

# Cab Fare Prediction

By

Ayushman Mishra

# Contents

## Chapters

## Page No.

### 1. Introduction

1.1 Problem Statement ..... 3

1.2 Data ..... 3

### 2. Methodology

2.1 Pre-Processing ..... 5

2.1.1 Feature Engineering .....7

2.1.2 Outlier Analysis ..... 9

2.1.3 Data Visualization ..... 11

2.1.4 Feature Selection ..... 13

2.1.5 Feature Scaling ..... 15

2.2 Model Development ..... 19

2.2.1 Decision Tree ..... 19

2.2.2 Linear Regression ..... 20

2.2.3 Random Forest ..... 21

2.2.4 eXtreme Gradient Boosting ..... 22

### 3. Conclusion

3.1 Model Evaluation ..... 23

3.2 Model Selection ..... 25

## Appendix

## References

# 1.Introduction

The project is about a cab company who has done its pilot project and now they are looking to predict the fare for their future transactional cases. As, nowadays there are number of cab companies like Uber, Ola, Meru Cabs etc. And these cab companies deliver services to lakhs of customers daily. Now it becomes really important to manage their data properly to come up with new business ideas to get best results. In this case, earn most revenues. So, it becomes really important estimate the fare prices accurately.

## 1.1 Problem Statement:

The objective of this project is to predict Cab Fare amount. You are a cab rental start-up company. You have successfully run the pilot project and now want to launch your cab service across the country. You have collected the historical data from your pilot project and now have a requirement to apply analytics for fare prediction. You need to design a system that predicts the fare amount for a cab ride in the city.

## 1.2 Data:

The details of data attributes in the dataset are as follows -

- **pickup\_datetime:** timestamp value indicating when the cab ride started.
- **pickup\_longitude:** float for longitude coordinate of where the cab ride started.
- **pickup\_latitude:** float for latitude coordinate of where the cab ride started.
- **dropoff\_longitude:** float for longitude coordinate of where the cab ride ended.
- **dropoff\_latitude:** float for latitude coordinate of where the cab ride ended.
- **passenger\_count:** an integer indicating the number of passengers in the cab ride.
- **fare\_amount:** fare of the given cab ride.

Here *fare\_amount* is our target variable. All others are our predictors.

## Cab Fare Prediction

	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
0	4.5	2009-06-15 17:26:21 UTC	-73.844311	40.721319	-73.841610	40.712278	1.0
1	16.9	2010-01-05 16:52:16 UTC	-74.016048	40.711303	-73.979268	40.782004	1.0
2	5.7	2011-08-18 00:35:00 UTC	-73.982738	40.761270	-73.991242	40.750562	2.0
3	7.7	2012-04-21 04:30:42 UTC	-73.987130	40.733143	-73.991567	40.758092	1.0
4	5.3	2010-03-09 07:51:00 UTC	-73.968095	40.768008	-73.956655	40.783762	1.0

**Fig. 1.2: Train Data**

	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
0	2015-01-27 13:08:24 UTC	-73.973320	40.763805	-73.981430	40.743835	1
1	2015-01-27 13:08:24 UTC	-73.986862	40.719383	-73.998886	40.739201	1
2	2011-10-08 11:53:44 UTC	-73.982524	40.751260	-73.979654	40.746139	1
3	2012-12-01 21:12:12 UTC	-73.981160	40.767807	-73.990448	40.751635	1
4	2012-12-01 21:12:12 UTC	-73.966046	40.789775	-73.988565	40.744427	1

