CS6375.002: Flight Fare Prediction

Sri Chanakya Chowdary Babu Yennana and **Apeksha Padaliya**
email: sxy210038@utdallas.edu
email: axp210162@utdallas.edu

Problem Statement

The aim of this project is to develop a machine learning model that can accurately predict the flight fare prices based on various features such as departure and arrival locations, dates of travel, number of stops, airline, and other relevant factors. The model should be able to analyze historical flight data and predict the prices for future flights with a high degree of accuracy. The main objective is to create a tool that can help travelers plan their trips and find the best flight deals, while also assisting airlines in optimizing their pricing strategies.

1 Dataset Description

The dataset is a Flight Fare Prediction Dataset by Machine-Hack. It contains information on flight tickets from various airlines traveling between different cities in India. The dataset has 10682 rows and 11 columns. The columns are:

- Airline: The name of the airline
- Date_of_Journey: The date of the journey
- Source: The source from which the service begins
- Destination: The destination where the service ends
- Route: The route taken by the flight to reach the destination
- Dep_Time: The time when the journey starts from the source
- Arrival_Time: The time when the journey ends at the destination
- Duration: Total duration of the flight
- Total_Stops: Total stops between the source and destination
- Additional_Info: Additional information about the flight
- Price: The price of the ticket

The dataset can be used for predicting the flight fare based on various factors such as date, time, route, stops, etc.

2 Data Analysis

The dataset was loaded using the pandas library which can be observed in the below screenshot.

Loading data

```
df = pd.read_excel('archive/Data_Train.xlsx')
```

```
df.head()
```

	Airline	Date_of_Journey	Source	Destination	Route	Dep_Time	Arrival_Time	Duration	Total_Stops	Additional_Info	Price
0	IndiGo	24/03/2019	Banglore	New Delhi	BLR → DEL	22:20	01:10 22 Mar	2h 50m	non-stop	No info	3897
1	Air India	1/05/2019	Kolkata	Banglore	CCU → IXR → BBI → BLR	05:50	13:15	7h 25m	2 stops	No info	7662
2	Jet Airways	9/06/2019	Delhi	Cochin	DEL → LKO → BOM → COK	09:25	04:25 10 Jun	19h	2 stops	No info	13882
3	IndiGo	12/05/2019	Kolkata	Banglore	CCU → NAG → BLR	18:05	23:30	5h 25m	1 stop	No info	6218
4	IndiGo	01/03/2019	Banglore	New Delhi	BLR → NAG → DEL	16:50	21:35	4h 45m	1 stop	No info	13302

The dataset is loaded as a DataFrame using pandas that will be stored in the variable "**df**". The first 5 rows of the dataset can be displayed using "**df.head()**" line of code.

2.1 Statistical Analysis

The pandas dataframe has many in-built methods wrapped in the object or variable storing the dataset as a dataframe which can be used for various purposes such as analysis, data transformation, exploration, and others. Here we used "**df.info()**" which prints the information about the dataset.

The dataset has a total of 10683 rows of data and 11 columns.

```
In [6]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10683 entries, 0 to 10682
Data columns (total 11 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Airline          10683 non-null  object
 1   Date_of_Journey  10683 non-null  object
 2   Source           10683 non-null  object
 3   Destination      10683 non-null  object
 4   Route            10682 non-null  object
 5   Dep_Time         10683 non-null  object
 6   Arrival_Time     10683 non-null  object
 7   Duration         10683 non-null  object
 8   Total_Stops      10682 non-null  object
 9   Additional_Info  10683 non-null  object
 10  Price            10683 non-null  int64
dtypes: int64(1), object(10)
memory usage: 918.2+ KB
```

As observed, the first 10 columns of the data do not have a lot of null values and are of the ***object*** data type which we are going to transform to data that is in ***int64*** type. The observed number of non-null entries/values of each column/feature are 10682 out of total 10683 for a few columns such as **'Route','Total_Stops'** indicating that there are null values in our data.

To check the total number of null values for each column we can use the following piece of code.

