^2 results for the OLS, Ridge and Lasso Regression

2- K-th Nearest Neighbor

K-th Nearest Neighbor (KNN), is a simple method that assigns a prediction to a new data point by finding the nearest k neighboring data points on the training data set in terms of the Euclidean distance to the new point and averaging the response values of these k data points to get a prediction for the new data point's response. To apply this method, the value for k needs to be chosen. One way of choosing this value is to apply cross validation on the data set but since the data set is very large for this project, this cross validation for some range of k values was estimated to last around 48 hours. To save on time, cross-validation was applied to a very small validation set and the result of this test is shown below on Figure 14.

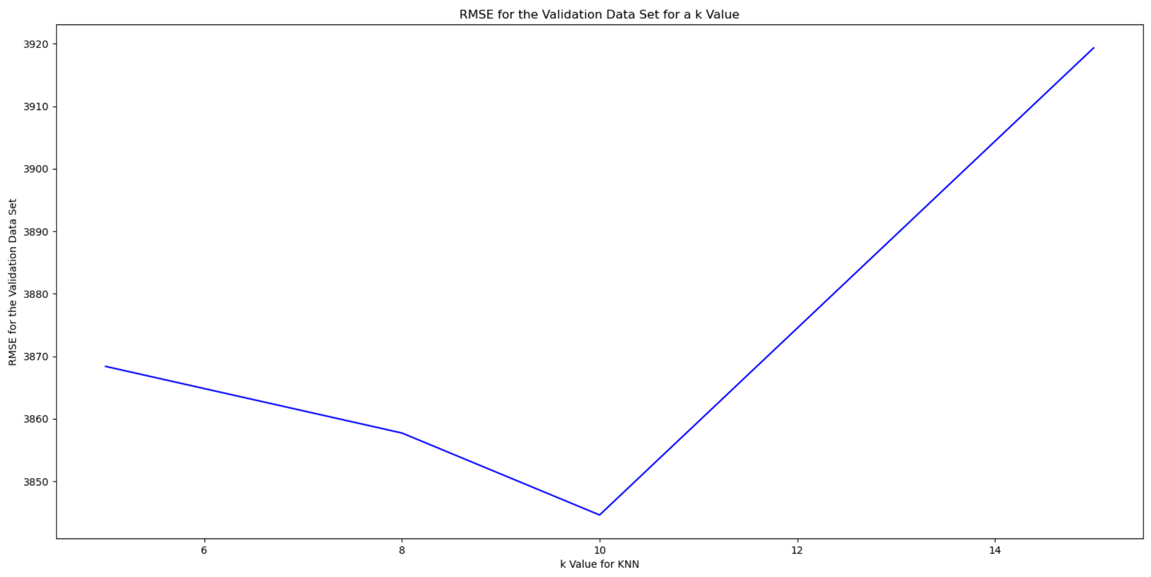


Fig.14 RMSE values for validation set for KNN

Using this figure, k value was chosen to be 10 and the application of KNN on the test data set used in the previous and later methods showed RMSE ($) to be 3780.50.

2- Decision Tree

Decision trees are algorithms based on dividing and segmenting the predictor space into a finite number of regions, each associated with a prediction value. Since the space is split into regions through nodes that seperate points based on the value of a single predictor, the structure ends up resembling a tree constructed from body to branches, the name Decision Tree is given [4]. An example of a decision tree is given in Figure 9.

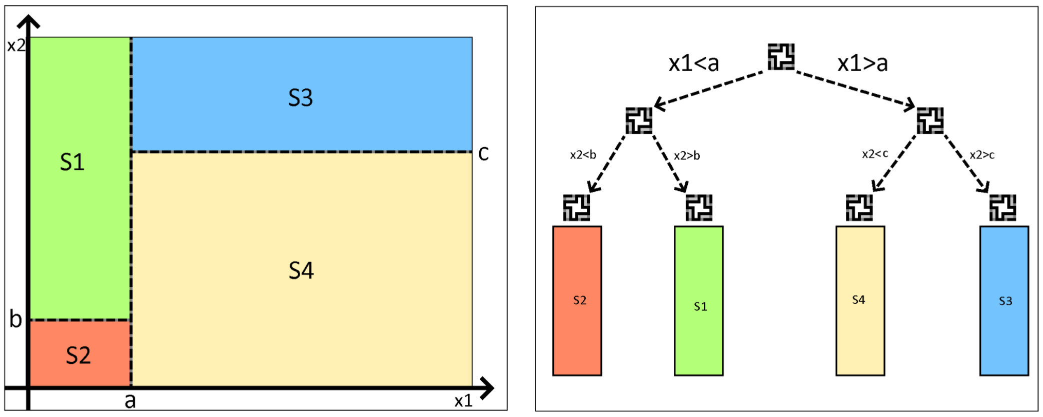


Fig.15 The tree like structure of decision tree algorithm visualized for two predictors and four regions

When new data arrives for the algorithm to predict on, this data is put on one of the regions constructed via the training data by moving through the internal nodes (nodes that are connected to other nodes) based on values for the predictors until a terminal node (a node that has no subsequent node and thus describes a region) is reached. Then, the value associated with that region is used as a prediction.

The reason for using a Decision Tree for this data set is that it is simple to understand how the algorithm reached the conclusion that it did, as the logic behind each choice is explained by the node structure. Along with that, for the plane prices, ticket prices for flights with similar features will be adequately closed since all airlines are in free market competition with each other. This indicates that, once the data is separated into correct regions, the mean of the prices in that region can be a good approximation for the ticket price.

For the implementation presented within this report, the decision tree algorithm aimed to minimize the RSS value for the training data obtained by randomly taking 80% of the data from the original data set and leaving the other 20% for the test data.

$$RSS\left(y_{predict}, y_{true}\right) = \left\|y_{predict} - y_{true}\right\|^2$$

To minimize RSS, the algorithm takes one of the regions that the training data was divided into and searches through all predictors (of which there are 24 after the preprocessing and one hot encoding step). All decision boundaries are considered for one predictor, and the data is split into two regions using this boundary. After reviewing all predictors and all possible decision boundaries in this manner, all ways of splitting the data into two are considered. For each, the RSS for the region is found. The final choice for the decision predictor and the decision boundary is made by minimizing the RSS values. Then, new nodes and a new region are added to the tree structure. This process would continue until each point was in its own region; however, the algorithm would overfit the training data. To ensure overfitting does not occur, the maximum number of regions the tree algorithm will end up with is limited. The minimum number of points that will be put on a single region is also defined to ensure no region focuses on so few points. Maximum number of regions is optimized by comparing different values through the average RMSE for 10 fold cross-validation. Note that 10-fold cross-validation is applied only to the training data to avoid overfitting the test data. The results of this hyperparameter optimization are shown in Figure 16.

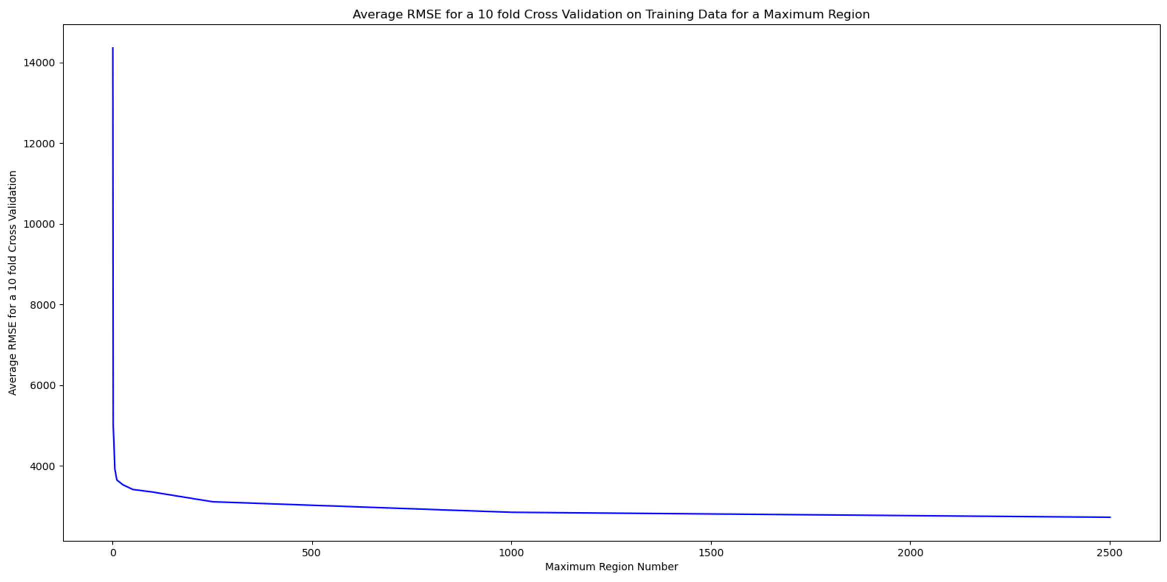


Fig.16 The Average RMSE for 10-fold Cross Validation on test data with respect to number of maximum regions

The average MSE for 10-fold cross-validation drops rapidly for small region counts, whereas it is stable for larger ones. Since the training time increases greatly with the number of regions, a trade-off was decided at 1000 regions to limit training time. Then, applying this value, a tree was trained on the training set as a whole, and the resulting tree structure was used for finding the predictions for the points on the test set. It was seen that the first node in the decision boundary used the class predictor for the split and the data was divided into economy and business classes. As shown before, the class predictor was the one with the most correlation to the ticket price, which means the algorithm is capable of finding the optimal node for the tree. For the test, RMSE($) was found to be 4494.55 $.

Next, to improve the performance of the decision tree, Bootstrap aggregation, or bagging was used to generate new training data sets by taking repeated samples from the original training set. For each of these new data sets, the training algorithm (same one used before with 1000 regions) was applied to find the decision tree. Since each data set is different due to random sampling, the decision trees built on these datasets differ and as a result the variance of the prediction may be lowered. To find a prediction for the test data, the mean of all decision trees constructed on the bagged data was taken. To decide how many data sets to generate, 10-fold cross validation was used for different numbers of trees, the result is shown on Figure 17.

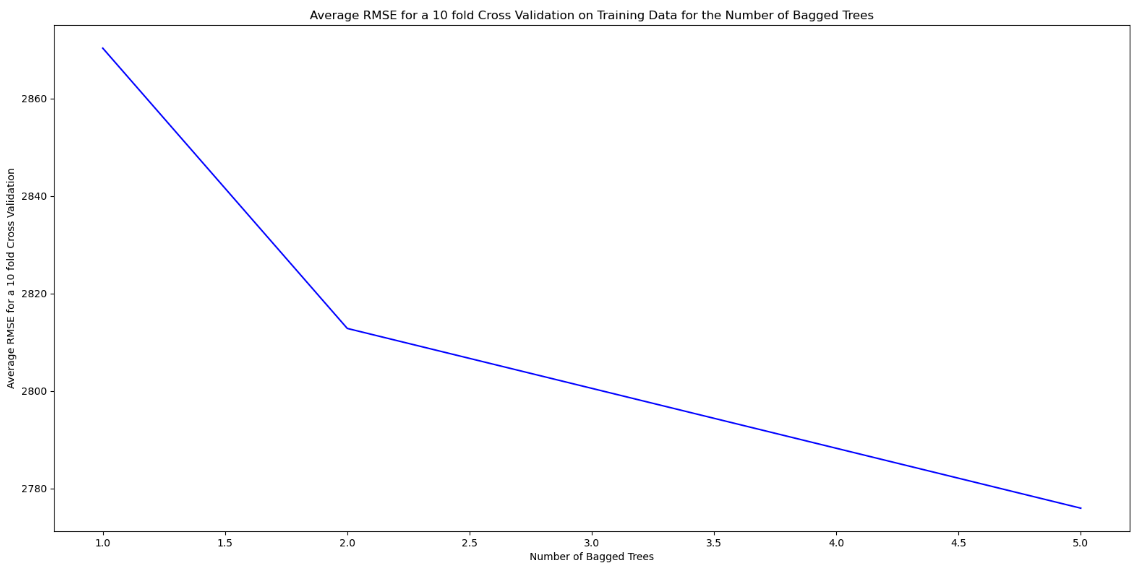


Fig.17 The Average RMSE for 10-fold Cross Validation on test data with respect to number of trees in bagging

As it can be seen, the improvement due to bagging trees is small with respect to the amount of time it takes to perform training. This can be explained by the fact that, since the number of points in the data set is high (>300k) and the number of predictors is low (24), the decision tree model undefits and methods designed to reduce overfitting such as bagging are not very useful. Hence, to manage the training time, the number of bagged trees was chosen to be 5. Using bagging with this number of trees, the RMSE($) on test data was found to be 4341.65 $.

.