**Fig. 1.2: Test Data**

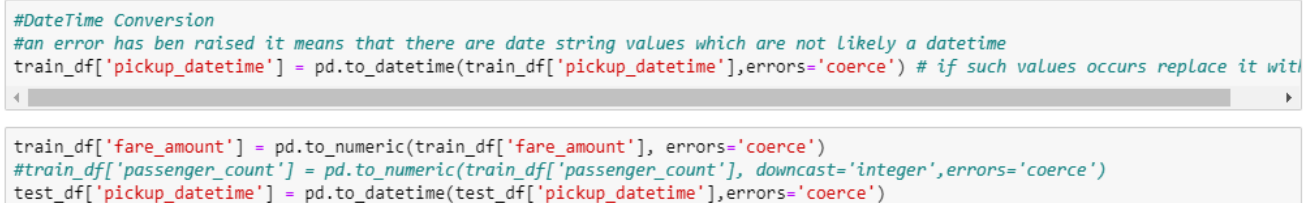
## 2.Methodology

### 2.1 Pre-Processing

Data preprocessing is a data mining technique that involves transforming raw data into an understandable format. Real-world data is often incomplete, inconsistent, and/or lacking in certain behaviors or trends, and is likely to contain many errors. Data preprocessing is a proven method of resolving such issues. Data preprocessing prepares raw data for further processing.

Before Outliers analysis our data will undergo under following process:

- i. Type conversion



```
#DateTime Conversion
#an error has ben raised it means that there are date string values which are not likely a datetime
train_df['pickup_datetime'] = pd.to_datetime(train_df['pickup_datetime'],errors='coerce') # if such values occurs replace it with NaT

train_df['fare_amount'] = pd.to_numeric(train_df['fare_amount'], errors='coerce')
#train_df['passenger_count'] = pd.to_numeric(train_df['passenger_count'], downcast='integer',errors='coerce')
test_df['pickup_datetime'] = pd.to_datetime(test_df['pickup_datetime'],errors='coerce')
```

**Fig.2.1.1 Type Conversion of variables.**

- ii. Data Cleaning

- We observed that many observations have decimal value for passenger\_count which is wrong. Also there are values like 5334 for passenger\_count which is practically impossible.
- In fare\_amount variable we observe that there are negative values and very high values like 54343 which is again practically impossible in cab business. We set the maximum threshold of 453 because only 2 observations have value more than 453. These are the outliers.
- Latitudes range from -90 to 90, and longitudes range from -180 to 180. But in our data we found that there are observation which are not lying under these ranges.
- There are observations whose co-ordinates are zero which is again practically not possible.
- There are observation whose pickup and drop-off coordinates are same.

## Cab Fare Prediction

We have to remove them.

```
#considering the above results from our experiment we will drop corrupted values
train_df.drop(train_df[train_df['fare_amount']<1].index, axis=0,inplace=True)
train_df.drop(train_df[train_df['fare_amount']>453].index, axis=0, inplace=True)
train_df.drop(train_df[train_df['passenger_count']<1].index, axis=0, inplace=True)
train_df.drop(train_df[train_df['passenger_count']>6].index, axis=0, inplace=True)
train_df.drop(train_df[train_df['pickup_latitude']>90].index, axis=0, inplace=True)
for i in ['pickup_longitude','pickup_latitude','dropoff_longitude','dropoff_latitude']:
    train_df.drop(train_df[train_df[i]==0].index, axis=0, inplace=True)

train_df=train_df[np.logical_and(train_df['pickup_longitude'] != train_df['dropoff_longitude'],
                                train_df['pickup_latitude'] != train_df['dropoff_latitude'])]
```

**Fig.2.1.2 Removing Irregular Values.**

We will remove all these corrupted observations.

### iii. Missing values Analysis

	features	count	missing_values_percentage
0	fare_amount	22	0.001419
1	pickup_datetime	1	0.000065
2	pickup_longitude	0	0.000000
3	pickup_latitude	0	0.000000
4	dropoff_longitude	0	0.000000
5	dropoff_latitude	0	0.000000
6	passenger_count	55	0.003548

**Fig.2.1.3 Missing Values Count.**

Following the standards of percentage of missing values we now have to decide to accept a variable or drop it for further operations.

Industry standards ask to follow following standards:

1. Missing value percentage < 30% : Accept the variable
2. Missing value percentage > 30 % : Drop the variable

Since the percentage of missing values is very less we can drop them. However we are not dropping any feature because the missing values percentage is very less.

## Cab Fare Prediction

### 2.1.1 Feature Engineering

We have pickup and drop-off coordinates in the form of latitudes and longitudes. Now we will calculate the distance between them.

The **Haversine** formula calculates the shortest distance between two points on a sphere using their latitudes and longitudes measured along the surface. It is important for use in navigation.