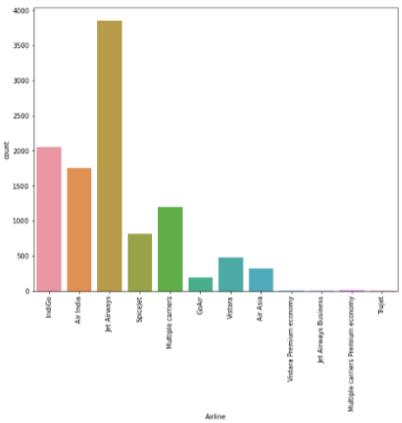
```
In [6]: df.isnull().sum()
```

```
Out[6]: 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
        dtype: int64
```

As we can see there are null values in 'Route' and 'Total_Stops' columns of the dataset. Since the count of null values are very less i.e. one for each column we are going to drop the rows containing null values in the data transformation process.

2.2 Univariate Analysis

```
In [8]: plt.figure(figsize=(10,8))
        sns.countplot(df.Airline)
        plt.xticks(rotation=90)
        plt.show()
```



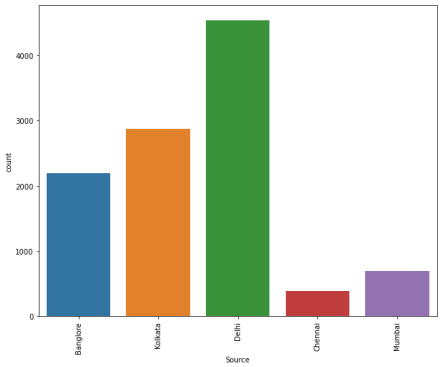
In the above visualization we can observe that a lot of people have flown the '**Jet Airways**' airlines followed by '**Indigo**' and '**Air India**'.

It can be observed that the 4 least preferred airlines are **'Trujet', 'Vistara Premium Economy', 'Jet Airways Business'** and '**Multiple carriers Premium Economy**'.

```
In [24]: for i in df.Airline.unique():
             print(i,':',len(df[df.Airline==i]))
```

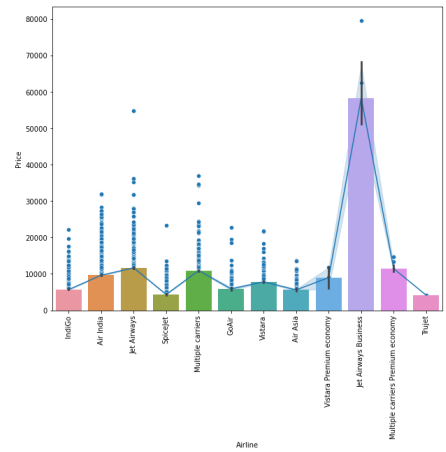
```
IndiGo : 2053
Air India : 1752
Jet Airways : 3849
SpiceJet : 818
Multiple carriers : 1196
GoAir : 194
Vistara : 479
Air Asia : 319
Vistara Premium economy : 3
Jet Airways Business : 6
Multiple carriers Premium economy : 13
Trujet : 1
```

```
In [9]: plt.figure(figsize=(10,8))
        sns.countplot(df.Source)
        plt.xticks(rotation=90)
        plt.show()
```

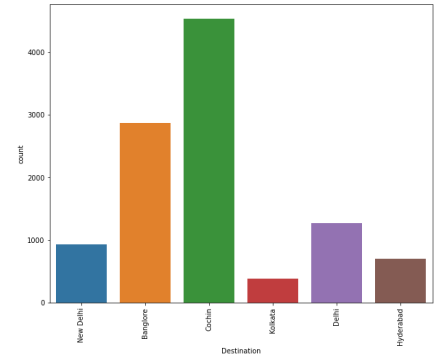


Most visitors come from **Delhi**, with a small number from **Kolkata** and very few from **Chennai**.

```
In [17]: plt.figure(figsize=(10,8))
         sns.lineplot(df['Airline'],df['Price'])
         sns.scatterplot(df['Airline'],df['Price'])
         sns.barplot(df['Airline'],df['Price'])
         plt.xticks(rotation=90)
         plt.show()
```

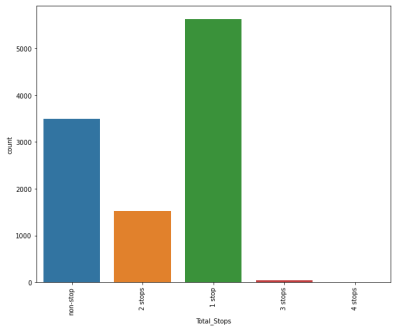



```
In [10]: plt.figure(figsize=(10,8))
         sns.countplot(df.Destination)
         plt.xticks(rotation=90)
         plt.show()
```



Most people have travelled to **Cochin** followed by **Bangalore** and very few have travelled to **Kolkata**.