To reduce the correlation between the different trees more (in the hopes of reducing the variance of the prediction), random forests method can be applied. For this method, at each node split of the tree only m predictors are considered for a boundary. To test out random forests, the m value was chosen as 5 ≈√24, the resulting RMSE($) on test data for 5 bagged trees with random forests was 22661.89 $. As it can be seen, the RMSE increased greatly when random forests were applied, hence when fewer predictors were considered at each step, the RMSE on the test data increased, and hence for this problem it was better to use all predictors at each of the steps. This can be explained by the fact that the decision trees underfit the dataset because it has low number of predictors and large number of data points, hence random forests, a method used for avoiding overfitting, is not very useful just as ridge and lasso methods were.

3-Neural Network

Neural networks are one of the most popular machine learning techniques in the literature due to their effectiveness in capturing nonlinear relationships in data [4]. A neural network consists of perceptrons which is a unit that takes the weighted summation of its inputs and passes them through a possibly nonlinear activation function f(.). Operation of a perceptron can be seen below:

$$y = f\left(\sum_i x_i w_i + b\right)$$

Perceptrons can be combined in multiple layers to form neural networks which are capable of representing very complex relations based on simple operations on a perceptron. A general structure for the multilayer neural networks are shown on Figure 18.

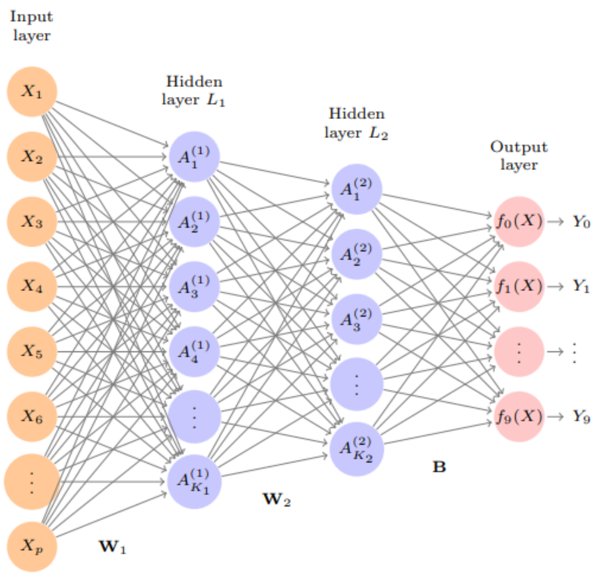


Fig.18 A general structure for the multilayer neural networks [4]

Weights and biases of a neural network can be learned using the training data by an algorithm called gradient descent. Similar to Newton-Raphson method, gradient descent updates the values of weights and biases by shifting them in the space so that the cost function is minimized, i.e gradient of the loss function gets closer to zero. The update values can be updated in the whole network using backpropagation. A general mathematical expression of backpropagation can be seen below:

$$w_{n+1} = w_n - \eta \frac{\partial J}{\partial w_n}$$

To train the data, the loss function can be calculated and used for gradient descent on the entire data set at once or only some portion of the data set can be investigated at one time as well. The method that was chosen for this adaptation was stochastic gradient descent which calculates the loss function on a single data point and updates weights accordingly. The advantage of this method is that there is a probability that the gradient for a single point may not be aligned with the gradient for the whole data set, the result of which is that the algorithm sometimes moves away from the direction that minimizes the loss. Hence, it is possible to escape local minima and land on global minimum with stochastic gradient descent whereas the disadvantage for this method is that the update process for the weights and biases is very noisy since only one out of many data points are considered at each time.

To apply stochastic gradient descent, the training algorithm goes over each data point once in a given training duration called epoch. The number of epochs can be taken as large as one desired with the caveat that an over-trained neural network overfits a given data set. To remedy this problem, early stopping was applied by reserving a portion out of the training data and testing the neural network on this validation data every so often (in every 50 epochs). Once the error on the validation data increases, the training is stopped and the trained network is tested on the test data.

For the first test, a single perceptron was tested out with the relu activation function (activation function used in all networks). Early stopping was triggered at 750 epochs (but for this case, a small change of loss on the validation set was listed as the exit clause to save up on the training time). After 750 epochs, this perceptron was tested on the test data and RMSE($) was found to be 13893.22 $. As it can be seen, the performance of a single perceptron is not as good as the tree methods but it beats the linear regression. This can be expected since the ability of a single perceptron to explain complex relations is very low.

The next step was to form layers and number of perceptrons. First, a network without any hidden layers was tested out. The input layer has 30 perceptrons and they are fully connected to the output layer perceptron. This number of perceptrons was chosen to be as such by observing the validation error for different neural networks with small epoch numbers. A full training could not be completed due to the extensive time required for each network. A possible problem with this method of choice is that a network that performs poorly in the beginning may improve later on but due to lack of time this cannot be uncovered. After deciding on the number of perceptrons, the network was trained with early stopping and the training was stopped at 3500 epochs. The resulting network was applied to the test data and the RMSE($) was found to be 4422.97$.

Then, neural networks with hidden layers were investigated. Once more, to save on the training time, only networks with single layers were tested out. To assess the capability of different networks, an initial training and test on validation data was performed for some setups and the best one to be found was the network with 8 perceptrons in the input layer, 6 perceptrons in the hidden layer and one perceptron as the output layer. After that, early stop training was applied to the chosen network but after 4000 epochs the training was stopped since it had already taken more than 16 hours. The resulting weights were applied to the test data to find RMSE($) as 4636.13 $. Note that validation error was still decreasing when the training was stopped and hence with more training time, the test error may have been decreased more.

Challenges Faced

The biggest challenge that was faced for this project was that due to the large number of data points and the relatively small number of predictors, all methods used so far in the project (linear regression, decision trees and neural networks) always underfit the data set. To solve this problem, the route may have been to use more complex structures in the methods, such as more regions in the decision tree and a more complex network for neural networks. However, such levels could not be attained due to the un-optimized nature of the program, increasing training times beyond justifiable levels. Both the decision tree and the neural network methods were extended to cover this problem and the training time was increased to as high as 16 hours for each method to get over underfitting and these changes have made the results better as shown throughout the report.
Another challenge is to find ways to optimize hyperparameters. As in the case of the maximum region numbers in the decision tree, and the number of perceptrons per layer in the neural network the optimization process takes too long. For this case, some hyper parameter values were tested out for a small amount of time and the promising values were tested more extensively. Applying this method, hyperparameters were chosen in a shorter amount of time with the trade-off that hyperparameters that perform poorly initially and then recover were omitted.

Conclusion

In summary, the RMSE($) on test data for all the methods tried out for the purposes of this project are shown on Table 2.

	RMSE ($) on Test Data
OLS	21973.28
Ridge Regression	21961.37
Lasso Regression	21965.48
KNN for k=10	3780.50

Simple Decision Tree	4494.55
Bagging Decision Tree	4341.66
Random Forest	22661.89
A Single Perceptron	13893.22
Neural Network Without Hidden Layer	4422.97
Neural Network With 1 Hidden Layer	4636.13

Table 2. RMSE result on the test data for all the methods used in the project

As it can be seen, due to the large number of data points and small number of predictors, the methods developed around limiting overfitting such as lasso, ridge and others were not successful in raising the performance of the methods, indicating that underfitting is the main problem with the methods chosen. This problem was lessened by increasing complexity and training time for neural networks and decision trees but could not be eliminated due to extensive training times.

Among the methods tested out for this data set, the worst one was linear regression. None of the modifications made to the regression models were capable of providing a competitive test error with respect to the other methods. This can be explained by the fact that there are non-linear relationships in the data set that the linear regression methods cannot explain which inhibits their performance. Decision trees with bagging on the other hand produced the lowest RMSE on test data bar KNN. This result can be reasoned by the fact that decision trees boil down the data set onto regions that are bounded by a small range for each predictor. Then, in such a region, the tickets sold by the airlines are similar to each other in terms of class, distance, days left till take-off etc. Due to the open market competition, it can be expected that these tickets will have similar prices to each other to stay competitive which can be exploited by the decision trees to predict ticket prices. Similar reasoning can be used to explain the fact that KNN was the most successful method tested out because this method too groups similar tickets together and then estimates the price within the group. The neural networks on the other hand were very competitive as well. Even without a hidden layer, the neural network was only just beaten by the bagged trees. Since neural networks are the most capable method in terms of expressing non-linear and complex relations, they were expected to perform very well. Note that since the training of the hidden layer neural network had taken unreasonably long, the potential performance of the neural networks may have been better than the decision trees when the perceptron numbers and layers were properly adjusted.

Throughout this project, it was observed that many methods could be employed to explain and predict a given data set, and the choice on this matter depends on the specific requirements of the analysis. Linear regression was limited on the performance aspect but it was easy to employ and had high explainability. Decision trees were more difficult to optimize but they still kept some explainability when the tree was not too large yet their performance surpassed linear regression. The neural networks were hard to optimize and their explainability was very low with the performance potential being higher. Thus, among these methods, the final choice should be made by taking all these trade-offs into consideration.

Work Distribution

For this project, the workloads of preprocessing the data- converting it to numbers, one hot encoding, and creating arrays from the data set- and the linear regression part, including the OLS, ridge, and lasso implementation, as well as initial data analysis were attributed to Efe Tarhan. On the other hand, the decision tree implementation along with bagged trees and random forests workload as well as the KNN implementation was given to Kemal Enes Akyüz. The neural network aspect of the project was distributed between both project team members.

References

[1] Bathwal, Shubham. "Flight Price Prediction." Kaggle. Accessed November 17, 2023. [Online]. Available: https://www.kaggle.com/datasets/shubhambathwal/flight-price-prediction.

[2] E. Dougherty, "Deciphering the Digits in Your Flight Number," Blue Sky PIT News Site, 09 Mar, 2020. Accessed: 18 Nov 2023. [Online]. Available: https://blueskypit.com/2020/03/09/deciphering-the-digits-in-your-flight-number/

[3] "Coefficient of Determination (R squared)," Newcastle University, 2023. Accessed: 19 Nov 2023.[Online]. Available: https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/statistics/regression-and-correlation/coefficient-of-determination-r-squared.html.

[4] D. Witten and G. James, An introduction to statistical learning with applications in R. Springer publication, 2013.

Appendix

A) Code for Data Analysis (DataAnalysis.py)

```
# Importing all the Required Libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Displaying the head of the dataset
df = pd.read_csv("/Users/efetarhan/Desktop/DATA/Clean_Dataset.csv")
df = df.drop('Unnamed: 0', axis=1)


#%% Displaying the correlation of the dataset

def preprocs(df):


    #df["stops"] = df["stops"].replace({'zero':0,'one':1,'two_or_more':2}).astype(int)
    #df["class"] = df["class"].replace({'Economy':0,'Business':1}).astype(int)

    stops_mapping = {'zero': 0, 'one': 1, 'two_or_more': 2}
    class_mapping = {'Economy': 0, 'Business': 1}
    df['stops'] = df['stops'].map(stops_mapping).astype(int)
    df['class'] = df['class'].map(class_mapping).astype(int)

    dummies_variables =
["airline","source_city","destination_city","departure_time","arrival_time"]
    dummies = pd.get_dummies(df[dummies_variables], drop_first= True)
    df = pd.concat([df,dummies],axis=1)
    df =
df.drop(["flight","airline","source_city","destination_city","departure_time","arrival_time"],a
xis=1)
    return df

df_preprocessed = preprocs(df)
plt.figure(dpi = 600,figsize=(20,10))
sns.heatmap(df_preprocessed.corr(),annot=True,cmap='Greens')
plt.title("Correlation Map of the Features in the Dataset")
plt.show()


plt.figure(dpi = 600,figsize=(15,5))
sns.heatmap(df.corr(),annot=True,cmap='Greens')
plt.title("Correlation Map of the Numerical Features in the Dataset")
plt.show()


#%% Price range of 6 airlines

plt.figure(figsize=(15,5),dpi = 600)
sns.boxplot(x=df['airline'],y=df['price'],palette='hls')
plt.title('Price Range of 6 Airline Compaines',fontsize=13)
plt.xlabel('Airline',fontsize=16)
plt.ylabel('Price ($)',fontsize=16)
plt.show()


#%% Days Left vs Ticket Price
plt.figure(dpi = 600, figsize=(20,8))
sns.lineplot(data=df,x='days_left',y='price',color='red')
plt.title('Days Left For Departure vs Ticket Price Plot')
plt.xlabel('Days Left for Departure')
plt.ylabel('Price ($)')
plt.show()


#%% Pie chart of portions

df2=df.groupby(['flight','airline','class'],as_index=False).count()
plt.figure(figsize=(10,8),dpi = 600)
```

```
df2['class'].value_counts().plot(kind='pie',autopct='%.2f',cmap='coolwarm',labels=['']*len(df2
['class'].value_counts()))
plt.title('Flight Class Portions in the Dataset')
plt.legend(['Economy','Business'])
plt.show()


#%% Class vs Price Plot

plt.figure(dpi = 600, figsize=(10,5))
sns.violinplot(x='class',y='price',data=df)
plt.title('Class Vs Ticket Price',fontsize=15)
plt.xlabel('Class',fontsize=15)
plt.ylabel('Price($)',fontsize=15)
plt.xticks(ticks=[0, 1], labels=['Economy', 'Business'])
plt.show()

#%% Departure and Arrival Time vs Price Plot

plt.figure(figsize=(24,10))
plt.subplot(1,2,1)
sns.boxplot(x='departure_time',y='price',data=df,palette = 'coolwarm')
plt.title('Departure Time Vs Ticket Price',fontsize=20)
plt.xlabel('Departure Time',fontsize=15)
plt.ylabel('Price($)',fontsize=15)
plt.subplot(1,2,2)
sns.boxplot(x='arrival_time',y='price',data=df,palette='Blues')
plt.title('Arrival Time Vs Ticket Price',fontsize=20)
plt.xlabel('Arrival Time',fontsize=15)
plt.ylabel('Price($)',fontsize=15)
plt.show()


#%% Source City and Destination City vs Price

plt.figure(dpi = 600, figsize=(24,10))
plt.subplot(1,2,1)
sns.boxplot(x='source_city',y='price',data=df)
plt.title('Source City vs Ticket Price',fontsize=20)
plt.xlabel('Source City',fontsize=15)
plt.ylabel('Price',fontsize=15)
plt.subplot(1,2,2)
sns.boxplot(x='destination_city',y='price',data=df,palette='coolwarm_r')
plt.title('Destination City vs Ticket Price',fontsize=20)
plt.xlabel('Destination City',fontsize=15)
plt.ylabel('Price',fontsize=15)
```

```
#%% Days Left for Departure vs the Price Plot
plt.figure(dpi = 600,figsize=(20,8))
sns.lineplot(data=df,x='days_left',y='price',color='blue',hue='airline',palette='hls')
plt.title('Days Left For Departure Versus Ticket Price of Airlines',fontsize=15)
plt.legend()
plt.xlabel('Days Left',fontsize=15)
plt.ylabel('Price($)',fontsize=15)
plt.show()
```