The haversine can be expressed in trigonometric function as:

$$\text{haversine}(\theta) = \sin^2\left(\frac{\theta}{2}\right)$$

The haversine of the central angle (which is  $d/r$ ) is calculated by the following formula:

$$\left(\frac{d}{r}\right) = \text{haversine}(\Phi_2 - \Phi_1) + \cos(\Phi_1)\cos(\Phi_2)\text{haversine}(\lambda_2 - \lambda_1)$$

where  $r$  is the radius of earth(6371 km),  $d$  is the distance between two points,  $\phi_1, \phi_2$  is latitude of the two points and  $\lambda_1, \lambda_2$  is longitude of the two points respectively.

Solving  $d$  by applying the inverse haversine or by using the inverse sine function, we get:

$$d = r \text{hav}^{-1}(h) = 2r \sin^{-1}(\sqrt{h})$$

or

$$d = 2r \sin^{-1} \left( \sqrt{\sin^2\left(\frac{\Phi_2 - \Phi_1}{2}\right) + \cos(\Phi_1)\cos(\Phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)} \right)$$

## Cab Fare Prediction

```
#Since we have given pickup and dropup co ordinates we will find the distance using haversine formula and we will create a new f
def haversine(a):
    lon1=a[0]
    lat1=a[1]
    lon2=a[2]
    lat2=a[3]

    # Calculate the great circle distance between two points
    # on the earth (specified in decimal degrees)

    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    # Radius of earth in kilometers is 6371
    km = 6371 * c
    return km
```

**Fig.2.1.1.1 Calculating Haversine Distance.**

We will sort the pickup\_datetime in ascending order and thus we will get the starting date. Now we know the starting date in our dataset. we will take a date 1 month before our starting date and take the difference in time for each row(pickup\_datetime) and create a new variable and then we will take that new variable into consideration. we will do this for both train and test dataset.

After that we will drop those features from which we have created new features.

Those are:

'pickup\_datetime','pickup\_longitude','pickup\_latitude','dropoff\_longitude','dropoff\_latitude'.

Our Date will look like as follows then:

```
train_df.head()
```

	fare_amount	passenger_count	distance	elapsed_sec
0	4.5	1	1.030764	16991672.0
1	16.9	1	8.450134	34615227.0
2	5.7	2	1.389525	85532591.0
3	7.7	1	2.799270	106887533.0
4	5.3	1	1.999157	40025951.0

**Fig.2.1.1.2 Train Data after feature engineering.**



## Cab Fare Prediction

```
test_df.head()
```

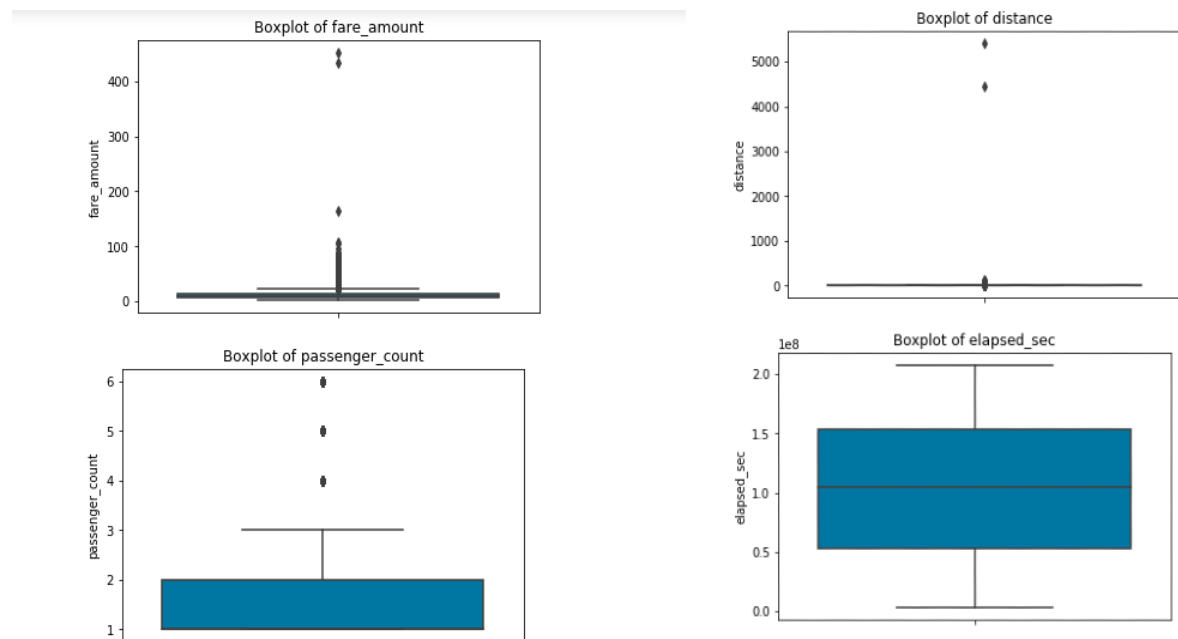
	passenger_count	distance	elapsed_sec
0	1	2.323259	194268995.0
1	1	2.425353	194268995.0
2	1	0.618628	89979715.0
3	1	1.961033	126301223.0
4	1	5.387301	126301223.0