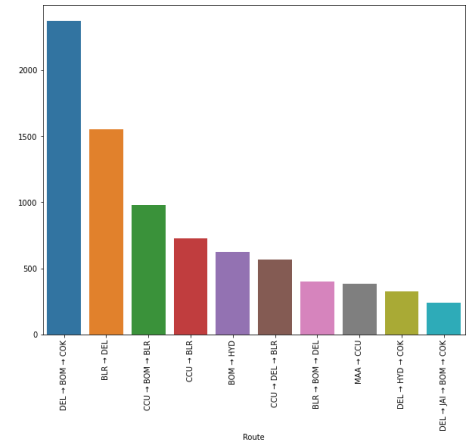
```
In [11]: plt.figure(figsize=(10,8))
         sns.countplot(df.Total_Stops)
         plt.xticks(rotation=90)
         plt.show()
```



It can be observed that most of the people have chosen the airlines with only 1 stop as first preference and preferred non-stop flights second. There could be many reasons for this pattern, one such being the prices for non-stop flights being higher than those with stops.

2.3 Bivariate analysis

```
In [54]: plt.figure(figsize=(10,8))
         sns.barplot(top10_routes.index,top10_routes)
         plt.xticks(rotation=90)
         plt.show()
```



The top 10 flight routes can be observed above with the most popular flight route starting from **Delhi (DEL)** and reaching **Cochin (COK)** with a stop at **Mumbai (BOM)**.

Jet Airways Business is the airline charging the highest cost for a booking and also very less number of people have booked it which can be observed from the above plot. The most affordable airlines are **Spicejet** and **Trujet**.

3 Data pre-processing

As we observed the null values in the data in very small numbers, we dropped the rows containing the null values and stored the data without null values in a dummy variable using the below piece of code.

The **Date_of_Journey** column of the dataset has been transformed to time stamps using the **to_datetime()** method of pandas which can be observed below.

Date and time column transformation

```
In [9]: dates = pd.to_datetime(df.Date_of_Journey)
        dates
```

```
Out[9]: 0       2019-03-24
        1       2019-01-05
        2       2019-09-06
        3       2019-12-05
        4       2019-01-03
                   ...
        10678   2019-09-04
        10679   2019-04-27
        10680   2019-04-27
        10681   2019-01-03
        10682   2019-09-05
        Name: Date_of_Journey, Length: 10683, dtype: datetime64[ns]
```

	Airline	Date_of_Journey	Source	Destination	Route	Dep_Time	Arrival_Time	Duration	Total_Stops	Additional_Info	Price
10681	Vistara	01/03/2019	Banglore	New Delhi	BLR → DEL	11:30	14:10	2h 40m	non-stop	No info	12648
10682	Air India	9/05/2019	Delhi	Cochin	DEL → GOI → BOM → COK	10:55	19:15	8h 20m	2 stops	No info	11753

10682 rows × 11 columns

We are going to split the processed time stamps into two new columns of the dataframe namely '**dep_month**' and '**dep_day**' using the code below.

Dropping unecessary columns

```
In [13]: dummy = dummy.drop(columns=['Date_of_Journey','Route','Dep_Time','Arrival_Time','Additional_Info'])
         dummy
```

Out[13]:

	Airline	Source	Destination	Duration	Total_Stops	Price	dep_month	dep_day	Dep_hour	Dep_minutes	Arr_hour	Arr_minutes
0	IndiGo	Banglore	New Delhi	170	non-stop	3897	3	24	22	20	1	10
1	Air India	Kolkata	Banglore	445	2 stops	7662	1	5	5	50	13	15
2	Jet Airways	Delhi	Cochin	1140	2 stops	13882	9	6	9	25	4	25
3	IndiGo	Kolkata	Banglore	325	1 stop	6218	12	5	18	5	23	30
4	IndiGo	Banglore	New Delhi	285	1 stop	13302	1	3	16	50	21	35
...	...	...	...	...	...	...	...	...	...	...	...	...
10678	Air Asia	Kolkata	Banglore	150	non-stop	4107	9	4	19	55	22	25
10679	Air India	Kolkata	Banglore	155	non-stop	4145	4	27	20	45	23	20
10680	Jet Airways	Banglore	Delhi	180	non-stop	7229	4	27	8	20	11	20
10681	Vistara	Banglore	New Delhi	160	non-stop	12648	1	3	11	30	14	10
10682	Air India	Delhi	Cochin	500	2 stops	11753	9	5	10	55	19	15

