# PRML Bonus Project

Susim Mukul Roy B20AI043

## Problem Statement

Flight ticket prices can be something hard to guess, today we might see a price, check out the price of the same flight tomorrow, it will be a different story. In this project, I predicted the **price** of flight tickets using 10 features like Source, Destination, Airline etc. The dataset has 10683 samples.

## Data analysis and Ideas

As is clear from the dataset, we need to break down some columns and remove some columns in the process. Here are the operations performed on the columns:

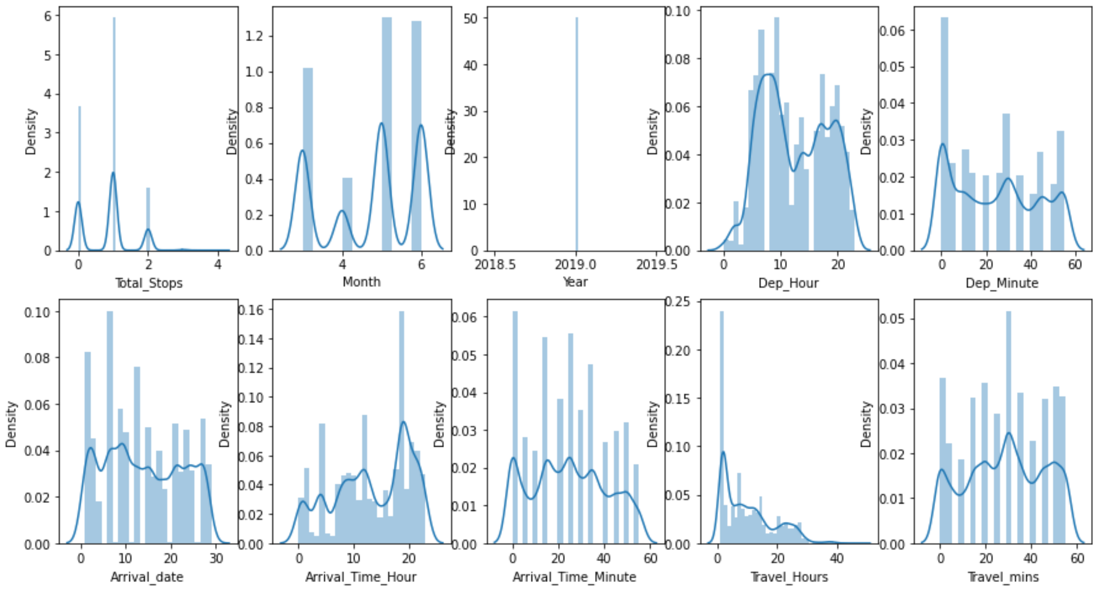
- Date of Journey – I divided the data here into *Date*, *Month* and *Year* columns.
- Route – I divided the data here into 4 columns *City1*, *City2*, *City3* and *City4* since the maximum number of cities a airplane travels is 4.
- Dep_Time – I divided the data here in *Dep_Hour* and *Dep_Minute*.
- Arrival_Time – I divided the data here into *Arrival_date*, *Time_of_Arrival* and then divided *Time_of_Arrival* into *Arrival_Time_Hour* and *Arrival_Time_Minute*.
- Duration – I divided the data here into *Travel_Hours* and *Travel_mins*.