**Fig.2.1.1.3 Test Data after feature engineering.**

## 2.1.2 Outlier Analysis

An *outlier* is an observation that lies an abnormal distance from other values in a random sample from a population. In a sense, this definition leaves it up to the analyst (or a consensus process) to decide what will be considered abnormal. Before abnormal observations can be singled out, it is necessary to characterize normal observations.

Now we plot boxplot for outlier's detection



**Fig.2.1.2.1 Boxplot of all continuous variables.**

## Cab Fare Prediction

In above plots we can see that variables “fare\_amount” and “distance” have outliers.

### Outlier’s Removal

We use KNN Imputation to remove outliers.

```
df = train_df.copy()

for var in ['fare_amount', 'distance', 'elapsed_sec']: # since the passenger count ranges from 1-6 and is fixed we will not consider it
    #Detect and replace with NA
    #Extract quartiles
    q75, q25 = np.percentile(train_df[var], [75, 25])
    #Calculate IQR
    iqr = q75 - q25
    # #Calculate inner and outer fence
    minimum = q25 - (iqr*1.5)
    maximum = q75 + (iqr*1.5)
    # #Replace with NA
    train_df.loc[train_df[var] < minimum, var] = np.nan
    train_df.loc[train_df[var] > maximum, var] = np.nan

# Calculate missing value
missing_val = pd.DataFrame(train_df.isnull().sum())
missing_val
```

	0
fare_amount	1342
passenger_count	0
distance	1333
elapsed_sec	0

```
clean_train_df = pd.DataFrame(KNN(k = 3).fit_transform(train_df), columns=train_df.columns, index=train_df.index)

Imputing row 1/15421 with 0 missing, elapsed time: 65.815
Imputing row 101/15421 with 0 missing, elapsed time: 65.819
Imputing row 201/15421 with 0 missing, elapsed time: 65.824
Imputing row 301/15421 with 0 missing, elapsed time: 65.828
Imputing row 401/15421 with 0 missing, elapsed time: 65.832
Imputing row 501/15421 with 0 missing, elapsed time: 65.838
Imputing row 601/15421 with 0 missing, elapsed time: 65.842
```