10683 rows × 12 columns

```
In [7]: #dummy['start_year']=pd.DatetimeIndex(dates).year
        dummy['dep_month']=pd.DatetimeIndex(dates).month
        dummy['dep_day']=pd.DatetimeIndex(dates).day
        dummy['Date_of_Journey']=dates
        dummy
```

Out[7]:

	Airline	Date_of_Journey	Source	Destination	Route	Dep_Time	Arrival_Time	Duration	Total_Stops	Additional_Info	Price	dep_month	dep_day
0	IndiGo	2019-03-24	Banglore	New Delhi	BLR → DEL	22:20	01:10 22 Mar	2h 50m	non-stop	No info	3897	3	24
1	Air India	2019-01-05	Kolkata	Banglore	CCU → IXR → BBI → BLR	05:50	13:15	7h 25m	2 stops	No info	7662	1	5
2	Jet Airways	2019-09-06	Delhi	Cochin	DEL → LKO → BOM → COK	09:25	04:25 10 Jun	19h	2 stops	No info	13882	9	6
3	IndiGo	2019-12-05	Kolkata	Banglore	CCU → NAG → BLR	18:05	23:30	5h 25m	1 stop	No info	6218	12	5
4	IndiGo	2019-01-03	Banglore	New Delhi	BLR → NAG → DEL	16:50	21:35	4h 45m	1 stop	No info	13302	1	3
...	...	...	...	...	...	...	...	...	...	...	...	...	...
10678	Air Asia	2019-09-04	Kolkata	Banglore	CCU → BLR	19:55	22:25	2h 30m	non-stop	No info	4107	9	4
10679	Air India	2019-04-27	Kolkata	Banglore	CCU → BLR	20:45	23:20	2h 35m	non-stop	No info	4145	4	27
10680	Jet Airways	2019-04-27	Banglore	Delhi	BLR → DEL	08:20	11:20	3h	non-stop	No info	7229	4	27
10681	Vistara	2019-01-03	Banglore	New Delhi	BLR → DEL	11:30	14:10	2h 40m	non-stop	No info	12648	1	3
10682	Air India	2019-09-05	Delhi	Cochin	DEL → GOI → BOM → COK	10:55	19:15	8h 20m	2 stops	No info	11753	9	5

10683 rows × 13 columns

Similarly, we are also going to transform the **Dep_Time** and **Arrival_time** columns to **'Dep_hour','Dep_minutes'** and **'Arr_hour', 'Arr_minutes'** respectively.

```
In [9]: dummy['Dep_hour']=pd.DatetimeIndex(pd.to_datetime(df['Dep_Time'])).hour
        dummy['Dep_minutes']=pd.DatetimeIndex(pd.to_datetime(df['Dep_Time'])).minute
        dummy['Arr_hour']=pd.DatetimeIndex(pd.to_datetime(df['Arrival_Time'])).hour
        dummy['Arr_minutes']=pd.DatetimeIndex(pd.to_datetime(df['Arrival_Time'])).minute
        dummy.head()
```

Out[9]:

	Airline	Date_of_Journey	Source	Destination	Route	Dep_Time	Arrival_Time	Duration	Total_Stops	Additional_Info	Price	dep_month	dep_day	Dep_hour
0	IndiGo	2019-03-24	Banglore	New Delhi	BLR → DEL	22:20	01:10 22 Mar	2h 50m	non-stop	No info	3897	3	24	22
1	Air India	2019-01-05	Kolkata	Banglore	CCU → IXR → BBI → BLR	05:50	13:15	7h 25m	2 stops	No info	7662	1	5	5
2	Jet Airways	2019-09-06	Delhi	Cochin	DEL → LKO → BOM → COK	09:25	04:25 10 Jun	19h	2 stops	No info	13882	9	6	9
3	IndiGo	2019-12-05	Kolkata	Banglore	CCU → NAG → BLR	18:05	23:30	5h 25m	1 stop	No info	6218	12	5	18
4	IndiGo	2019-01-03	Banglore	New Delhi	BLR → NAG → DEL	16:50	21:35	4h 45m	1 stop	No info	13302	1	3	16

We are going to transform the **Duration** column of the data to only minutes by using a custom function as shown below in the screenshot.

