Kemal Enes Akyüz - 22003521
Efe Tarhan - 22002840

EEE485 - Statistical Learning and Data Analytics
Term Project First Report

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

Introduction

The project presented within this report aims to determine the relation between plane ticket prices and different features extracted from the conditions when the ticket has been bought. These features can be the airline company, arrival city, destination city, arrival time, destination time, how many days the ticket has been bought before the flight class of the flight, and some more. Several machine learning approaches have been tried to analyze and create a model that can make meaningful price predictions to accomplish this goal. All machine learning algorithms in this project have been written by hand, and only essential (numpy, pandas) Python libraries have been used for the task. This report will briefly describe the problem and the used dataset, explain the methods used, and discuss expected challenges throughout the project.

Problem and Dataset Description

The problem that will be dealt with in this project is the prediction of ticket prices of the six biggest airline companies in India between 6 major cities. Several categorical and numerical features in the dataset can 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 Time**: 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**: This feature indicates how many days earlier the ticket has been bought before the flight is to take 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 Time**: 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.

The dataset has been updated using Spyder Python editor. The first five rows of the updated dataset can be examined from Figure 2.

Fig.2 First 5 entries of the updated plane ticket price prediction dataset

Applied Methods

Three methods have been applied for training a model that can be used for predicting plane ticket prices. These methods are linear regression, random forest regression, and deep learning.

1- Linear Regression

For linear regression, two different metrics have been used to evaluate the model's success: the MSE and the R^2 score. The definition and meaning of these metrics can be seen below:

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

$$MSE\left(y_{predict}, y_{true}\right) = \frac{1}{N}\left|\left|y_{predict} - y_{true}\right|\right|^2$$

R^2 (Coefficient of Determination) Score [3]:
R-squared is a measure of the goodness of a fit of the model, it measures how much of the variation of the test can be expressed with the model.

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

The first method used for training the dataset is linear regression. Linear regression can be successful compared to other methods when capturing linear relationships between features in the data and it is rather simple to explain and visualize the data which makes it ideal to apply on this data to become the benchmark for the methods to come. The common formula for finding the parameters of the linear regression model is the following:

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

This formula comes from the least squares problem of linear algebra where a vector is being tried to project onto a subspace that has the minimum distance between each other, i.e., that minimizes the following cost function:

$$C = \left|\left|y_{predicted} - y_{train}\right|\right|^2$$

This method has been tried on Python and R2, and MSE validation methods have been used for the evaluation of the performance of the operation. All linear regression tests have been completed after standardizing the data by subtracting the mean and dividing by standard deviation. Also %20 of the dataset has been allocated for testing and the other %80 has been used for k-fold cross validation. After training the OLS (Ordinary Least Squares) algorithm using 10-fold cross validation, the following plots have been obtained for the values of MSE and tests, which can be seen in Figures 3 and 4.

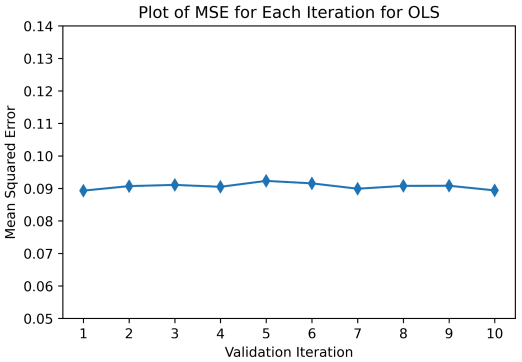

Fig.3 MSE values for each iteration of 10-fold validation of OLS

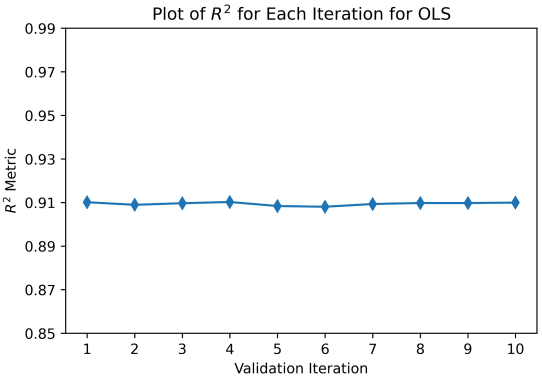

Fig.4 R^2 values for each iteration of 10-fold validation of OLS

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) can be used to decrease the model's overfitting 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 centralized, the lambda parameter has been adjusted using the 10-fold cross-validation. The results can be seen in Figures 5 and 6.