B) Code for Linear Regression Part (Preprocess_LinearRegression_v3.py)

```
#%% IMPORTING NECESSARY LIBRARIES

'''
Importing following libraries for the project:

Pandas: Will be used for extracting data from the csv files and preprocessing

Numpy: Will be used for linear algebra operations

Matplotlib: Will be used for visualizing results
'''

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#%% FUNCTION DEFINITIONS

def split(X, y, test_size=0.2, random_state=None):
    '''
    Splits a given dataset with features and labels into 2 sets with given proportions

    Parameters
    ----------
    X : np.ndarray
        2D numpy array that contains the features of the data
    y : np.ndarray
        1D numpy vector that contains the values of each data point
    test_size : float between 0 and 1
        Proportion of the data that will be assigned to test set. The default is 0.2.
    random_state : integer, optional
        Numpy random seed. The default is None.

    Returns
    -------
    X_train : np.ndarray
        (1-test_size) proportion of the shuffled X matrix .
    X_test : np.ndarray
        test_size proportion of the shuffled X matrix .
    y_train : np.ndarray
        (1-test_size) proportion of the shuffled y vector .
    y_test : np.ndarray
        test_size proportion of the shuffled y vector.

    '''
    if random_state is not None:
        np.random.seed(random_state)
    n_samples = X.shape[0]
    indices = np.arange(n_samples)
    np.random.shuffle(indices)
    test_size = int(n_samples * test_size)
    train_indices = indices[:-test_size]
    test_indices = indices[-test_size:]
    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    return X_train, X_test, y_train, y_test

def centralize(X):
    '''

    Centralizes the features of the matrix X by subtracting mean from each column:

    Parameters
    ----------
    X : np.ndarray
        Feature matrix that will be centralized

    Returns
    -------
    X_c : np.ndarray
        Centralized matrix.

    '''
    X_c = X - np.mean(X,axis = 0)
    return X_c

def standardize(X):
    '''
    Standarizes the matrix X by subtracting the mean and dividng by the standart
    deviation of each column. Gaussian normalization.

    Parameters
    ----------
    X : np.ndarray
        Feature matrix.

    Returns
    -------
    standardized_data : np.ndarray
        Standardized feature matrix.

    '''
    means = np.mean(X, axis=0)
    stds = np.std(X, axis=0)
    standardized_data = (X - means) / stds
    return standardized_data

def normalize(X):
    '''
    Normalizes the feature matrix X by subtracting the minimum value of each
    column and dividng by the difference between maximum and minimum value for
    each column

    Parameters
    ----------
    X : np.ndarray
        Feature matrix.

    Returns
    -------
    X_normalized : np.ndarray
        Normalized feature matrix.

    '''
    min_vals = X.min(axis=0)
    max_vals = X.max(axis=0)
    scale = np.where(max_vals - min_vals != 0, max_vals - min_vals, 1)
    X_normalized = (X - min_vals) / scale
    return X_normalized,min_vals,scale

def fit_ols(X, y):
    '''
    Finds the least squares estimate parameters for a given feature matrix and
    values.

    Parameters
    ----------
    X : np.ndarray
        Feature matrix.
    y : np.ndarray
        Value vector.

    Returns
    -------
    beta : np.ndarray
        Obtained OLS parameters for the given X and y.

    '''

    beta = np.linalg.inv(X.T @ X) @ X.T @ y
    return beta

def fit_ridgels(X,y,lbd):
    '''
    This function finds the estimated parameters for a given lambda value,
    centralized feature matrix and value vector using the ridge regression.

    Parameters
    ----------
    X : np.ndarray
```

```
        Feature matrix.
    y : np.ndarray
        value vector.
    lbd : float
        Regularization factor for ridge regression.

    Returns
    -------
    beta : np.ndarray
        Parameters obtained from ridge regression.

    '''
    beta = np.linalg.inv(X.T @ X + lbd*np.eye(X.shape[1])) @ X.T @ y
    return beta

def fit_lasso(X, y, alpha=1.0, num_iterations=1000, learning_rate=0.0001, tolerance=1e-4):
    '''
    This function finds the estimated parameters for a given lambda value,
    centralized feature matrix and value vector using the lasso regression.

    Parameters
    ----------
    X : np.ndarray
        Feature matrix.
    y : np.ndarray
        value vector.
    alpha : float
        Regularization factor for lasso regression. The default is 1.0.
    num_iterations : integer, optional
        Number of iterations that the regression parameters will be updated. The default is 1000.
    learning_rate : float, optional
        The learning rate for updating the values. The default is 0.0001.
    tolerance : float, optional
        Tolerance value that automatically stops the regression iterations. The default is 1e-4.

    Returns
    -------
    w : np.ndarray
        Parameters obtained from lasso regression.

    '''
    #X = X[:,1:]
    m, n = X.shape
    w = np.zeros((n, 1))
    prev_cost = float('inf')

    for i in range(num_iterations):
        # Compute predictions
        y_pred = np.dot(X, w)

        # Compute cost with L1 regularization (Lasso)
        cost = np.mean((y_pred - y) ** 2) + alpha * np.sum(np.abs(w))

        # Check for convergence
        if np.abs(prev_cost - cost) < tolerance:
            print(f"Convergence reached after {i} iterations.")
            break

        prev_cost = cost

        # Compute gradients
        dw = (1/m) * np.dot(X.T, (y_pred - y)) + alpha * np.sign(w)
        #db = (1/m) * np.sum(y_pred - y)

        # Update weights and bias
        w -= learning_rate * dw
        #b -= learning_rate * db
    return w


def predict(X, beta):
    '''
    Finds the values corresponding to a data (feature) matrix using linear regression
    parameters.

    Parameters
    ----------
    X : np.ndarray
        Feature matrix.
    beta : np.ndarray
        Regression coefficients.

    Returns
    -------
    np.ndarray
        predicted value vector corresponding to X and y.

    '''
    return X @ beta

def mse(y_test,y_pred):
    '''
    Finds the mean squared error between the predicted and real y vectors

    Parameters
    ----------
    y_test : np.ndarray
        Vector that contains the real values.
    y_pred : np.ndarray
        Vector that contains the predicted values.

    Returns
    -------
    MSE : float
        Mean squared error between the y_test and y_pred.

    '''
    MSE = np.mean((y_test - y_pred) ** 2)
    return MSE
```

```
def r2(y_test,y_pred):
    '''
    Coefficient of determination between y_test and y_pred vectors

    Parameters
    ----------
    y_test : np.ndarray
        Vector that contains the real values.
    y_pred : np.ndarray
        Vector that contains the predicted values.

    Returns
    -------
    R2 : float
        R2 score for the vectors y_test and y_pred.

    '''
    R2 = 1 - (np.sum((y_test - y_pred) ** 2) / np.sum((y_test - np.mean(y_test)) ** 2))
    return R2



def data_process(path):
    '''
    This function gets a csv file location and converts the csv file to a feature
    vector using feature conversions to one-hot-encoding or ordinal encoding.

    Parameters
    ----------
    path : string
        Absolute path to the csv file.

    Returns
    -------
    X_all : np.ndarray
        All features data combined in a numpy array with an additional ones for bias parameter.
    X_all_ : np.ndarray
        All features data combined in a numpy array.
    Y_all : np.ndarray
        All values in a numpy vector.
    '''

    data = pd.read_csv(path)
    data.drop(columns=['Unnamed: 0'], inplace=True)
    time_of_day_mapping = {
        'Early_Morning': 1,
        'Morning': 2,
        'Afternoon': 3,
        'Evening': 4,
        'Night': 5,
        'Late_Night': 6
    }
    data['departure_time'] = data['departure_time'].map(time_of_day_mapping)
    data['arrival_time'] = data['arrival_time'].map(time_of_day_mapping)
    # Convert 'stops' to integer
    stops_mapping = {'zero': 0, 'one': 1, 'two_or_more': 2}
    data['stops'] = data['stops'].map(stops_mapping)

    class_mapping = {'Economy' : 0, 'Business' : 1}
    data['class'] = data['class'].map(class_mapping)

    # Drop the original 'flight' column
    data.drop(columns=['flight'], inplace=True)

    airline_map = {'Air_India': np.array([1,0,0,0,0,0]),
           'AirAsia': np.array([0,1,0,0,0,0]),
           'GO_FIRST': np.array([0,0,1,0,0,0]),
           'Indigo': np.array([0,0,0,1,0,0]),
           'SpiceJet': np.array([0,0,0,0,1,0]),
           'Vistara': np.array([0,0,0,0,0,1])}

    airline = np.array(data['airline'].map(airline_map).to_list()).reshape((len(data),6))
    city_map = {'Bangalore': np.array([1,0,0,0,0,0]),
           'Chennai': np.array([0,1,0,0,0,0]),
           'Delhi': np.array([0,0,1,0,0,0]),
           'Hyderabad': np.array([0,0,0,1,0,0]),
           'Mumbai': np.array([0,0,0,0,1,0]),
           'Kolkata': np.array([0,0,0,0,0,1])}

    source_city = np.array(data['source_city'].map(city_map).to_list()).reshape((len(data),6))
    destination_city =
np.array(data['destination_city'].map(city_map).to_list()).reshape((len(data),6))
    departure_time = np.array(data['departure_time'].to_list()).reshape((len(data),1))
    stops = np.array(data['stops'].to_list()).reshape((len(data),1))
    arrival_time = np.array(data['arrival_time'].to_list()).reshape((len(data),1))
    flclass = np.array(data['class'].to_list()).reshape((len(data),1))
    duration = np.array(data['duration'].to_list()).reshape((len(data),1))
    days_left = np.array(data['days_left'].to_list()).reshape((len(data),1))
    price = np.array(data['price'].to_list()).reshape((len(data),1))
    ones = np.ones_like(flclass)

    X_all = np.hstack((ones,airline,source_city,destination_city,
                  departure_time,stops,arrival_time,
                  flclass,duration,days_left)).astype(float)

    X_all_ = X_all[:,1:]
    Y_all = price.astype(float)

    return X_all,X_all_,Y_all

def k_fold_split(X, y, k, shuffle=True, random_state=None):
    '''
    Divides the data into k subpieces that will be used for k iterations of training
    for cross validation or other purposes.

    Parameters
    ----------
    X : np.ndarray
        Feature matrix that will be divided into subpieces after shuffling.
    y : np.ndarray
        Value vector that will be divided into subpieces after shuffling.
    k : int
        number of splits for the cross validation.
    shuffle : bool, optional
        Statement for if the data will be shuffled before training. The default is True.
    random_state : int, optional
        Random seed for initalizing the random module of numpy. The default is None.

    Returns
    -------
    folds : list
        A list that contains a test-train package for each iteration.

    '''

    if random_state is not None:
        np.random.seed(random_state)

    if shuffle:
        indices = np.arange(X.shape[0])
        np.random.shuffle(indices)
        X = X[indices]
        y = y[indices]

    fold_sizes = X.shape[0] // k
    folds = []

    for i in range(k-1):
        test_indices = np.arange(i * fold_sizes, (i + 1) * fold_sizes)
        train_indices = np.setdiff1d(indices, test_indices)

        X_train, y_train = X[train_indices], y[train_indices]
        X_test, y_test = X[test_indices], y[test_indices]

        folds.append((X_train, y_train, X_test, y_test))

    test_indices = np.arange((k-1) * fold_sizes, X.shape[0])
    train_indices = np.setdiff1d(indices, test_indices)

    X_train, y_train = X[train_indices], y[train_indices]
    X_test, y_test = X[test_indices], y[test_indices]
    folds.append((X_train, y_train, X_test, y_test))
    return folds

def train_ols(path):
    '''
    Trains the ordinary least squares algorithm by using a document from a given
    path  using the k-folds cross validation

    Parameters
    ----------
    path : string
        Path to the data csv.
    Returns
    -------
    mse_list : list
        A list that contains the MSE values for each iteration.
    r2_list : list
        A list that contains the R2 scores for each iteration.
    cross_val_beta : np.ndarray
        Mean of the beta values obtained through cross validation process.

    '''

    X_all,X_all_,Y_all = data_process(path)
    X_norm = standardize(X_all_)
    X_train, X_test, y_train, y_test = split(X_norm, Y_all,0.2,42)
    beta = fit_ols(X_train, y_train)
    test_pred = predict(X_test, beta)
    test_rmse = mse(y_test, test_pred)**0.5
    test_r2 = r2(y_test, test_pred)
    return test_rmse,test_r2

def train_ridge(path):
    '''
    Trains the ridge regression algorithm by using a document from a given
    path  using the k-folds cross validation that tries a value of lambda for each
    iteration.

    Parameters
    ----------
    path : string
        Path to the data csv.

    Returns
    -------
    mse_list : list
        A list that contains the MSE values for each iteration.
    r2_list : list
        A list that contains the R2 scores for each iteration.
    cross_val_beta : np.ndarray
        Mean of the beta values obtained through cross validation process.
    lambdas : list
        A list that contains the tried regularization parameters.

    '''

    lambdas = [10.0**(i+2) for i in np.arange(-5,6,1)]
    X_all,X_all_,Y_all = data_process(path)
    rmse_list = []
    r2_list = []
    betas = []
    X_norm = standardize(X_all_)
    #Y_norm = standardize(Y_all)
    X_train, X_test, y_train, y_test = split(X_norm, Y_all,0.2,42)
    folds = k_fold_split(X_train, y_train, 10,True,42)
    cnt = 0
    for (X_train, y_train, X_val, y_val) in folds:
        beta = fit_ridgels(X_train, y_train,lambdas[cnt])
        y_pred = predict(X_val, beta)
        dum_mse = mse(y_val, y_pred)**0.5
        dum_r2 = r2(y_val, y_pred)
        betas.append(beta)
        rmse_list.append(dum_mse)
        r2_list.append(dum_r2)
        cnt += 1
        print("%{} of the process is completed !".format(cnt/len(folds)*100))
    best_beta = betas[rmse_list.index(min(rmse_list))]
    test_pred = predict(X_test, best_beta)
    test_rmse = mse(y_test, test_pred)**0.5
    test_r2 = r2(y_test, test_pred)
    return rmse_list,r2_list,lambdas,test_rmse,test_r2


def train_lasso(path):
    '''
    Trains the lasso regression algorithm by using a document from a given
    path  using the k-folds cross validation that tries a value of lambda for each
    iteration.

    Parameters
    ----------
    path : string
        Path to the data csv.

    Returns
    -------
    mse_list : list
        A list that contains the MSE values for each iteration.
    r2_list : list
        A list that contains the R2 scores for each iteration.
    cross_val_beta : np.ndarray
        Mean of the beta values obtained through cross validation process.
    lambdas : list
        A list that contains the tried regularization parameters.

    '''
    lambdas = [10.0**(i+2) for i in np.arange(-5,6,1)]
    X_all,X_all_,Y_all = data_process(path)
    rmse_list = []
    r2_list = []
    betas = []
    X_norm = standardize(X_all_)
    #Y_norm = standardize(Y_all)
    X_train, X_test, y_train, y_test = split(X_norm, Y_all,0.2,42)
    folds = k_fold_split(X_train, y_train, 10,True,42)
    cnt = 0
    for (X_train, y_train, X_val, y_val) in folds:
        beta = fit_lasso(X_train, y_train,lambdas[cnt],500,learning_rate= 0.01,tolerance = 1e-4)
        y_pred = predict(X_val, beta)
        dum_mse = mse(y_val, y_pred)**0.5
        dum_r2 = r2(y_val, y_pred)
        betas.append(beta)
        rmse_list.append(dum_mse)
        r2_list.append(dum_r2)
        cnt += 1
        print("%{} of the process is completed !".format(cnt/len(folds)*100))
    best_beta = betas[rmse_list.index(min(rmse_list))]
    test_pred = predict(X_test, best_beta)
    test_rmse = mse(y_test, test_pred)**0.5
    test_r2 = r2(y_test, test_pred)
    return rmse_list,r2_list,lambdas,test_rmse,test_r2

#%% Training of the OLS (Ordinary Least Squares)
test_rmse,test_r2 = train_ols("/Users/efetarhan/Desktop/DATA/Clean_Dataset.csv")
print("Test RMSE of Ordinary Least Squares Estimate: {}".format(test_rmse))
print("R^2 of Ordinary Least Squares Estimate: {}".format(test_r2))



#%% Ridge Regression

rmse_list,r2_list,lambdas,test_rmse,test_r2 =
train_ridge("/Users/efetarhan/Desktop/DATA/Clean_Dataset.csv")

plt.figure(figsize = (9,4),dpi = 600)
plt.plot(lambdas[:-1],rmse_list,'-d')
plt.xlabel("Value of $\lambda$ parameter")
plt.ylabel("Root Mean Squared Error ($)")
plt.xscale("log")
plt.title("Plot of RMSE for Each Iteration for Ridge")
plt.xticks([lambdas[2*i] for i in range(len(lambdas)//2)])
plt.yticks(np.arange(21000,28000,800))
plt.show()

plt.figure(figsize = (9,4),dpi = 600)
plt.plot(lambdas[:-1],r2_list,'-d')
plt.xlabel("Value of $\lambda$ parameter")
plt.ylabel("$R^2$ Metric")
plt.xscale("log")
plt.title("Plot of $R^2$ for Each Iteration for Ridge")
plt.xticks([lambdas[2*i] for i in range(len(lambdas)//2)])
plt.yticks(np.arange(-0.6,0.15,0.07))
plt.show()

#%% Lasso Regression

rmse_list,r2_list,lambdas,test_rmse,test_r2 =
train_lasso("/Users/efetarhan/Desktop/DATA/Clean_Dataset.csv")

plt.figure(figsize = (9,4),dpi = 600)
plt.plot(lambdas[:-1],rmse_list,'-d')
plt.xlabel("Value of $\lambda$ parameter")
plt.ylabel("Root Mean Squared Error")
plt.xscale("log")
plt.title("Plot of RMSE for Each Iteration for Lasso")
plt.xticks([lambdas[2*i] for i in range(len(lambdas)//2)])
plt.yticks(np.arange(21000,38000,1500))
plt.show()

plt.figure(figsize = (9,4),dpi = 600)
plt.plot(lambdas[:-1],r2_list,'-d')
plt.xlabel("Value of $\lambda$ parameter")
plt.ylabel("$R^2$ Metric")
plt.xscale("log")
plt.title("Plot of $R^2$ for Each Iteration for Lasso")
plt.xticks([lambdas[2*i] for i in range(len(lambdas)//2)])
plt.yticks(np.arange(-2,0.15,0.2))
plt.show()
```