Converting duration to minutes

```
In [10]: def process_duration(test):
             ans = 0
             for i in test.split(' '):
                 if 'h' in i:
                     ans+=int(i.split('h')[0]) * 60
                 elif 'm' in i:
                     ans+=int(i.split('m')[0]) * 1
             return ans

         process_duration('2h 30m')
Out[10]: 150

In [11]: dummy['Duration']=dummy['Duration'].apply(process_duration)
         dummy
```

Out[11]:

	Airline	Date_of_Journey	Source	Destination	Route	Dep_Time	Arrival_Time	Duration	Total_Stops	Additional_Info	Price	dep_month	dep_day	De
0	IndiGo	2019-03-24	Banglore	New Delhi	BLR → DEL	22:20	01:10 22 Mar	170	non-stop	No info	3897	3	24	
1	Air India	2019-01-05	Kolkata	Banglore	CCU → IXR → BBI → BLR	05:50	13:15	445	2 stops	No info	7662	1	5	
2	Jet Airways	2019-09-06	Delhi	Cochin	DEL → LKO → BOM → COK	09:25	04:25 10 Jun	1140	2 stops	No info	13882	9	6	
3	IndiGo	2019-12-05	Kolkata	Banglore	CCU → NAG → BLR	18:05	23:30	325	1 stop	No info	6218	12	5	
4	IndiGo	2019-01-03	Banglore	New Delhi	BLR → NAG → DEL	16:50	21:35	285	1 stop	No info	13302	1	3	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Now we drop the unnecessary columns or those features which we don't need for training the model using the below piece of code.

Before splitting the data to train and test we are going to label encode our categorical columns of the data (**Airline, Source, Destination, Total_Stops**) using the Scikitlearn library's predefined '**LabelEncoder()**' for the columns (**Airline, Source, Destination)** and a custom label encoding method using a dictionary for the **Total_Stops** column. We stored the LabelEncoders in a dictionary for easy access based on the column of the data.

We finally save the processed data as a csv file using the below piece of code.

```
In [61]: dummy.to_csv('refined.csv')
```

4 Data Splitting into train and test set

First, we load the refined data using pandas as shown below.

Label Encoding

```
In [14]: encoders = {}
         for i in ['Airline','Source','Destination']:
             le = LabelEncoder()
             dummy[i]=le.fit_transform(dummy[i])
             encoders[i]=le
         dummy.head()
```

Out[14]:

	Airline	Source	Destination	Duration	Total_Stops	Price	dep_month	dep_day	Dep_hour	Dep_minutes	Arr_hour	Arr_minutes
0	3	0	5	170	non-stop	3897	3	24	22	20	1	10
1	1	3	0	445	2 stops	7662	1	5	5	50	13	15
2	4	2	1	1140	2 stops	13882	9	6	9	25	4	25
3	3	3	0	325	1 stop	6218	12	5	18	5	23	30
4	3	0	5	285	1 stop	13302	1	3	16	50	21	35

```
In [16]: stops = {'non-stop':0, '2 stops':2, '1 stop':1, '3 stops':3, '4 stops':4}
         def encode_stops(arr):
             return stops[arr]
         dummy['Total_Stops'] = dummy['Total_Stops'].apply(encode_stops)

In [19]: dummy.head()
```

Out[19]:

	Airline	Source	Destination	Duration	Total_Stops	Price	dep_month	dep_day	Dep_hour	Dep_minutes	Arr_hour	Arr_minutes
0	3	0	5	170	0	3897	3	24	22	20	1	10
1	1	3	0	445	2	7662	1	5	5	50	13	15
2	4	2	1	1140	2	13882	9	6	9	25	4	25
3	3	3	0	325	1	6218	12	5	18	5	23	30
4	3	0	5	285	1	13302	1	3	16	50	21	35

Loading Data

```
In [2]: df = pd.read_csv('refined.csv')
        df = df.iloc[:,1:]
        df
```

Out[2]:

	Airline	Source	Destination	Duration	Total_Stops	Price	dep_month	dep_day	Dep_hour	Dep_minutes	Arr_hour	Arr_minutes
0	3	0	5	170	0	3897	3	24	22	20	1	10
1	1	3	0	445	2	7662	1	5	5	50	13	15
2	4	2	1	1140	2	13882	9	6	9	25	4	25
3	3	3	0	325	1	6218	12	5	18	5	23	30
4	3	0	5	285	1	13302	1	3	16	50	21	35
...	...	...	...	...	...	...	...	...	...	...	...	...
10677	0	3	0	150	0	4107	9	4	19	55	22	25
10678	1	3	0	155	0	4145	4	27	20	45	23	20
10679	4	0	2	180	0	7229	4	27	8	20	11	20
10680	10	0	5	160	0	12648	1	3	11	30	14	10
10681	1	2	1	500	2	11753	9	5	10	55	19	15

10682 rows × 12 columns

Using scikitlearn library we split the data into train and test variables as show below. We get 8545 rows of data for training and 2137 rows of data for testing (80% for training and 20% for testing).

Data Split

```
In [3]: x = df.drop(columns='Price')
        y = df['Price']

In [4]: x_train, x_test, y_train, y_test = train_test_split(
                                            x,
                                            y,
                                            random_state=1234,test_size = 0.20,
                                            shuffle=True
                                            )
        print(x_train.shape)
        print(x_test.shape)
        print(y_train.shape)
        print(y_test.shape)

        (8545, 11)
        (2137, 11)
        (8545,)
        (2137,)
```

5 Model training and analysis

We have trained different models based on different regression based supervised algorithms as listed below:

- Decision Tree Regressor
- Random Forest Regressor
- XGBoost Regressor
- K Nearest Neighbour Regressor
- Extra Trees Regressor
- AdaBoost Regressor

Decision Tree Regresor

```
In [218]: dt = DecisionTreeRegressor()
          dt.fit(x_train, y_train)
          print('Train Score:',df.score(x_train,y_train))
          print('Test Score:',df.score(x_test,y_test))

          Train Score: 0.9717818150899008
          Test Score: 0.7241279904089539

In [219]: y_pred = dt.predict(x_test)
          print('MAE:',mean_absolute_error(y_pred,y_test))
          print('R2_score',r2_score(y_test,y_pred))

          MAE: 1347.4930510060833
          R2_score 0.7225208591337251
```

Random Forest Regresor

```
In [214]: rf = RandomForestRegressor()
          rf.fit(x_train, y_train)
          print('Train Score:',rf.score(x_train,y_train))
          print('Test Score:',rf.score(x_test,y_test))

          Train Score: 0.9513186575229488
          Test Score: 0.8165708335268179

In [215]: y_pred = rf.predict(x_test)
          print('MAE:',mean_absolute_error(y_pred,y_test))
          print('R2_score',r2_score(y_test,y_pred))

          MAE: 1177.8445946111292
          R2_score 0.8165708335268179
```

XGBoost

```
In [223]: xgb = XGBRegressor()
          xgb.fit(x_train, y_train)
          print('Train Score:',xgb.score(x_train,y_train))
          print('Test Score:',xgb.score(x_test,y_test))

          Train Score: 0.9388032504450972
          Test Score: 0.8450509462070098

In [224]: y_pred = xgb.predict(x_test)
          print('MAE:',mean_absolute_error(y_pred,y_test))
          print('R2_score',r2_score(y_test,y_pred))

          MAE: 1166.465082178397
          R2_score 0.8450509462070098
```

KNN

```
In [227]: knn = KNeighborsRegressor()
          knn.fit(x_train,y_train)
          print('Train Score:',knn.score(x_train,y_train))
          print('Test Score:',knn.score(x_test,y_test))
          print()

          Train Score: 0.7053820365092627
          Test Score: 0.5675141565846251

In [229]: y_pred = knn.predict(x_test)
          print('MAE:',mean_absolute_error(y_pred,y_test))
          print('R2_score',r2_score(y_test,y_pred))

          MAE: 1828.4948058025268
          R2_score 0.5675141565846251
```

ExtraTreesRegressor

```
In [207]: etr = ExtraTreesRegressor()
          etr.fit(x_train, y_train)
          print('Train Score:',etr.score(x_train,y_train))
          print('Test Score:',etr.score(x_test,y_test))

          Train Score: 0.9717808035798382
          Test Score: 0.7691292550589371

In [208]: y_pred = etr.predict(x_test)
          print('MAE:',mean_absolute_error(y_pred,y_test))
          print('R2_score',r2_score(y_test,y_pred))

          MAE: 1279.390108329434
          R2_score 0.7691292550589371
```