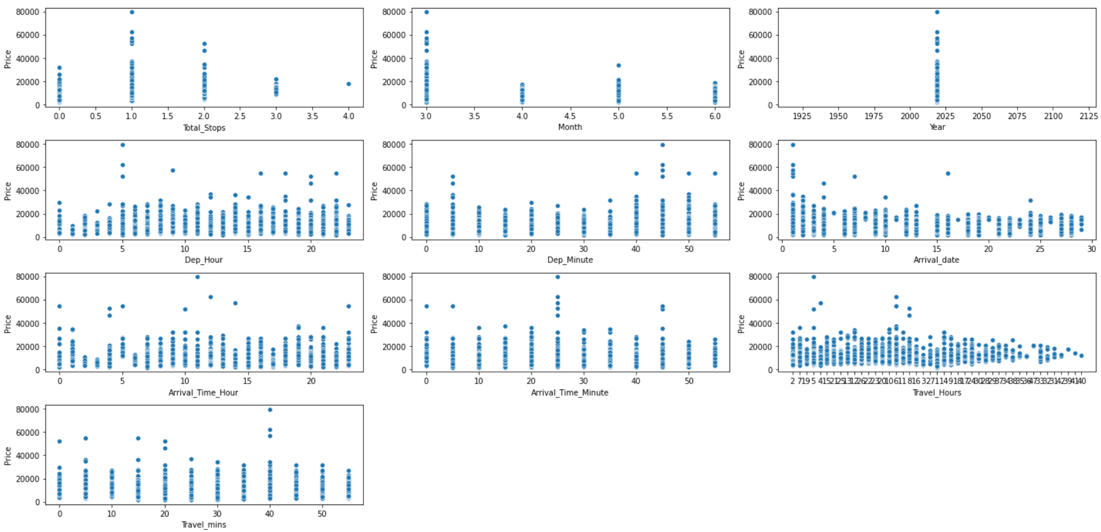
Next, on visualizing the column *Total_Stops*, I saw a value '*non-stop*' which I changed to *0.*

```
Airline                  0
Date_of_Journey          0
Source                   0
Destination              0
Route                    1
Dep_Time                 0
Arrival_Time             0
Duration                 0
Total_Stops              1
Additional_Info          0
Price                    0
Date                     0
Month                    0
Year                     0
City1                    1
City2                    1
City3                 3492
City4                 9117
Dep_Hour                 0
Dep_Minute               0
Arrival_date          6348
Time_of_Arrival          0
Arrival_Time_Hour        0
Arrival_Time_Minute      0
Travel_Hours             0
Travel_mins           1032
dtype: int64
```

In dealing with the **NaN** values as shown above, I saw *City3, City4, Arrival_date* and *Travel_mins* to have a lot of them. I replaced some of them with '*None*'/ '*0*' values.

On visualizing the newly made columns, I found them as follows:




By careful observations like Year column has only 2019 as its value, I decided to dropped the *City4*, *Date_of_Journey*, *Route*,*Time_of_Arrival*, *Duration*, *Arrival_Time* and *Dep_Time* columns.

Following this, I dropped the only row which has a **NaN** value and dropped *Dep_Minute, Arrival, Time_Minute, Travel_mins, Year* and *City1* columns.

# Model Selection

Based on the data analysis, I decided to try the following models-

KNN, Decision Tree Regressor, Random Forest Regressor, Convolutional Neural Network, Multi-Layer Perceptron, XGBoost, SVM, Lasso regression and AdaBoost.

The best hyperparameters for these models were chosen by **30%** splitting of the dataset into **train** and **test** and then finding the *mean_squared_error(MSE).*

For the **CNN** architecture, I tried different combinations of layers and ultimately found the following structure to be giving the best results:

```
model = Sequential()
model.add(Conv1D(256, 3, padding='same', activation="relu", input_shape=(x_train.shape[1],1)))
model.add(BatchNormalization())
model.add(MaxPooling1D(pool_size=3))
model.add(Conv1D(128, 3,padding='same', activation="relu"))
#model.add(BatchNormalization())
model.add(MaxPooling1D(pool_size=3))
model.add(Conv1D(128, 3,padding='same',activation="relu"))
model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dense(64, activation="relu"))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(loss ='mean_squared_error', optimizer = "RMSProp")
model.summary()
```

Similarly, for **MLP**, I tried different combinations of layers and ultimately found the following structure to be giving the best results:

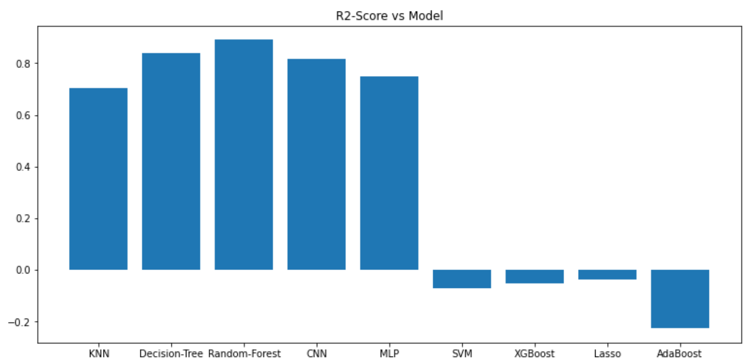
```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(13, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 1)
        )
    def forward(self, x):
        return self.layers(x)

net = MLP()
summary(net,(1,13))
```