C) Code for KNN (KNN.py)

```
#%% IMPORTING NECESSARY LIBRARIES
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


#%% FUNCTION DEFINITIONS

def standardize(X):
    means = np.mean(X, axis=0)
    stds = np.std(X, axis=0)
    standardized_data = (X - means) / stds
    return standardized_data

def train_test_split(X, y, test_size=0.2, random_state=None):

    if random_state is not None:
        np.random.seed(random_state)

    n_samples = X.shape[0]

    indices = np.arange(n_samples)
    np.random.shuffle(indices)

    test_size = int(n_samples * test_size)
    train_indices = indices[:-test_size]
    test_indices = indices[-test_size:]

    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    return X_train, X_test, y_train, y_test


#%%

# Load the dataset
data = pd.read_csv('/Users/asus/Desktop/485P/Clean_Dataset.csv')
# Remove the 'Unnamed: 0' column
data.drop(columns=['Unnamed: 0'], inplace=True)
# Custom mapping for 'departure_time' and 'arrival_time'
```

```
time_of_day_mapping = {
    'Early_Morning': 1,
    'Morning': 2,
    'Afternoon': 3,
    'Evening': 4,
    'Night': 5,
    'Late_Night': 6
}
data['departure_time'] = data['departure_time'].map(time_of_day_mapping)
data['arrival_time'] = data['arrival_time'].map(time_of_day_mapping)

# Convert 'stops' to integer
stops_mapping = {'zero': 0, 'one': 1, 'two_or_more': 2}
data['stops'] = data['stops'].map(stops_mapping)

class_mapping = {'Economy' : 0, 'Business' : 1}
data['class'] = data['class'].map(class_mapping)

# Extract the numerical part of the 'flight' feature
flight_number = data['flight'].str.extract('(\d+)').astype(int)

# Find the index of the 'flight' column
flight_col_index = data.columns.get_loc('flight')

# Insert the 'flight_number' column in the place of 'flight'
data.insert(flight_col_index, 'flight_number', flight_number)

# Drop the original 'flight' column
data.drop(columns=['flight'], inplace=True)
#%%
airline_map = {'Air_India': np.array([1,0,0,0,0,0]),
        'AirAsia': np.array([0,1,0,0,0,0]),
        'GO_FIRST': np.array([0,0,1,0,0,0]),
        'Indigo': np.array([0,0,0,1,0,0]),
        'SpiceJet': np.array([0,0,0,0,1,0]),
        'Vistara': np.array([0,0,0,0,0,1])}

airline = np.array(data['airline'].map(airline_map).to_list()).reshape((len(data),6))
#%%
city_map = {'Bangalore': np.array([1,0,0,0,0,0]),
        'Chennai': np.array([0,1,0,0,0,0]),
        'Delhi': np.array([0,0,1,0,0,0]),
        'Hyderabad': np.array([0,0,0,1,0,0]),
        'Mumbai': np.array([0,0,0,0,1,0]),
        'Kolkata': np.array([0,0,0,0,0,1])}

source_city = np.array(data['source_city'].map(city_map).to_list()).reshape((len(data),6))
destination_city =
np.array(data['destination_city'].map(city_map).to_list()).reshape((len(data),6))
#%%
departure_time = np.array(data['departure_time'].to_list()).reshape((len(data),1))
stops = np.array(data['stops'].to_list()).reshape((len(data),1))
arrival_time = np.array(data['arrival_time'].to_list()).reshape((len(data),1))
flclass = np.array(data['class'].to_list()).reshape((len(data),1))
duration = np.array(data['duration'].to_list()).reshape((len(data),1))
```

```
days_left = np.array(data['days_left'].to_list()).reshape((len(data),1))
price = np.array(data['price'].to_list()).reshape((len(data),1))
flight = np.array(data['flight_number'].to_list()).reshape((len(data),1))

# The variable used for indicating to which tree region one variable belongs
# to (all data points are initially grouped into the region 0)
tree_regions = np.zeros_like(flclass)
#%%
X_all = np.hstack((standardize(airline),standardize(source_city),
                standardize(destination_city),
                standardize(departure_time),standardize(stops),
                standardize(arrival_time),
                standardize(flclass),standardize(duration),
                standardize(days_left))).astype(float)

y_all = price.astype(float)


#%%

# Performing a test-train split
X_train, X_test, y_train, y_test = train_test_split(X_all,y_all,0.2,42)


# Variable used for keeping record of the cumulative rss
RSS = 0

# How many neighbors will be involved in the KNN algorithm
k = 10

# For each point in the data set
for n in range(X_test.shape[0]):

    # Predicting a result using KNN by first finding the Euclidian distances
    # between the tes point and any of the data points in the training set
    # Note that norm of the differences vector can be used for the Euclidian
    # distances

    Euclidian_distances = np.linalg.norm(X_train - X_test[n], axis=1)

    # Now, we'll find the minimums of this distances array as follows.
    # This function returns the indices of the minimum k distance values
    minimums = np.argsort(Euclidian_distances)[0:k]

    # Now, the nearest k points will ve used for predicitng the price of the
    # data point by taking the mean of the k points
    y_prediction = np.mean(y_train[minimums])

    # Addition to the RSS for a single data point
    RSS += (y_prediction-y_test[n])**2

    print("{PR:.2f}%".format(PR = n/X_test.shape[0]*100))

# Computing the MSE for the average predictions and real price values on the
# test data set
```

```
        MSE_test = RSS/y_test.shape[0]
        RMSE_n = np.sqrt(MSE_test)
        print("RMSE on test data (for k = {knn}) = {r}".format(knn = k, r=RMSE_n))
```