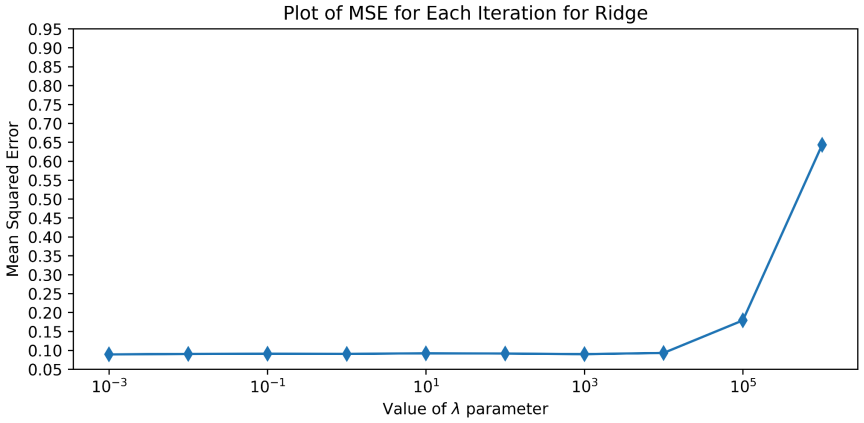

Fig.5 MSE values for each iteration of 10-fold validation of ridge regression

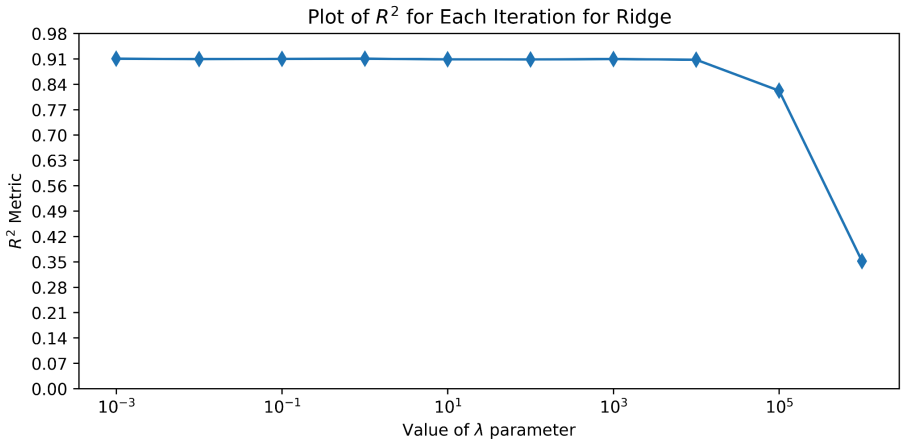

Fig.6 R^2 values for each iteration of 10-fold validation of ridge regression

As can be seen, increasing the lambda value decreased the model's performance. This happened because the linear regression model suffers from underfitting; additional regularization decreases the model's fit to the given dataset. After completing the ridge, the model has tried lasso regression to see if any parameter can be eliminated that creates a sparse parameter set. The cost function of the lasso can be seen below:

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

To optimize the parameters using this cost function, an iterative numerical approach has been used. Results of 30-fold iteration using lasso regression can be seen from Figures 7 and 8.