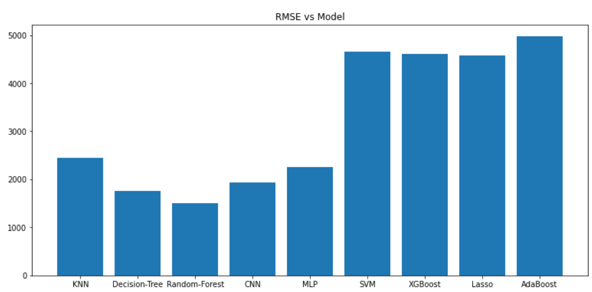
# <u>Metric analysis</u>

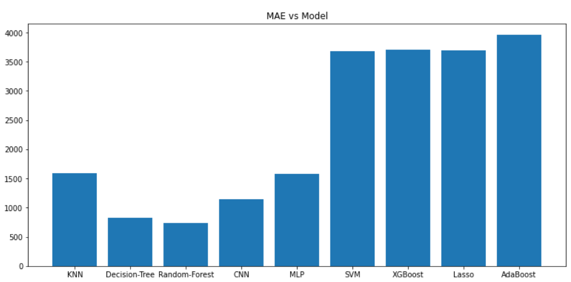
Using a comparison table, I saw the performance of each model which is as follows:

| | Models | RMSE | R2-Score | MAE |
|---|---|---|---|---|
| 0 | KNN | 2445.845119 | 0.703460 | 1594.377847 |
| 1 | Decision-Tree | 1758.927946 | 0.841071 | 825.967867 |
| 2 | Random-Forest | 1498.251527 | 0.890369 | 735.410582 |
| 3 | CNN | 1928.287528 | 0.815681 | 1145.884029 |
| 4 | MLP | 2256.504709 | 0.747595 | 1582.397217 |
| 5 | SVM | 4652.705413 | -0.073092 | 3685.599592 |
| 6 | XGBoost | 4604.837239 | -0.051125 | 3704.298584 |
| 7 | Lasso | 4575.808789 | -0.037914 | 3688.883440 |
| 8 | AdaBoost | 4973.267200 | -0.226053 | 3960.388398 |

The metric that was chosen for the comparison between the models is *mean_squared_error*. We also calculated *mean_absolute_error* and *r2_score* along with that. The following plots show their performance:

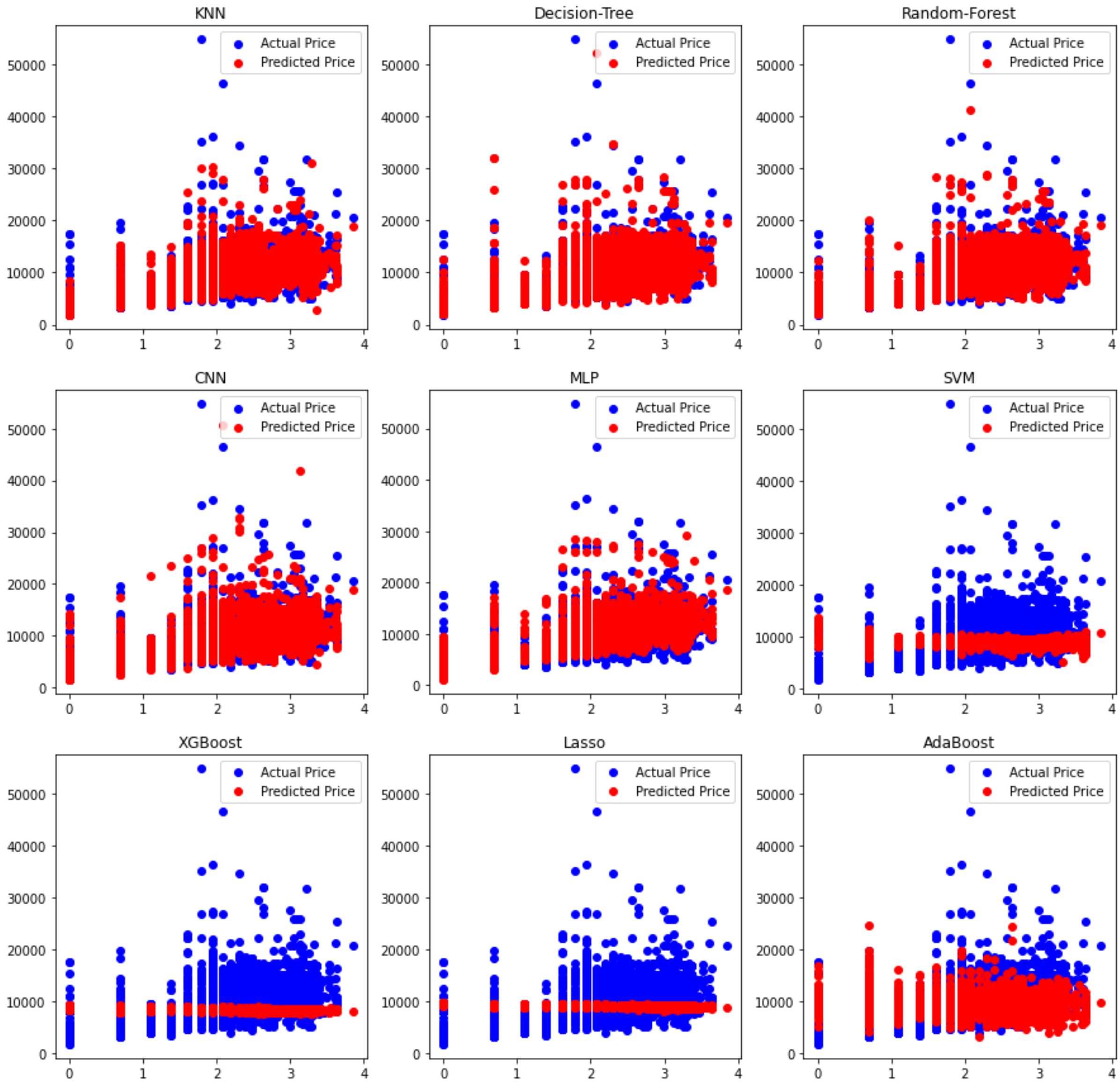


As we see, in all the cases, **Random Forest Regressor** gives the best results in all the three metrics. Hence, I declare that as the best model for our dataset. After this, I observed after many trials runs that **Decision Tree Regressor** and **CNNs** have close r2-scores, MSE and MAE. Hence both of them has the same performance on an average.

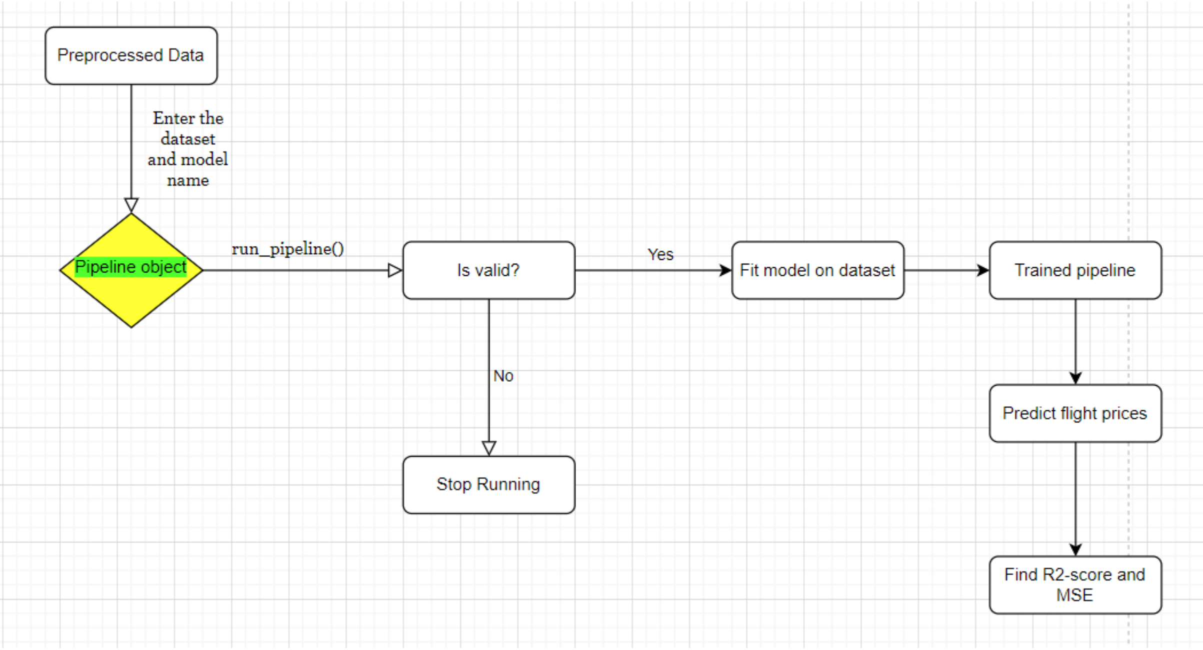
For further clarity of predicted values vs actual values, I made the following scatter plots:




From the above made observations, I chose *Random Forest Regressor*, *CNN* and *Decision Tree Regressor* as the choices to be given to the user for doing his flight price prediction.

# PipeLine

The pipeline used takes *dataset* and *model_name* and gives us the ***r2_score*** and ***MSE*** metrics on the dataset. The flowchart of the working is as follows:




Link to my Github Repo: [Github Repo]