D) Code for Decision Tree (Decision_Tree_Test_Final.py)

```
# -*- coding: utf-8 -*-


#%% IMPORTING NECESSARY LIBRARIES
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#%% FUNCTION DEFINITIONS

def standardize(X):
    means = np.mean(X, axis=0)
    stds = np.std(X, axis=0)
    standardized_data = (X - means) / stds
    return standardized_data

def train_test_split(X, y, test_size=0.2, random_state=None):

    if random_state is not None:
        np.random.seed(random_state)

    n_samples = X.shape[0]

    indices = np.arange(n_samples)
    np.random.shuffle(indices)

    test_size = int(n_samples * test_size)
    train_indices = indices[:-test_size]
    test_indices = indices[-test_size:]

    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    return X_train, X_test, y_train, y_test

def k_fold_split(X, y, k, shuffle=True, random_state=None):

    if random_state is not None:
        np.random.seed(random_state)

    if shuffle:
        indices = np.arange(X.shape[0])
        np.random.shuffle(indices)
        X_temp = X[indices]
        y_temp = y[indices]

    fold_sizes = X.shape[0] // k
    folds = []

    for i in range(k-1):
        test_indices = np.arange(i * fold_sizes, (i + 1) * fold_sizes)
        train_indices = np.setdiff1d(indices, test_indices)

        X_train, y_train = X_temp[train_indices], y_temp[train_indices]
        X_test, y_test = X_temp[test_indices], y_temp[test_indices]

        folds.append((X_train, y_train, X_test, y_test))

    test_indices = np.arange((k-1) * fold_sizes, X.shape[0])
    train_indices = np.setdiff1d(indices, test_indices)

    X_train, y_train = X_temp[train_indices], y_temp[train_indices]
    X_test, y_test = X_temp[test_indices], y_temp[test_indices]

    folds.append((X_train, y_train, X_test, y_test))
    return folds

def find_RSS(a,b):
    RSS = np.sum((a - b) ** 2)
    return RSS

def find_RSS_1(a,b):
    RSS = np.sum(abs(a - b))
    return RSS

def find_R_values(X,y):
    # Creating an array that'll contain the mean for every region created by
    # the tree algorithm, this array is used for estimating values for new data
    # (The size of the array is the same as the number of regions in the tree)
    R_values = np.zeros(int(np.max(X[:,-1]))+1)

    # For each region in the tree (region number is in -1th column)
    for i in range (0,int(np.max(X[:,-1]))+1):
        # Finding the points that are all in the region i
        R_indices = np.argwhere(X[:,-1]==i)
        # Noting the means of the points on the region i
        R_values[i] = np.mean(y[R_indices])
    return R_values

def choose_predictor(X_reg,y_reg,Dec_Bou):
    # This function chooses among which of the predictors will be the best to
    # split the data from and which value is the best decision boundary

    # This array will hold the minimum RSS value [0] for each predictor and the
    # decision boundary it denotes [1]
    RSS_pre = np.zeros((X_reg.shape[1]-1,2))
    # For each predictor (note that last column is used for tree regions)
    for pre in range(0,X_reg.shape[1]-1):
        # This variable will hold the minimum RSS value for the predictor
        min_RSS = 10**50
        # This variable will denote the decision boundary for minimum RSS
        min_val = 0
        for bou in Dec_Bou[pre]:
            # Finding the data points under the decision boundary
            down_reg = np.squeeze(y_reg[np.argwhere(X_reg[:,pre]<bou)])
            # Finding RSS for the points under the decision boundary
            RSS_down = np.linalg.norm(down_reg-np.mean(down_reg))**2
            # Finding the data points over the decision boundary
            up_reg = np.squeeze(y_reg[np.argwhere(X_reg[:,pre]>=bou)])
            # Finding RSS for the points over the decision boundary
            RSS_up = np.linalg.norm(up_reg-np.mean(up_reg))**2
            if ((RSS_down + RSS_up)<min_RSS):
                min_RSS = RSS_down + RSS_up
                min_val = bou
        # This is the best decision boundary for the given predictor
        RSS_pre[pre,0] = min_RSS
        RSS_pre[pre,1] = min_val

    cho_pre = np.argmin(RSS_pre[:,0])
    pre_value = RSS_pre[cho_pre,1]

    return cho_pre, pre_value

def create_decision_boundaries(X_sorted):
    # This function creates possible decision boundaries based on the sorted
    # data set

    # Creating the dictionary that will contain the outputs
    Dec_Bou = dict.fromkeys((range(X_sorted.shape[1]-1)),[])

    # For each predictor, finding the unique values for that predictor
    for pre in range(0,X_sorted.shape[1]-1):
        Dec_Bou[pre]=(np.unique(X_sorted[:,pre]))

    return Dec_Bou

def Decision_Tree_Constructer(X_train,y_train,max_reg,min_points):
    # max_reg : maximum number of regions to be created by the tree branches
    # min_points : number of points after which a region will not be seperated
    X = X_train
    y = y_train

    # Creating the dictionary that will contain all the node relations by
    # summining the next two nodes to follow if it is not a terminal node and
    # the number of the region that it flows to if it is a terminal node
    Tree_Nodes = {0: [True,0]}
    # For terminal nodes:
    # {i: [Is end node = True, number of the region it denotes]}
    # For internal nodes:
    # {j : [Is terminal node = False, predictor it splits, value of split,
    #       node 1 (precitor < value), node 2 (precitor >= value)]}

    # Creating an array contaning all the values of the predictors in ascending
    # order (this array will be useful for the tree algorithm where the jumbled up
    # data order does not matter)
    X_sorted = np.zeros_like(X)
    for pre in range(0,X.shape[1]-1):
        X_sorted[:,pre] = np.sort(X[:,pre])

    # Creating the array that will contain the would be decision boundaries
    # for each predictor
    Dec_Bou = create_decision_boundaries(X_sorted)

    # The variable used for keeping track of number of regions
    num_reg = 0
    # The variable used for keeping track of number of nodes
    num_nodes = 0

    while(True):

        # For each node in the tree
        for node in list(Tree_Nodes):
            # Checking whether the max region limit is reached
            if num_reg >= max_reg:
                # If maximum region number is reach the tree construction ends
                break
            # If it is a terminal node
            if (Tree_Nodes[node][0] == True):
                # Region associated with the chosen node
                node_reg =  Tree_Nodes[node][1]
                # Getting all the data in the terminal node region
                X_reg = np.squeeze(X[np.argwhere(X[:,-1]==node_reg),:])
                y_reg = np.squeeze(y[np.argwhere(X[:,-1]==node_reg)])
                # If there are already less than the minimum amount of points
                # in a region, it will not be further seperated
                if (len(X_reg.shape)==2):
                # Choosing a predictor to split and where to split it
                cho_pre, pre_value = choose_predictor(X_reg,y_reg,Dec_Bou)
                # Creating two new terminal nodes for the split
                Tree_Nodes[num_nodes+1] = [True,node_reg]
                X[np.argwhere((X[:,cho_pre]<pre_value) & (X[:,-1]==node_reg)),-1] = node_reg
                Tree_Nodes[num_nodes+2] = [True,num_reg+1]
                X[np.argwhere((X[:,cho_pre]>=pre_value) & (X[:,-1]==node_reg)),-1] = num_reg+1
                # Altering the original node because it is no longer terminal
                Tree_Nodes[node] = [False,cho_pre,pre_value,num_nodes+1,num_nodes+2]
                num_nodes += 2
                num_reg += 1

        if num_reg >= max_reg:
            break
    return Tree_Nodes, X

def predict(X,Tree_Nodes,R_values):
    # This function predicts the price values for a given preddictor set based
    # on the region in which it falls on the tree structure and the mean at
    # that region

    # Temporary variable used for keeping the nodes to which points are diverted
    X_nodes = np.zeros(X.shape[0])

    # The variable that'll hold the predicted value of price
    y_val_pre = np.zeros(X.shape[0])

    # For each node
    for node in list(Tree_Nodes):
        # If the node is not a terminal node, the points are sent to the
        # following nodes through the structure of the tree
        if Tree_Nodes[node][0] == False:
            X_nodes[np.argwhere((X_nodes[:] == node) &
(X[:,Tree_Nodes[node][1]]<Tree_Nodes[node][2]))] = Tree_Nodes[node][3]
            X_nodes[np.argwhere((X_nodes[:] == node) &
(X[:,Tree_Nodes[node][1]]>=Tree_Nodes[node][2]))] = Tree_Nodes[node][4]
        # If the node is a terminal node, the points are used to label regions
        if Tree_Nodes[node][0] == True:
            try:
                y_val_pre[np.argwhere(X_nodes[:]==node)] = R_values[Tree_Nodes[node][1]]
            except:
                print()
    return y_val_pre

#%%

# Load the dataset
data = pd.read_csv('/Users/asus/Desktop/485P/Clean_Dataset.csv')
```

```
# Remove the 'Unnamed: 0' column
data.drop(columns=['Unnamed: 0'], inplace=True)

# Custom mapping for 'departure_time' and 'arrival_time'
time_of_day_mapping = {
   'Early_Morning': 1,
   'Morning': 2,
   'Afternoon': 3,
   'Evening': 4,
   'Night': 5,
   'Late_Night': 6
}
data['departure_time'] = data['departure_time'].map(time_of_day_mapping)
data['arrival_time'] = data['arrival_time'].map(time_of_day_mapping)

# Convert 'stops' to integer
stops_mapping = {'zero': 0, 'one': 1, 'two_or_more': 2}
data['stops'] = data['stops'].map(stops_mapping)

class_mapping = {'Economy' : 0, 'Business' : 1}
data['class'] = data['class'].map(class_mapping)

# Extract the numerical part of the 'flight' feature
flight_number = data['flight'].str.extract('(\d+)').astype(int)

# Find the index of the 'flight' column
flight_col_index = data.columns.get_loc('flight')

# Insert the 'flight_number' column in the place of 'flight'
data.insert(flight_col_index, 'flight_number', flight_number)

# Drop the original 'flight' column
data.drop(columns=['flight'], inplace=True)
#%%
airline_map = {'Air_India': np.array([1,0,0,0,0,0]),
       'AirAsia': np.array([0,1,0,0,0,0]),
       'GO_FIRST': np.array([0,0,1,0,0,0]),
       'Indigo': np.array([0,0,0,1,0,0]),
       'SpiceJet': np.array([0,0,0,0,1,0]),
       'Vistara': np.array([0,0,0,0,0,1])}

airline = np.array(data['airline'].map(airline_map).to_list()).reshape((len(data),6))
#%%
city_map = {'Bangalore': np.array([1,0,0,0,0,0]),
       'Chennai': np.array([0,1,0,0,0,0]),
       'Delhi': np.array([0,0,1,0,0,0]),
       'Hyderabad': np.array([0,0,0,1,0,0]),
       'Mumbai': np.array([0,0,0,0,1,0]),
    'Kolkata': np.array([0,0,0,0,0,1])}

source_city = np.array(data['source_city'].map(city_map).to_list()).reshape((len(data),6))
destination_city = np.array(data['destination_city'].map(city_map).to_list()).reshape((len(data),6))
#%%
departure_time = np.array(data['departure_time'].to_list()).reshape((len(data),1))
stops = np.array(data['stops'].to_list()).reshape((len(data),1))
arrival_time = np.array(data['arrival_time'].to_list()).reshape((len(data),1))
flclass = np.array(data['class'].to_list()).reshape((len(data),1))
duration = np.array(data['duration'].to_list()).reshape((len(data),1))
days_left = np.array(data['days_left'].to_list()).reshape((len(data),1))
price = np.array(data['price'].to_list()).reshape((len(data),1))
flight = np.array(data['flight_number'].to_list()).reshape((len(data),1))

# The variable used for indicating to which tree region one variable belongs
# to (all data points are initially grouped into the region 0)
tree_regions = np.zeros_like(flclass)
#%%
X_all = np.hstack((airline,source_city,destination_city,
                departure_time,stops,arrival_time,
                flclass,duration,days_left,tree_regions)).astype(float)

y_all = price.astype(float)

# Standardizing the predictors and the response
# X_all[:,0:-1] = standardize(X_all[:,0:-1])
# y_all = standardize(y_all)




# Performing a test-train split
X_tr, X_te, y_tr, y_te = train_test_split(X_all,y_all,0.2,42)

max_reg = 1000
min_points = 100

RMSE_cv = 0

# Creating the tree structure based on minimizing RSS, the algorithm will
# attach an integer region data to all of the data points in the training data
Tree_Nodes, X_train = Decision_Tree_Constructer(X_tr,y_tr,max_reg,min_points)

# Finding the mean for each region by using the price values associated with
# the points in that region
R_values = find_R_values(X_tr,y_tr)

# Assigning region values to the points in the validation data set based on
# the nodes designated by the tree algorithm and using the mean values inside
# the regions to predict the points in the validation set
y_te_pre = predict(X_te,Tree_Nodes,R_values)

# Computing the MSE for the predictions and real price values on the valida-
# tion data set
MSE_te = find_RSS(np.squeeze(y_te),np.squeeze(y_te_pre))/y_te.shape[0]
RMSE = np.sqrt(MSE_te)

print("RMSE for the test data \
    (using {mr} max regions): \n".format(mr = max_reg+1))
print(RMSE)
```