AdaBoost Regressor

```
In [201]: adb = AdaBoostRegressor()
          adb.fit(x_train, y_train)
          print('Train Score:',adb.score(x_train,y_train))
          print('Test Score:',adb.score(x_test,y_test))

          Train Score: 0.5236350154893887
          Test Score: 0.470446722344

In [202]: y_pred = adb.predict(x_test)
          print('MAE:',mean_absolute_error(y_pred,y_test))
          print('R2_score',r2_score(y_test,y_pred))

          MAE: 2723.8665586101283
          R2_score 0.470446722344
```

6 Hyperparameter Tuning

Since few of the models are performing well on training data and not so well on testing data it could be a sign of overfitting which can be reduced by many methods, one of which is hyper parameters tuning. The hyper parameters were tuned using Grid Search with 5-fold cross validation methodology.

After tuning hyper parameters, the following results were observed where test scores or r2 scores on testing data have increased and train scores have decreased solving the problem of overfitting for some models. A decrease in mean absolute error can also be observed.

Decision Tree Regressor

Tuning

```
In [114]: param_grid = {
              'max_depth': [None, 5, 10],
              'min_samples_split': [3, 4, 5],
              'min_samples_leaf': [1, 2, 3],
              'max_features': ['auto', 'sqrt', 'log2']
          }

          dt_reg = DecisionTreeRegressor(random_state=0)
          fitmodel = GridSearchCV(estimator=dt_reg, param_grid=param_grid, cv=5, n_jobs=-1,verbose = 1)
          fitmodel.fit(x_train, y_train)
          print(fitmodel.best_estimator_, fitmodel.best_params_, fitmodel.best_score_)

          Fitting 5 folds for each of 81 candidates, totalling 405 fits
          DecisionTreeRegressor(max_depth=10, max_features='auto', min_samples_leaf=2,
                                min_samples_split=5, random_state=0) {'max_depth': 10, 'max_features': 'auto', 'min_samples_leaf': 2,
          'min_samples_split': 5} 0.7287079855517497

In [212]: dt = DecisionTreeRegressor(max_depth=10, max_features='auto', min_samples_leaf=2,
                                     min_samples_split=5)
          dt.fit(x_train, y_train)
          print('Train Score:',dt.score(x_train,y_train))
          print('Test Score:',dt.score(x_test,y_test))

          Train Score: 0.8521273258128419
          Test Score: 0.7820190228194326

In [213]: y_pred = dt.predict(x_test)
          print('MAE:',mean_absolute_error(y_pred,y_test))
          print('R2_score',r2_score(y_test,y_pred))

          MAE: 1324.9109534144775
          R2_score 0.7820190228194326
```

AdaBoost Regressor

Tuning

```
In [73]: # Define the parameter grid to search
         param_grid = {
             'base_estimator': [DecisionTreeRegressor(max_features=15),
                                DecisionTreeRegressor(max_depth=10)],
             'n_estimators': [200, 300],
             'learning_rate': [0.1, 0.3]
         }

         # Create an instance of the AdaBoostRegressor
         ada_reg = AdaBoostRegressor()

         # Create a GridSearchCV object
         grid_search = GridSearchCV(estimator=ada_reg, param_grid=param_grid, cv=5, n_jobs=-1, verbose = 1)

         # Fit the GridSearchCV object to the data
         grid_search.fit(x_train, y_train)

         # Print the best parameters and best score
         print("Best parameters: ", grid_search.best_params_)
         print("Best score: ", grid_search.best_score_)

         Fitting 5 folds for each of 8 candidates, totalling 40 fits
         Best parameters:  {'base_estimator': DecisionTreeRegressor(max_features=15), 'learning_rate': 0.3, 'n_estimators': 300}
         Best score:  0.7939170031657092

In [203]: adb = AdaBoostRegressor(base_estimator= DecisionTreeRegressor(max_depth=15), learning_rate= 0.3, n_estimators= 300)
          adb.fit(x_train, y_train)
          print('Train Score:',adb.score(x_train,y_train))
          print('Test Score:',adb.score(x_test,y_test))

          Train Score: 0.9438070312670521
          Test Score: 0.8342911023961097

In [204]: y_pred = adb.predict(x_test)
          print('MAE:',mean_absolute_error(y_pred,y_test))
          print('R2_score',r2_score(y_test,y_pred))

          MAE: 1191.572356606128
          R2_score 0.8342911023961097
```