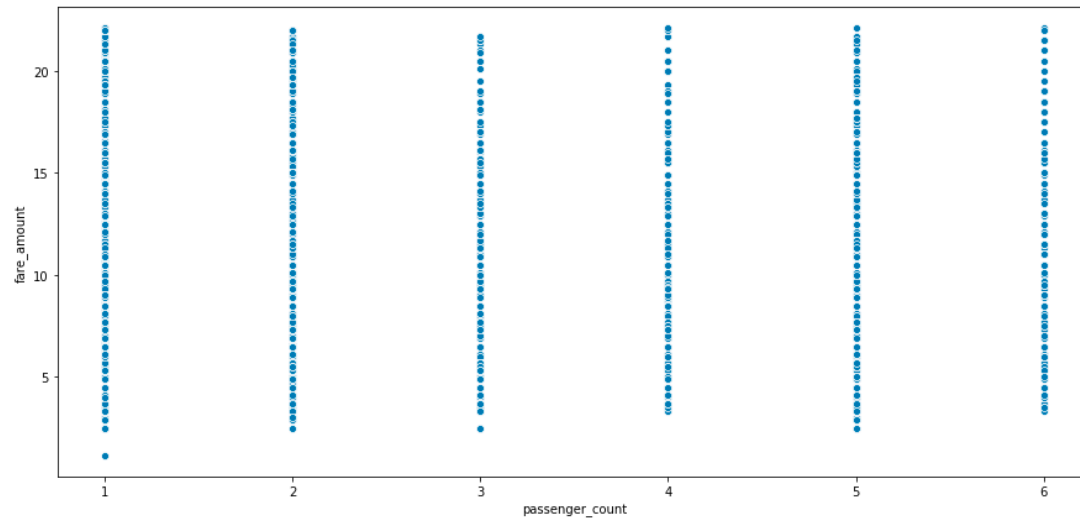
**Fig.2.1.2.2 Outlier Removal Using KNN-Imputation.**

## Cab Fare Prediction

### 2.1.3 Data Visualization

```
#Relationship bwtween passenger_count and fare_amount
plt.figure(figsize=(15,7))
sns.scatterplot(x=train_df['passenger_count'],y=train_df['fare_amount'])
```

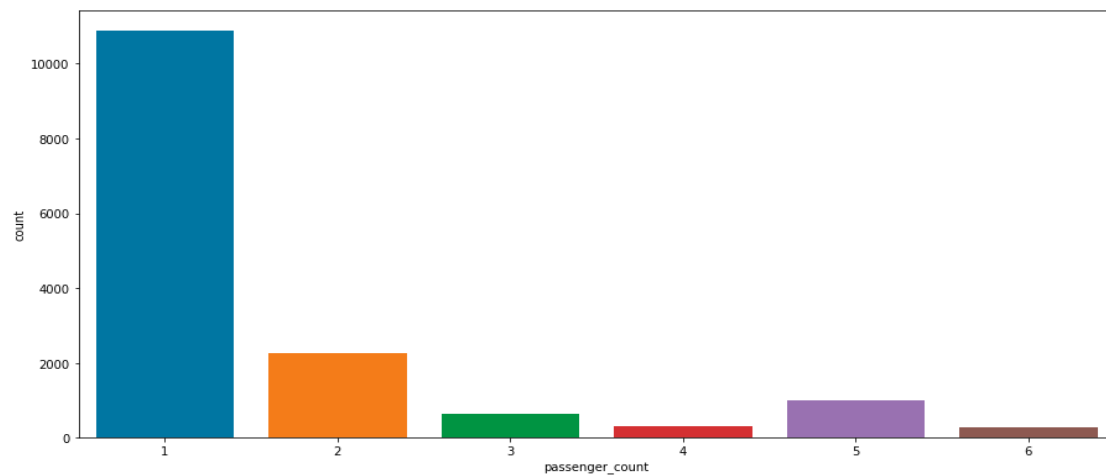
<matplotlib.axes.\_subplots.AxesSubplot at 0x1cd120b0da0>



**Fig.2.1.3.1 Relation between passengen\_count and fare\_amount.**

```
# Count plot on passenger count
plt.figure(figsize=(15,7))
sns.countplot(x="passenger_count", data=clean_train_df)
```