E) Code for Bagged Trees (Bagging_Test_Final.py)

```
# -*- coding: utf-8 -*-


#%% IMPORTING NECESSARY LIBRARIES
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#%% FUNCTION DEFINITIONS

def standardize(X):
    means = np.mean(X, axis=0)
    stds = np.std(X, axis=0)
    standardized_data = (X - means) / stds
    return standardized_data

def train_test_split(X, y, test_size=0.2, random_state=None):

    if random_state is not None:
        np.random.seed(random_state)

    n_samples = X.shape[0]

    indices = np.arange(n_samples)
    np.random.shuffle(indices)

    test_size = int(n_samples * test_size)
    train_indices = indices[:-test_size]
    test_indices = indices[-test_size:]

    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    return X_train, X_test, y_train, y_test
```

```
def k_fold_split(X, y, k, shuffle=True, random_state=None):

    if random_state is not None:
        np.random.seed(random_state)

    if shuffle:
        indices = np.arange(X.shape[0])
        np.random.shuffle(indices)
        X_temp = X[indices]
        y_temp = y[indices]

    fold_sizes = X.shape[0] // k
    folds = []

    for i in range(k-1):
        test_indices = np.arange(i * fold_sizes, (i + 1) * fold_sizes)
        train_indices = np.setdiff1d(indices, test_indices)

        X_train, y_train = X_temp[train_indices], y_temp[train_indices]
        X_test, y_test = X_temp[test_indices], y_temp[test_indices]

        folds.append((X_train, y_train, X_test, y_test))

    test_indices = np.arange((k-1) * fold_sizes, X.shape[0])
    train_indices = np.setdiff1d(indices, test_indices)

    X_train, y_train = X_temp[train_indices], y_temp[train_indices]
    X_test, y_test = X_temp[test_indices], y_temp[test_indices]

    folds.append((X_train, y_train, X_test, y_test))
    return folds

def find_RSS(a,b):
    RSS = np.sum((a - b) ** 2)
    return RSS

def find_RSS_1(a,b):
    RSS = np.sum(abs(a - b))
    return RSS

def find_R_values(X,y):
    # Creating an array that'll contain the mean for every region created by
    # the tree algorithm, this array is used for estimating values for new data
    # (The size of the array is the same as the number of regions in the tree)
    R_values = np.zeros(int(np.max(X[:,-1]))+1)

    # For each region in the tree (region number is in -1th column)
    for i in range (0,int(np.max(X[:,-1]))+1):
        # Finding the points that are all in the region i
        R_indices = np.argwhere(X[:,-1]==i)
        # Noting the means of the points on the region i
        R_values[i] = np.mean(y[R_indices])
    return R_values

def choose_predictor(X_reg,y_reg,Dec_Bou):
    # This function chooses among which of the predictors will be the best to
    # split the data from and which value is the best decision boundary

    # This array will hold the minimum RSS value [0] for each predictor and the
    # decision boundary it denotes [1]
    RSS_pre = np.zeros((X_reg.shape[1]-1,2))
    # For each predictor (note that last column is used for tree regions)
    for pre in range(0,X_reg.shape[1]-1):
        # This variable will hold the minimum RSS value for the predictor
        min_RSS = 10**50
        # This variable will denote the decision boundary for minimum RSS
        min_val = 0
        for bou in Dec_Bou[pre]:
            # Finding the data points under the decision boundary
            down_reg = np.squeeze(y_reg[np.argwhere(X_reg[:,pre]<bou)])
            # Finding RSS for the points under the decision boundary
            RSS_down = np.linalg.norm(down_reg-np.mean(down_reg))**2
            # Finding the data points over the decision boundary
            up_reg = np.squeeze(y_reg[np.argwhere(X_reg[:,pre]>=bou)])
            # Finding RSS for the points over the decision boundary
            RSS_up = np.linalg.norm(up_reg-np.mean(up_reg))**2
            if ((RSS_down + RSS_up)<min_RSS):
                min_RSS = RSS_down + RSS_up
                min_val = bou
        # This is the best decision boundary for the given predictor
        RSS_pre[pre,0] = min_RSS
        RSS_pre[pre,1] = min_val

    cho_pre = np.argmin(RSS_pre[:,0])
    pre_value = RSS_pre[cho_pre,1]

    return cho_pre, pre_value

def create_decision_boundaries(X_sorted):
    # This function creates possible decision boundaries based on the sorted
    # data set

    # Creating the dictionary that will contain the outputs
    Dec_Bou = dict.fromkeys((range(X_sorted.shape[1]-1)),[])

    # For each predictor, finding the unique values for that predictor
    for pre in range(0,X_sorted.shape[1]-1):
        Dec_Bou[pre]=(np.unique(X_sorted[:,pre]))

    return Dec_Bou

def Decision_Tree_Constructer(X_train,y_train,max_reg,min_points):
    # max_reg : maximum number of regions to be created by the tree branches
    # min_points : number of points after which a region will not be seperated
    X = X_train
    y = y_train

    # Creating the dictionary that will contain all the node relations by
    # summining the next two nodes to follow if it is not a terminal node and
    # the number of the region that it flows to if it is a terminal node
    Tree_Nodes = {0: [True,0]}
    # For terminal nodes:
    # {i: [Is end node = True, number of the region it denotes]}
    # For internal nodes:
    # {j : [Is terminal node = False, predictor it splits, value of split,
    #       node 1 (precitor < value), node 2 (precitor >= value)]}

    # Creating an array contaning all the values of the predictors in ascending
    # order (this array will be useful for the tree algorithm where the jumbled up
    # data order does not matter)
    X_sorted = np.zeros_like(X)
    for pre in range(0,X.shape[1]-1):
        X_sorted[:,pre] = np.sort(X[:,pre])

    # Creating the array that will contain the would be decision boundaries
    # for each predictor
    Dec_Bou = create_decision_boundaries(X_sorted)

    # The variable used for keeping track of number of regions
    num_reg = 0
    # The variable used for keeping track of number of nodes
    num_nodes = 0

    while(True):

        # For each node in the tree
        for node in list(Tree_Nodes):
            # Checking whether the max region limit is reached
            if num_reg >= max_reg:
                # If maximum region number is reach the tree construction ends
                break
            # If it is a terminal node
            if (Tree_Nodes[node][0] == True):
                # Region associated with the chosen node
                node_reg =  Tree_Nodes[node][1]
                # Getting all the data in the terminal node region
                X_reg = np.squeeze(X[np.argwhere(X[:,-1]==node_reg),:])
                y_reg = np.squeeze(y[np.argwhere(X[:,-1]==node_reg)])
                # If there are already less than the minimum amount of points
                # in a region, it will not be further seperated
                if (len(X_reg.shape)==2):
                    # Choosing a predictor to split and where to split it
                    cho_pre, pre_value = choose_predictor(X_reg,y_reg,Dec_Bou)
                    # Creating two new terminal nodes for the split
                    Tree_Nodes[num_nodes+1] = [True,node_reg]
                    X[np.argwhere((X[:,cho_pre]<pre_value) & (X[:,-1]==node_reg)),-1] = node_reg
                    Tree_Nodes[num_nodes+2] = [True,num_reg+1]
                    X[np.argwhere((X[:,cho_pre]>=pre_value) & (X[:,-1]==node_reg)),-1] = num_reg+1
                    # Altering the original node because it is no longer terminal
                    Tree_Nodes[node] = [False,cho_pre,pre_value,num_nodes+1,num_nodes+2]
                    num_nodes += 2
                    num_reg += 1

        if num_reg >= max_reg:
            break
    return Tree_Nodes, X

def predict(X,Tree_Nodes,R_values):
    # This function predicts the price values for a given preddictor set based
    # on the region in which it falls on the tree structure and the mean at
    # that region

    # Temporary variable used for keeping the nodes to which points are diverted
    X_nodes = np.zeros(X.shape[0])

    # The variable that'll hold the predicted value of price
    y_val_pre = np.zeros(X.shape[0])

    # For each node
    for node in list(Tree_Nodes):
        # If the node is not a terminal node, the points are sent to the
        # following nodes through the structure of the tree
        if Tree_Nodes[node][0] == False:
            X_nodes[np.argwhere((X_nodes[:] == node) &
(X[:,Tree_Nodes[node][1]]<Tree_Nodes[node][2]))] = Tree_Nodes[node][3]
            X_nodes[np.argwhere((X_nodes[:] == node) &
(X[:,Tree_Nodes[node][1]]>=Tree_Nodes[node][2]))] = Tree_Nodes[node][4]
        # If the node is a terminal node, the points are used to label regions
        if Tree_Nodes[node][0] == True:
            try:
                y_val_pre[np.argwhere(X_nodes[:]==node)] = R_values[Tree_Nodes[node][1]]
            except:
```

```
            print()
    return y_val_pre

#%%

# Load the dataset
data = pd.read_csv('/Users/asus/Desktop/485P/Clean_Dataset.csv')
# Remove the 'Unnamed: 0' column
data.drop(columns=['Unnamed: 0'], inplace=True)
# Custom mapping for 'departure_time' and 'arrival_time'
time_of_day_mapping = {
    'Early_Morning': 1,
    'Morning': 2,
    'Afternoon': 3,
    'Evening': 4,
    'Night': 5,
    'Late_Night': 6
}
data['departure_time'] = data['departure_time'].map(time_of_day_mapping)
data['arrival_time'] = data['arrival_time'].map(time_of_day_mapping)

# Convert 'stops' to integer
stops_mapping = {'zero': 0, 'one': 1, 'two_or_more': 2}
data['stops'] = data['stops'].map(stops_mapping)

class_mapping = {'Economy' : 0, 'Business' : 1}
data['class'] = data['class'].map(class_mapping)

# Extract the numerical part of the 'flight' feature
flight_number = data['flight'].str.extract('(\d+)').astype(int)

# Find the index of the 'flight' column
flight_col_index = data.columns.get_loc('flight')

# Insert the 'flight_number' column in the place of 'flight'
data.insert(flight_col_index, 'flight_number', flight_number)

# Drop the original 'flight' column
data.drop(columns=['flight'], inplace=True)
#%%
airline_map = {'Air_India': np.array([1,0,0,0,0,0]),
        'AirAsia': np.array([0,1,0,0,0,0]),
        'GO_FIRST': np.array([0,0,1,0,0,0]),
        'Indigo': np.array([0,0,0,1,0,0]),
        'SpiceJet': np.array([0,0,0,0,1,0]),
        'Vistara': np.array([0,0,0,0,0,1])}

airline = np.array(data['airline'].map(airline_map).to_list()).reshape((len(data),6))
```

```
#%%
city_map = {'Bangalore': np.array([1,0,0,0,0,0]),
        'Chennai': np.array([0,1,0,0,0,0]),
        'Delhi': np.array([0,0,1,0,0,0]),
        'Hyderabad': np.array([0,0,0,1,0,0]),
        'Mumbai': np.array([0,0,0,0,1,0]),
        'Kolkata': np.array([0,0,0,0,0,1])}

source_city = np.array(data['source_city'].map(city_map).to_list()).reshape((len(data),6))
destination_city = np.array(data['destination_city'].map(city_map).to_list()).reshape((len(data),6))
#%%
departure_time = np.array(data['departure_time'].to_list()).reshape((len(data),1))
stops = np.array(data['stops'].to_list()).reshape((len(data),1))
arrival_time = np.array(data['arrival_time'].to_list()).reshape((len(data),1))
flclass = np.array(data['class'].to_list()).reshape((len(data),1))
duration = np.array(data['duration'].to_list()).reshape((len(data),1))
days_left = np.array(data['days_left'].to_list()).reshape((len(data),1))
price = np.array(data['price'].to_list()).reshape((len(data),1))
flight = np.array(data['flight_number'].to_list()).reshape((len(data),1))

# The variable used for indicating to which tree region one variable belongs
# to (all data points are initially grouped into the region 0)
tree_regions = np.zeros_like(flclass)
#%%
X_all = np.hstack((airline,source_city,destination_city,
                departure_time,stops,arrival_time,
                flclass,duration,days_left,tree_regions)).astype(float)

y_all = price.astype(float)

# Standardizing the predictors and the response
# X_all[:,0:-1] = standardize(X_all[:,0:-1])
# y_all = standardize(y_all)

np.random.seed(42)

# Performing a test-train split
X_tr, X_te, y_tr, y_te = train_test_split(X_all,y_all,0.2,42)

ave_te_pre = np.zeros_like(y_te)

d=5

for i in range(0,d):

    # Performing a test-train split
    X_tr, X_te, y_tr, y_te = train_test_split(X_all,y_all,0.2,42)
    np.random.seed()
    indices = np.random.choice(len(X_tr), len(X_tr), replace=True)
    X_bootstrap = X_tr[indices]
    y_bootstrap = y_tr[indices]

    max_reg = 1000
    min_points = 100

    # Creating the tree structure based on minimizing RSS, the algorithm will
    # attach an integer region data to all of the data points in the training data
    Tree_Nodes, X_train = Decision_Tree_Constructer(X_bootstrap,y_bootstrap,max_reg,min_points)

    # Finding the mean for each region by using the price values associated with
    # the points in that region
    R_values = find_R_values(X_bootstrap,y_bootstrap)

    # Assigning region values to the points in the validation data set based on
    # the nodes designated by the tree algorithm and using the mean values inside
    # the regions to predict the points in the validation set
    ave_te_pre[:,0] += np.squeeze(predict(X_te,Tree_Nodes,R_values))

# Averaging the predictions made by the tress
ave_te_pre[:] /= d

# Performing a test-train split
X_tr, X_te, y_tr, y_te = train_test_split(X_all,y_all,0.2,42)

# Computing the MSE for the average predictions and real price values on the
# test data set
MSE_te = find_RSS(np.squeeze(y_te),np.squeeze(ave_te_pre))/y_te.shape[0]
RMSE = np.sqrt(MSE_te)

print("RMSE for the test data from the average response of {d} Bagged Trees: \n".format(d=d))
print(RMSE)
```