7 Inferences

	Before optimisation			
	Train Score	Test Score	R2_score	MAE
Decision Tree	0.971	0.72	0.722	1329.18
Random Forest	0.951	0.816	0.816	1177.84
XGBoost	0.938	0.845	0.845	1166.46
KNN	0.705	0.567	0.567	1828.49
Extra Trees	0.971	0.769	0.769	1279.39
AdaBoost	0.538	0.484	0.47	2677.59
	After optimisation			
	Train Score	Test Score	R2_score	MAE
Decision Tree	0.852	0.782	0.782	1329.18
Random Forest	0.92	0.833	0.835	1149.25
XGBoost	0.934	0.837	0.837	1172.73
KNN	0.971	0.543	0.543	1936.84
Extra Trees	0.954	0.816	0.816	1168.8
AdaBoost	0.944	0.832	0.834	1188.22

8 Discussion

Our dataset is of medium difficulty. To reduce overfitting, we did hyper parameters tuning. The hyper parameters were tuned using Grid Search with 5-fold cross validation methodology.

The models performed better after hyper parameter optimization except KNN which can be deduced from the values observed for the metrics r2_score, train and test score and mean absolute error. This can be due to the fact KNN is a good algorithm for datasets with a small number of features, as it relies on finding the closest neighbors in the feature space. However, as the number of features increases, the distance between points becomes less meaningful, and the algorithm may have difficulty finding similar points. This is known as the "curse of dimensionality," and it can cause KNN to perform poorly on datasets with many features. Curse of dimensionality refers to the various challenges and problems that arise when working with high-dimensional data, i.e., datasets with many features or dimensions. It can lead to sparsity (In high-dimensional spaces, data points tend to be far apart from each other, making the space sparse. This sparsity makes it difficult for algorithms to find patterns or relationships among the data points, and can lead to poor model performance), distance measures (In high-dimensional spaces, the difference between the nearest and farthest data points tends to be smaller, making distance-based measures less meaningful. This affects the performance of algorithms that rely on distance measures, such as k-Nearest Neighbors or clustering algorithms), increased computational complexity (As the number of dimensions increases, the computational requirements for processing the data and training models grow exponentially. This increase in computational complexity can make it difficult or infeasible to use certain algorithms on high-dimensional datasets).

In such cases, other algorithms like decision trees or neural networks may be more appropriate for achieving accurate results. It can also be due to overfitting as if the hyperparameter tuning process results in a model that is too complex or specialized to the training data, it may perform worse on the test data. For example, selecting a very small value for k might lead to overfitting, where the model captures noise in the training data and generalizes poorly to new data. The k-NN algorithm has some limitations, such as sensitivity to irrelevant or noisy features and reliance on a good distance metric. If the dataset has these characteristics, even after tuning, the k-NN model might not yield the best performance compared to other algorithms. So, for this dataset dimensionality reduction techniques might give a bit better result with k-NN.

We achieved best accuracy with XGBoost, which stands for Extreme Gradient Boosting and is a scalable, distributed gradient-boosted decision tree (GBDT) machine learning library. It provides parallel tree boosting and is the leading machine learning library for regression, classification, and ranking problems. It can be because of regularization (XGBoost incorporates L1 (Lasso) and L2 (Ridge) regularization techniques to prevent overfitting, which results in more generalized models and improved performance on unseen data), sparsity awareness (XGBoost can handle sparse data efficiently, which makes it suitable for high-dimensional and missing-value datasets), cross-validation (XGBoost has built-in cross-validation functionality that helps in model evaluation and selection), gradient boosting (XGBoost is based on the gradient boosting framework, which iteratively builds weak learners (decision trees) and combines their predictions to form a strong model. By focusing on reducing the error from the previous iteration, gradient boosting can achieve high accuracy even with relatively simple base learners), column block and feature parallelism (XGBoost uses efficient algorithms for finding the best splits in the decision trees, which can lead to more accurate and robust models. It also supports parallel and distributed computing, enabling it to handle large datasets and train models faster), early stopping (XGBoost supports early stopping, which means it can stop the training process when there is no significant improvement in model performance for a specified number of iterations. This helps avoid overfitting and reduces training time), pruning (XGBoost prunes trees during the boosting process by removing branches that do not contribute to the reduction of the loss function, leading to simpler and more accurate models).

For the other models our accuracy increases as we go from decision tree to random forest and then it remains comparable in case of Ada-Boost and XGBoost after hyperparameter tuning.