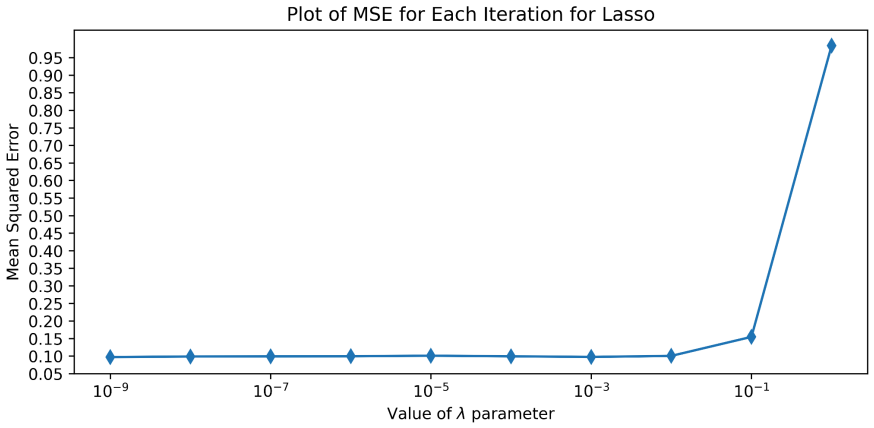

Fig.7 MSE values for each iteration of 30-fold validation of lasso regression

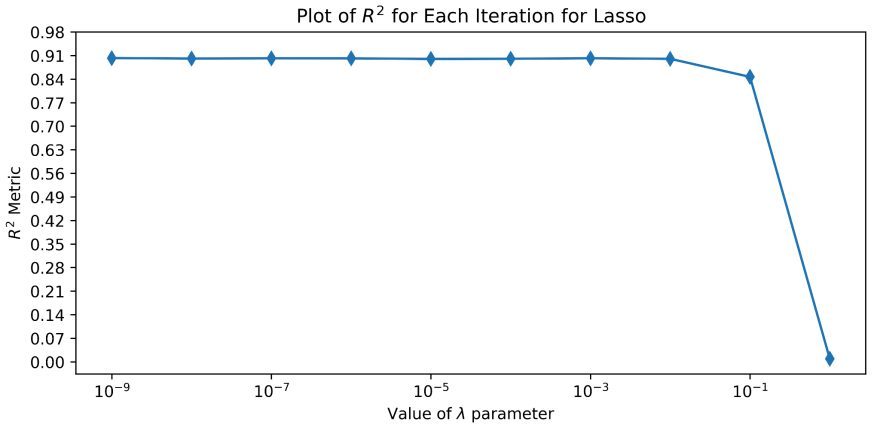

Fig.8 R^2 values for each iteration of 30-fold validation of lasso regression

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.

	MSE	R^2
OLS	0.0897	0.910
Ridge Regression	0.0896	0.910
Lasso Regression	0.0979	0.901

Table 1. MSE and R^2 results for the linear regression

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 value (or a class for the classification problems). Since the rules used for splitting the space into regions, namely through nodes that judge points based on the value of one single predictor, end 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.

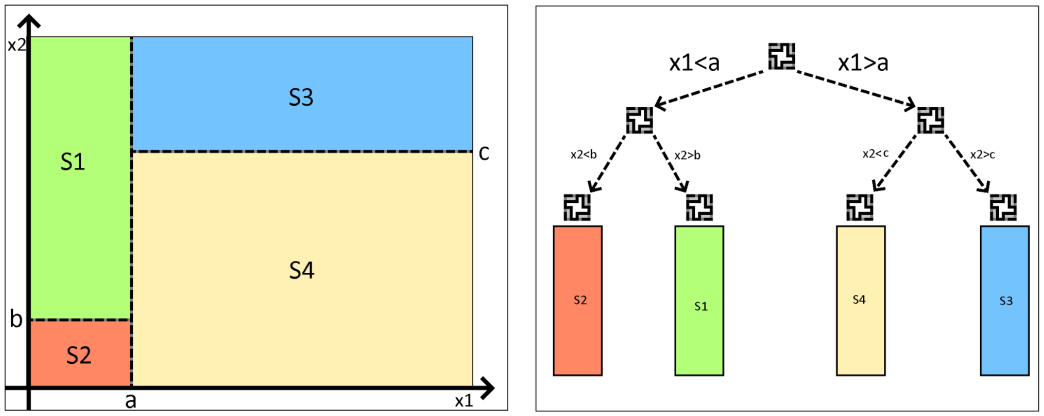

Figure 9: The tree like structure of decision tree algorithm visualized for two predictors and four regions

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 (or the class) 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.

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

To minimize RSS, the algorithm takes one of the regions that the training data was divided into (if it is at the first iteration, it takes the whole training data set) 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 with the said predictor lower than the boundary is separated into a new region. 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 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 splitting the data into is defined.
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. This hyper-parameter is optimized by comparing different values through the average MSE for each fold in the 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 10.