F) Code for Random Forests (RandomForest_Test_Final.py)

```
# -*- coding: utf-8 -*-


#%% IMPORTING NECESSARY LIBRARIES
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#%% FUNCTION DEFINITIONS

def standardize(X):
    means = np.mean(X, axis=0)
    stds = np.std(X, axis=0)
    standardized_data = (X - means) / stds
    return standardized_data

def train_test_split(X, y, test_size=0.2, random_state=None):

    if random_state is not None:
        np.random.seed(random_state)

    n_samples = X.shape[0]

    indices = np.arange(n_samples)
    np.random.shuffle(indices)

    test_size = int(n_samples * test_size)
    train_indices = indices[:-test_size]
    test_indices = indices[-test_size:]

    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    return X_train, X_test, y_train, y_test

def k_fold_split(X, y, k, shuffle=True, random_state=None):

    if random_state is not None:
        np.random.seed(random_state)

    if shuffle:
        indices = np.arange(X.shape[0])
        np.random.shuffle(indices)
        X_temp = X[indices]
        y_temp = y[indices]

    fold_sizes = X.shape[0] // k
    folds = []

    for i in range(k-1):
        test_indices = np.arange(i * fold_sizes, (i + 1) * fold_sizes)
        train_indices = np.setdiff1d(indices, test_indices)

        X_train, y_train = X_temp[train_indices], y_temp[train_indices]
        X_test, y_test = X_temp[test_indices], y_temp[test_indices]

        folds.append((X_train, y_train, X_test, y_test))

    test_indices = np.arange((k-1) * fold_sizes, X.shape[0])
    train_indices = np.setdiff1d(indices, test_indices)
    X_train, y_train = X_temp[train_indices], y_temp[train_indices]
    X_test, y_test = X_temp[test_indices], y_temp[test_indices]

    folds.append((X_train, y_train, X_test, y_test))
    return folds

def find_RSS(a,b):
    RSS = np.sum((a - b) ** 2)
    return RSS

def find_RSS_1(a,b):
    RSS = np.sum(abs(a - b))
    return RSS

def find_R_values(X,y):
    # Creating an array that'll contain the mean for every region created by
    # the tree algorithm, this array is used for estimating values for new data
    # (The size of the array is the same as the number of regions in the tree)
    R_values = np.zeros(int(np.max(X[:,-1]))+1)

    # For each region in the tree (region number is in -1th column)
    for i in range (0,int(np.max(X[:,-1]))+1):
        # Finding the points that are all in the region i
        R_indices = np.argwhere(X[:,-1]==i)
        # Noting the means of the points on the region i
        R_values[i] = np.mean(y[R_indices])
    return R_values

def choose_predictor(X_reg,y_reg,Dec_Bou):
    # This function chooses among which of the predictors will be the best to
    # split the data from and which value is the best decision boundary

    # This array will hold the minimum RSS value [0] for each predictor and the
    # decision boundary it denotes [1]
    RSS_pre = np.zeros((X_reg.shape[1]-1,2))

    # Selection m=sqrt(p)=5 predictors
    selected_predictors = np.random.choice(np.arange(0, X_reg.shape[1]-1), 5, replace=False)

    # For each predictor (note that last column is used for tree regions)
    for pre in selected_predictors:
        # This variable will hold the minimum RSS value for the predictor
        min_RSS = 10**50
        # This variable will denote the decision boundary for minimum RSS
        min_val = 0
        for bou in Dec_Bou[pre]:
            # Finding the data points under the decision boundary
            down_reg = np.squeeze(y_reg[np.argwhere(X_reg[:,pre]<bou)])
            # Finding RSS for the points under the decision boundary
            RSS_down = np.linalg.norm(down_reg-np.mean(down_reg))**2
            # Finding the data points over the decision boundary
            up_reg = np.squeeze(y_reg[np.argwhere(X_reg[:,pre]>=bou)])
            # Finding RSS for the points over the decision boundary
            RSS_up = np.linalg.norm(up_reg-np.mean(up_reg))**2
            if ((RSS_down + RSS_up)<min_RSS):
                min_RSS = RSS_down + RSS_up
                min_val = bou
        # This is the best decision boundary for the given predictor
        RSS_pre[pre,0] = min_RSS
        RSS_pre[pre,1] = min_val

    cho_pre = np.argmin(RSS_pre[:,0])
    pre_value = RSS_pre[cho_pre,1]

    return cho_pre, pre_value

def create_decision_boundaries(X_sorted):
    # This function creates possible decision boundaries based on the sorted
    # data set

    # Creating the dictionary that will contain the outputs
    Dec_Bou = dict.fromkeys((range(X_sorted.shape[1]-1)),[])

    # For each predictor, finding the unique values for that predictor
    for pre in range(0,X_sorted.shape[1]-1):
        Dec_Bou[pre]=(np.unique(X_sorted[:,pre]))

    return Dec_Bou

def Decision_Tree_Constructer(X_train,y_train,max_reg,min_points):
    # max_reg : maximum number of regions to be created by the tree branches
    # min_points : number of points after which a region will not be seperated
    X = X_train
    y = y_train

    # Creating the dictionary that will contain all the node relations by
    # summining the next two nodes to follow if it is not a terminal node and
    # the number of the region that it flows to if it is a terminal node
    Tree_Nodes = {0: [True,0]}
    # For terminal nodes:
    # {i: [Is end node = True, number of the region it denotes]}
    # For internal nodes:
    # {j : [Is terminal node = False, predictor it splits, value of split,
    #      node 1 (precitor < value), node 2 (precitor >= value)]}
    # Creating an array contaning all the values of the predictors in ascending
    # order (this array will be useful for the tree algorithm where the jumbled up
    # data order does not matter)
    X_sorted = np.zeros_like(X)
    for pre in range(0,X.shape[1]-1):
        X_sorted[:,pre] = np.sort(X[:,pre])

    # Creating the array that will contain the would be decision boundaries
    # for each predictor
    Dec_Bou = create_decision_boundaries(X_sorted)

    # The variable used for keeping track of number of regions
    num_reg = 0
    # The variable used for keeping track of number of nodes
    num_nodes = 0

    while(True):

        # For each node in the tree
        for node in list(Tree_Nodes):
            # Checking whether the max region limit is reached
            if num_reg >= max_reg:
                # If maximum region number is reach the tree construction ends
                break
            # If it is a terminal node
            if (Tree_Nodes[node][0] == True):
                # Region associated with the chosen node
                node_reg =  Tree_Nodes[node][1]
                # Getting all the data in the terminal node region
                X_reg = np.squeeze(X[np.argwhere(X[:,-1]==node_reg),:])
                y_reg = np.squeeze(y[np.argwhere(X[:,-1]==node_reg)])
                # If there are already less than the minimum amount of points
                # in a region, it will not be further seperated
                if (len(X_reg.shape)==2):
                    # Choosing a predictor to split and where to split it
                    cho_pre, pre_value = choose_predictor(X_reg,y_reg,Dec_Bou)
                    # Creating two new terminal nodes for the split
                    Tree_Nodes[num_nodes+1] = [True,node_reg]
                    X[np.argwhere((X[:,cho_pre]<pre_value) & (X[:,-1]==node_reg)),-1] = node_reg
                    Tree_Nodes[num_nodes+2] = [True,num_reg+1]
                    X[np.argwhere((X[:,cho_pre]>=pre_value) & (X[:,-1]==node_reg)),-1] = num_reg+1
                    # Altering the original node because it is no longer terminal
                    Tree_Nodes[node] = [False,cho_pre,pre_value,num_nodes+1,num_nodes+2]
                    num_nodes += 2
                    num_reg += 1

        if num_reg >= max_reg:
            break
    return Tree_Nodes, X

def predict(X,Tree_Nodes,R_values):
    # This function predicts the price values for a given preddictor set based
    # on the region in which it falls on the tree structure and the mean at
    # that region

    # Temporary variable used for keeping the nodes to which points are diverted
    X_nodes = np.zeros(X.shape[0])

    # The variable that'll hold the predicted value of price
    y_val_pre = np.zeros(X.shape[0])

    # For each node
    for node in list(Tree_Nodes):
        # If the node is not a terminal node, the points are sent to the
        # following nodes through the structure of the tree
        if Tree_Nodes[node][0] == False:
            X_nodes[np.argwhere((X_nodes[:] == node) &
(X[:,Tree_Nodes[node][1]]<Tree_Nodes[node][2]))] = Tree_Nodes[node][3]
            X_nodes[np.argwhere((X_nodes[:] == node) &
(X[:,Tree_Nodes[node][1]]>=Tree_Nodes[node][2]))] = Tree_Nodes[node][4]
        # If the node is a terminal node, the points are used to label regions
        if Tree_Nodes[node][0] == True:
            try:
                y_val_pre[np.argwhere(X_nodes[:]==node)] = R_values[Tree_Nodes[node][1]]
            except:
                pass
    return y_val_pre

#%%

# Load the dataset
data = pd.read_csv('/Users/asus/Desktop/485P/Clean_Dataset.csv')
# Remove the 'Unnamed: 0' column
data.drop(columns=['Unnamed: 0'], inplace=True)
# Custom mapping for 'departure_time' and 'arrival_time'
time_of_day_mapping = {
    'Early_Morning': 1,
    'Morning': 2,
    'Afternoon': 3,
    'Evening': 4,
    'Night': 5,
    'Late_Night': 6
}
data['departure_time'] = data['departure_time'].map(time_of_day_mapping)
data['arrival_time'] = data['arrival_time'].map(time_of_day_mapping)
```

```
# Convert 'stops' to integer
stops_mapping = {'zero': 0, 'one': 1, 'two_or_more': 2}
data['stops'] = data['stops'].map(stops_mapping)

class_mapping = {'Economy' : 0, 'Business' : 1}
data['class'] = data['class'].map(class_mapping)

# Extract the numerical part of the 'flight' feature
flight_number = data['flight'].str.extract('(\d+)').astype(int)

# Find the index of the 'flight' column
flight_col_index = data.columns.get_loc('flight')

# Insert the 'flight_number' column in the place of 'flight'
data.insert(flight_col_index, 'flight_number', flight_number)

# Drop the original 'flight' column
data.drop(columns=['flight'], inplace=True)
#%%
airline_map = {'Air_India': np.array([1,0,0,0,0,0]),
        'AirAsia': np.array([0,1,0,0,0,0]),
        'GO_FIRST': np.array([0,0,1,0,0,0]),
        'Indigo': np.array([0,0,0,1,0,0]),
        'SpiceJet': np.array([0,0,0,0,1,0]),
        'Vistara': np.array([0,0,0,0,0,1])}

airline = np.array(data['airline'].map(airline_map).to_list()).reshape((len(data),6))
#%%
city_map = {'Bangalore': np.array([1,0,0,0,0,0]),
        'Chennai': np.array([0,1,0,0,0,0]),
        'Delhi': np.array([0,0,1,0,0,0]),
        'Hyderabad': np.array([0,0,0,1,0,0]),
        'Mumbai': np.array([0,0,0,0,1,0]),
        'Kolkata': np.array([0,0,0,0,0,1])}

source_city = np.array(data['source_city'].map(city_map).to_list()).reshape((len(data),6))
destination_city = np.array(data['destination_city'].map(city_map).to_list()).reshape((len(data),6))
#%%
departure_time = np.array(data['departure_time'].to_list()).reshape((len(data),1))
stops = np.array(data['stops'].to_list()).reshape((len(data),1))
arrival_time = np.array(data['arrival_time'].to_list()).reshape((len(data),1))
flclass = np.array(data['class'].to_list()).reshape((len(data),1))
duration = np.array(data['duration'].to_list()).reshape((len(data),1))
days_left = np.array(data['days_left'].to_list()).reshape((len(data),1))
price = np.array(data['price'].to_list()).reshape((len(data),1))
flight = np.array(data['flight_number'].to_list()).reshape((len(data),1))

# The variable used for indicating to which tree region one variable belongs
# to (all data points are initially grouped into the region 0)
tree_regions = np.zeros_like(flclass)
#%%
X_all = np.hstack((airline,source_city,destination_city,
                departure_time,stops,arrival_time,
                flclass,duration,days_left,tree_regions)).astype(float)

y_all = price.astype(float)

# Standardizing the predictors and the response
# X_all[:,0:-1] = standardize(X_all[:,0:-1])
# y_all = standardize(y_all)

np.random.seed(42)

# Performing a test-train split
X_tr, X_te, y_tr, y_te = train_test_split(X_all,y_all,0.2,42)

ave_te_pre = np.zeros_like(y_te)

d=5

for i in range(0,d):

    # Performing a test-train split
    X_tr, X_te, y_tr, y_te = train_test_split(X_all,y_all,0.2,42)

    np.random.seed()
    indices = np.random.choice(len(X_tr), len(X_tr), replace=True)
    X_bootstrap = X_tr[indices]
    y_bootstrap = y_tr[indices]

    max_reg = 1000
    min_points = 100

    # Creating the tree structure based on minimizing RSS, the algorithm will
    # attach an integer region data to all of the data points in the training data
    Tree_Nodes, X_train = Decision_Tree_Constructer(X_bootstrap,y_bootstrap,max_reg,min_points)

    # Finding the mean for each region by using the price values associated with
    # the points in that region
    R_values = find_R_values(X_bootstrap,y_bootstrap)

    # Assigning region values to the points in the validation data set based on
    # the nodes designated by the tree algorithm and using the mean values inside
    # the regions to predict the points in the validation set
    ave_te_pre[:,0] += np.squeeze(predict(X_te,Tree_Nodes,R_values))
```

```
# Averaging the predictions made by the tress
ave_te_pre[:] /= d

# Performing a test-train split
X_tr, X_te, y_tr, y_te = train_test_split(X_all,y_all,0.2,42)

# Computing the MSE for the average predictions and real price values on the
# test data set
MSE_te = find_RSS(np.squeeze(y_te),np.squeeze(ave_te_pre))/y_te.shape[0]
RMSE = np.sqrt(MSE_te)

print("RMSE for the test data from the average response of {d} Bagged Trees with Random Forests:
\n".format(d=d))
print(RMSE)
```