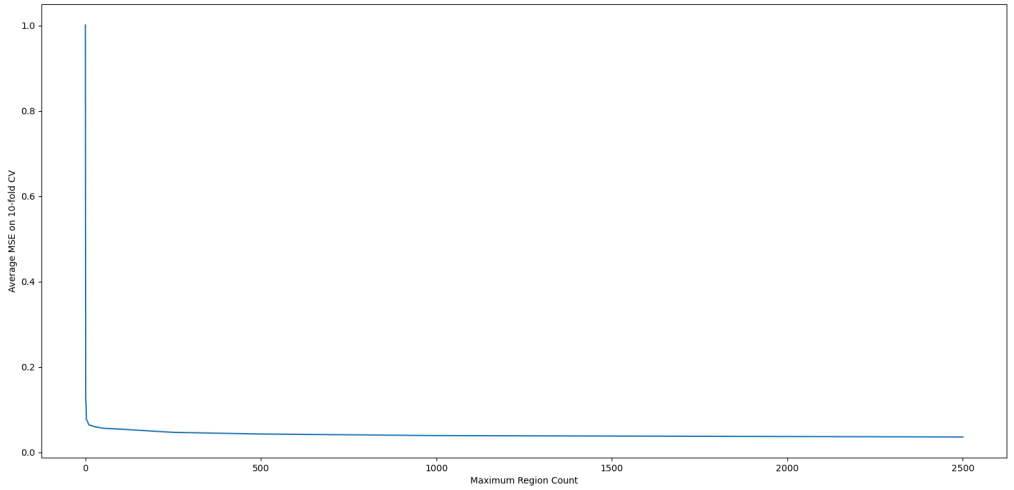

Figure 10: The Average MSE for 10-fold Cross Validation on test data with respect to maximum regions

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, reaching around an hour for 2500 regions cross-validation, a trade-off was decided at 1000 regions. 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. The resulting MSE is shown in Table 2.

	MSE
Linear Regression	0.0896
Decision Tree	0.0392

Table 2. MSE result for the decision tree and linear regression

For the following stages of the project, the aim is to introduce numerous trees trained on different data sets generated from the same training set with the method of bootstrapping, or in other words using Random Forests method to further improve the result of the Decision Trees.

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. 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 11.

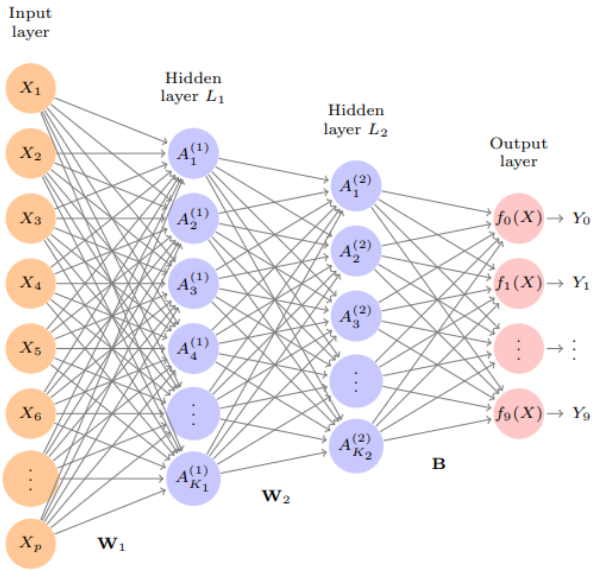

Figure 11: A general structure for the multilayer neural networks [4]

Weights and biases of a neural network can be learned using the 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:

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

Neural network implementation will be completed in the final report and demonstration of the project.

Expected Challenges

The biggest challenge that was faced up until this point of the project was that due to the large number of data points and the relatively small number of predictors, both methods used so far in the project (linear regression and decision trees) always underfit the data set. This problem resulted in the fact that some performance needed to be included. For example, the decision tree algorithm may work better for more regions, even if only slightly. However, such levels could not be attained due to the un-optimized nature of the program, increasing training times beyond justifiable levels. One challenge in the remaining part of the project may be to overcome this issue and modify the methods already employed and those that will be employed to be adequately complex for the data set.
Another challenge is to find ways to optimize hyperparameters. As in the case of the maximum region numbers in the decision tree, the optimization process takes too long, and considering that some of the methods to be employed will have more hyperparameters to optimize, the hyper-parameter optimization process may become challenging.
Lastly, the explainability of the solutions needs to be higher for the part of the project presented in this report. Though both linear regression and decision trees are easily explainable methods, connecting their results with intuitive expectations about the data set and then coming up with diagrams and presentations that'll show these relations may be challenging.

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, were attributed to Efe Tarhan. On the other hand, the decision tree implementation workload was given to Kemal Enes Akyüz. The neural network aspect of the project will be 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 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