G) Code for Neural Networks (NeuralNet.py)

```
#%% IMPORTING NECESSARY LIBRARIES
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


#%% FUNCTION DEFINITIONS

def standardize(X):
  means = np.mean(X, axis=0)
  stds = np.std(X, axis=0)
  standardized_data = (X - means) / stds
  return standardized_data

def train_test_split(X, y, test_size=0.2, random_state=None):

  if random_state is not None:
    np.random.seed(random_state)

  n_samples = X.shape[0]

  indices = np.arange(n_samples)
  np.random.shuffle(indices)

  test_size = int(n_samples * test_size)
  train_indices = indices[:-test_size]
  test_indices = indices[-test_size:]

  X_train, X_test = X[train_indices], X[test_indices]
  y_train, y_test = y[train_indices], y[test_indices]

  return X_train, X_test, y_train, y_test
```

```
def find_RSS(a,b):
    RSS = np.sum((a - b) ** 2)
    return RSS

def find_RSS_1(a,b):
    RSS = np.sum(abs(a - b))
    return RSS


#%% Class Definition

class SimpleNeuralNetwork:
    def __init__(self, layers, learning_rate=0.001):
        self.layers = layers
        self.learning_rate = learning_rate
        self.weights, self.biases = self.initialize_weights_he(layers)

    @staticmethod
    def sigmoid(x):
        return 1 / (1 + np.exp(-x))

    @staticmethod
    def sigmoid_derivative(x):
        return SimpleNeuralNetwork.sigmoid(x) * (1 - SimpleNeuralNetwork.sigmoid(x))

    @staticmethod
    def tanh(x):
        return np.tanh(x)

    @staticmethod
    def tanh_derivative(x):
        return 1 - np.tanh(x)**2

    @staticmethod
    def relu(x):
        return np.maximum(0, x)

    @staticmethod
    def relu_derivative(x):
        return (x > 0).astype(float)

    @staticmethod
    def leaky_relu(x, alpha=0.01):
        return np.where(x > 0, x, x * alpha)

    @staticmethod
    def leaky_relu_derivative(x, alpha=0.01):
        dx = np.ones_like(x)
        dx[x < 0] = alpha
        return dx

    @staticmethod
    def identity(x):
        return x

    @staticmethod
    def identity_derivative(x):
        return np.ones_like(x)

    @staticmethod
    def initialize_weights_he(layers):
        weights = []
        biases = []
        for i in range(len(layers) - 1):
            weight = np.random.randn(layers[i + 1], layers[i]) * np.sqrt(2 / layers[i])
            bias = np.zeros((layers[i + 1], 1))
            weights.append(weight)
            biases.append(bias)
        return weights, biases

    def feedforward(self, X):
        activations = [X.reshape(-1, 1)]
        for i in range(len(self.weights)):
            func = self.relu if i < len(self.weights) - 1 else self.identity
            activations.append(func(np.dot(self.weights[i], activations[i]) + self.biases[i]))
        return activations

    def backpropagation(self, y, activations):
        deltas = []
        y = y.reshape(-1, 1)
        for i in reversed(range(len(self.weights))):
            if i == len(self.weights) - 1:
                error = activations[-1] - y
                delta = error * self.identity_derivative(activations[i + 1])
            else:
                error = np.dot(self.weights[i + 1].T, deltas[-1])
                delta = error * self.relu_derivative(activations[i + 1])
            deltas.append(delta)

            weight_adjustment = np.dot(delta, activations[i].T)
            bias_adjustment = np.sum(delta, axis=1, keepdims=True)

            self.weights[i] -= self.learning_rate * weight_adjustment
            self.biases[i] -= self.learning_rate * bias_adjustment
    def train(self, X_train, y_train, epochs):
        for epoch in range(epochs):
            total_error = 0
            for X, y in zip(X_train, y_train):
                X = X.reshape(-1, 1)
                y = y.reshape(-1, 1)

                activations = self.feedforward(X)
                self.backpropagation(y, activations)

                total_error += np.mean((y - activations[-1]) ** 2)

            avg_error = total_error / len(X_train)
            print(f"Epoch {epoch + 1}/{epochs}, Error: {avg_error}")

    @staticmethod
    def normalize_data(X):
        return X / np.max(X)


#%%

# Load the dataset
data = pd.read_csv('/Users/asus/Desktop/485P/Clean_Dataset.csv')
# Remove the 'Unnamed: 0' column
data.drop(columns=['Unnamed: 0'], inplace=True)
# Custom mapping for 'departure_time' and 'arrival_time'
time_of_day_mapping = {
    'Early_Morning': 1,
    'Morning': 2,
    'Afternoon': 3,
    'Evening': 4,
    'Night': 5,
    'Late_Night': 6
}
data['departure_time'] = data['departure_time'].map(time_of_day_mapping)
data['arrival_time'] = data['arrival_time'].map(time_of_day_mapping)

# Convert 'stops' to integer
stops_mapping = {'zero': 0, 'one': 1, 'two_or_more': 2}
data['stops'] = data['stops'].map(stops_mapping)

class_mapping = {'Economy' : 0, 'Business' : 1}
data['class'] = data['class'].map(class_mapping)

# Extract the numerical part of the 'flight' feature
flight_number = data['flight'].str.extract('(\d+)').astype(int)
```

```
# Find the index of the 'flight' column
flight_col_index = data.columns.get_loc('flight')

# Insert the 'flight_number' column in the place of 'flight'
data.insert(flight_col_index, 'flight_number', flight_number)

# Drop the original 'flight' column
data.drop(columns=['flight'], inplace=True)
#%%
airline_map = {'Air_India': np.array([1,0,0,0,0,0]),
        'AirAsia': np.array([0,1,0,0,0,0]),
        'GO_FIRST': np.array([0,0,1,0,0,0]),
        'Indigo': np.array([0,0,0,1,0,0]),
        'SpiceJet': np.array([0,0,0,0,1,0]),
        'Vistara': np.array([0,0,0,0,0,1])}

airline = np.array(data['airline'].map(airline_map).to_list()).reshape((len(data),6))
#%%
city_map = {'Bangalore': np.array([1,0,0,0,0,0]),
        'Chennai': np.array([0,1,0,0,0,0]),
        'Delhi': np.array([0,0,1,0,0,0]),
        'Hyderabad': np.array([0,0,0,1,0,0]),
        'Mumbai': np.array([0,0,0,0,1,0]),
        'Kolkata': np.array([0,0,0,0,0,1])}

source_city = np.array(data['source_city'].map(city_map).to_list()).reshape((len(data),6))
destination_city = np.array(data['destination_city'].map(city_map).to_list()).reshape((len(data),6))
#%%
departure_time = np.array(data['departure_time'].to_list()).reshape((len(data),1))
stops = np.array(data['stops'].to_list()).reshape((len(data),1))
arrival_time = np.array(data['arrival_time'].to_list()).reshape((len(data),1))
flclass = np.array(data['class'].to_list()).reshape((len(data),1))
duration = np.array(data['duration'].to_list()).reshape((len(data),1))
days_left = np.array(data['days_left'].to_list()).reshape((len(data),1))
price = np.array(data['price'].to_list()).reshape((len(data),1))
flight = np.array(data['flight_number'].to_list()).reshape((len(data),1))

# The variable used for indicating to which tree region one variable belongs
# to (all data points are initially grouped into the region 0)
tree_regions = np.zeros_like(flclass)
#%%
X_all = np.hstack((airline,source_city,destination_city,
                departure_time/np.max(departure_time),stops/np.max(stops),
                arrival_time/np.max(arrival_time),
                flclass,duration/np.max(duration),
                days_left/np.max(days_left))).astype(float)

y_all = price.astype(float)
```

```
#%%

# Performing a test-train split
X_train, X_test, y_train, y_test = train_test_split(X_all,y_all,0.2,42)

# Creating the neural network
nn = SimpleNeuralNetwork(layers=[24,8,6,1], learning_rate=10**(-10))

# Seperating a validation set to be used for early stopping
X_nn_train, X_val, y_nn_train, y_val = train_test_split(X_train, y_train, 0.3, 42)

# Variable used for counting how many epoch have passed to limit the training time
esn = 0

# Variable used for stopping the training by early stopping
running = True

# Variable used for judging RMSE on the validation data
RMSE_o = 10**10

while running:

    nn.train(X_nn_train, y_nn_train, epochs = 50)

    RSS = 0
    for n in range(X_val.shape[0]):
        activations = nn.feedforward(X_val[n,:])
        RSS += (activations[-1]-y_val[n])**2

    # Computing the MSE for the average predictions and real price values on the
    # validation data set
    MSE_test = RSS/y_val.shape[0]
    RMSE_n = np.sqrt(MSE_test)
    print("RMSE on validation data = {r}".format(r=RMSE_n[0][0]))

    if RMSE_n > RMSE_o:
        running = False
        print("Ended training due to early stopping")

    RMSE_o = RMSE_n

    esn += 1

    if esn == 200:
        running = False
        print("Ended training due to reaching 10000 epochs")
```

```
RSS = 0
for n in range(X_test.shape[0]):
    activations = nn.feedforward(X_test[n,:])
    RSS += (activations[-1]-y_test[n])**2

# Computing the MSE for the average predictions and real price values on the
# test data set
MSE_test = RSS/y_test.shape[0]
RMSE_n = np.sqrt(MSE_test)
print("RMSE on test data = {r}".format(r=RMSE_n[0][0]))
```