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)
    mse_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_norm,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_ols(X_train, y_train)
        y_pred = predict(X_val, beta)
        dum_mse = mse(y_val, y_pred)
        dum_r2 = r2(y_val, y_pred)
        betas.append(beta)
        mse_list.append(dum_mse)
        r2_list.append(dum_r2)
        cnt += 1
        print("%{} of the process is completed !".format(cnt/len(folds)*100))
    cross_val_beta = sum(betas)/len(betas)
    test_pred = predict(X_test, cross_val_beta)
    test_mse = mse(y_test, test_pred)
    test_r2 = r2(y_test, test_pred)
    return mse_list,r2_list,cross_val_beta,test_mse,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)
    mse_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_norm,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)
        dum_r2 = r2(y_val, y_pred)
        betas.append(beta)
        mse_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[mse_list.index(min(mse_list))]
    test_pred = predict(X_test, best_beta)
    test_mse = mse(y_test, test_pred)
    test_r2 = r2(y_test, test_pred)
    return mse_list,r2_list,lambdas,test_mse,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-4) for i in np.arange(-5,6,1)]
    X_all,X_all_,Y_all = data_process(path)
    mse_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_norm,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)
        dum_r2 = r2(y_val, y_pred)
        betas.append(beta)
        mse_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[mse_list.index(min(mse_list))]
    test_pred = predict(X_test, best_beta)
    test_mse = mse(y_test, test_pred)
    test_r2 = r2(y_test, test_pred)
    return mse_list,r2_list,lambdas,test_mse,test_r2

#%% Training of the OLS (Ordinary Least Squares)
mse_list,r2_list,beta,test_mse,test_r2 =
train_ols("/Users/efetarhan/Desktop/DATA/Clean_Dataset.csv")

plt.figure(dpi = 600)
plt.plot([i for i in range(1,11)],mse_list,'-d')
plt.xlabel("Validation Iteration")
```

```
plt.ylabel("Mean Squared Error")
plt.title("Plot of MSE for Each Iteration for OLS")
plt.xticks([i for i in range(1,11)])
plt.yticks(np.arange(0.05,0.15,0.01))

plt.figure(dpi = 600)
plt.plot([i for i in range(1,11)],r2_list,'-d')
plt.xlabel("Validation Iteration")
plt.ylabel("$R^2$ Metric")
plt.title("Plot of $R^2$ for Each Iteration for OLS")
plt.xticks([i for i in range(1,11)])
plt.yticks(np.arange(0.85,1,0.02))

#%% Ridge Regression

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

plt.figure(figsize = (9,4),dpi = 600)
plt.plot(lambdas[:-1],mse_list,'-d')
plt.xlabel("Value of $\lambda$ parameter")
plt.ylabel("Mean Squared Error")
plt.xscale("log")
plt.title("Plot of MSE for Each Iteration for Ridge")
plt.xticks([lambdas[2*i] for i in range(len(lambdas)//2)])
plt.yticks(np.arange(0.05,1,0.05))

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(0,1,0.07))

#%% Lasso Regression

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

plt.figure(figsize = (9,4),dpi = 600)
plt.plot(lambdas[:-1],mse_list,'-d')
plt.xlabel("Value of $\lambda$ parameter")
plt.ylabel("Mean Squared Error")
plt.xscale("log")
plt.title("Plot of MSE for Each Iteration for Lasso")
plt.xticks([lambdas[2*i] for i in range(len(lambdas)//2)])
```

```
plt.yticks(np.arange(0.05,1,0.05))

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(0,1,0.07))
```

B) Code for Regression Tree Part (Decision_Tree1.py)

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


#%% IMPORTING NECESSARY LIBRARIES
import pandas as pd
import numpy as np

#%% 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:
         y_val_pre[np.argwhere(X_nodes[:]==node)] = R_values[Tree_Nodes[node][1]]
   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)
# Data dimensions
#print(data.shape)
# Summary of the data set
#print(data.info())

#%%
#Response

# 'price' :            The price of the flight which is a continuous variable
#                    and target of the prediction task of this dataset


# Predictors:

# 'airline' :          6 categorical values that include the names of the
#                    airline companies

# 'flight' :           The planes' flight code which is a categorical value

# 'source_city' :      6 categorical values that denote the names of the
#                    cities that the flight is being departed

# 'departure_time' :   6 categorical time labels indicating departure time of
#                    the flights (quantized version of the 24 hours into
#                    6 intervals)

# 'stops' :            3 categorical values that address the number
#                    of stops of the flight

# 'arrival_time' :     6 categorical time labels indicating arrival time of
#                    the flights (quantized version of the 24 hours into
#                    6 intervals)

# 'destination_city' : 6 categorical values that denote the landing location
#                    of the flight

# 'class' :            2 categorical values which indicate if the flight is
#                    "economy" or "business" class

# 'duration' :         A continuous feature that represents how long the
#                    flight is expected to last
```

```
# 'days_left' :        This feature indicates how many days earlier the
#                    ticket has been bought before the flight

#%%

# 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)

X_tr, X_te, y_tr, y_te = train_test_split(X_all,y_all,0.2,42)



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_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]

print("MSE for the test data (for 1000 max regions): \n")
print(MSE_te)
```