<matplotlib.axes.\_subplots.AxesSubplot at 0x1ccfea9a518>

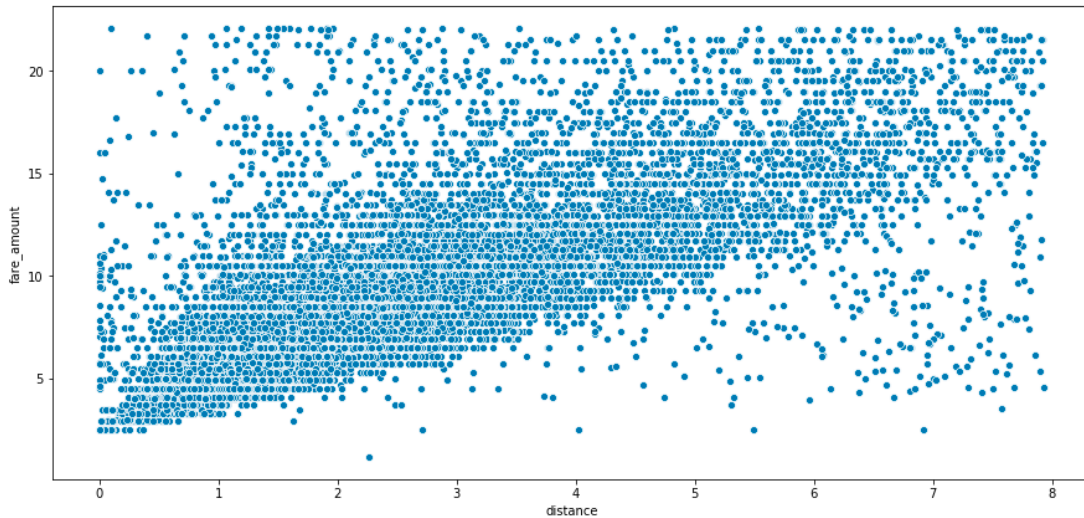


**Fig.2.1.3.2 count plot of passenger count.**

## Cab Fare Prediction

```
#check Relationship between fare_amount and distance
plt.figure(figsize=(15,7))
sns.scatterplot(y=clean_train_df['fare_amount'],x=clean_train_df['distance'])

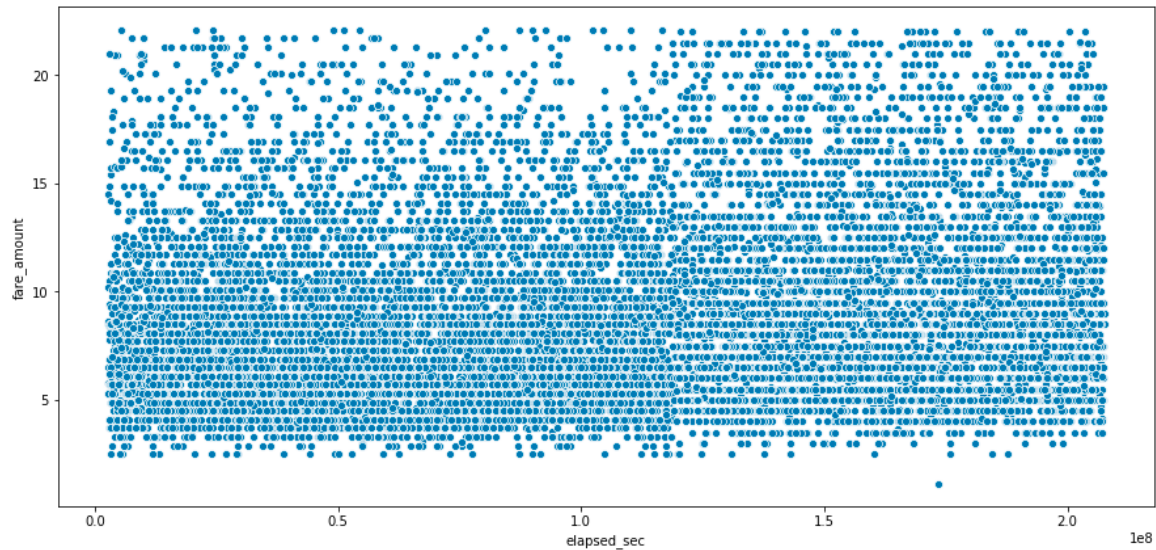
<matplotlib.axes._subplots.AxesSubplot at 0x1ccfeabc048>
```



**Fig.2.1.3.3 Relationship between fare amount and distance.**

```
#check Relationship between fare_amount and elapsed_sec
plt.figure(figsize=(15,7))
sns.scatterplot(y=clean_train_df['fare_amount'],x=clean_train_df['elapsed_sec'])

<matplotlib.axes._subplots.AxesSubplot at 0x1ccfe868240>
```



**Fig.2.1.3.4 Relationship between fare amount and elapsed sec.**

## Cab Fare Prediction

### 2.1.4 Feature Selection

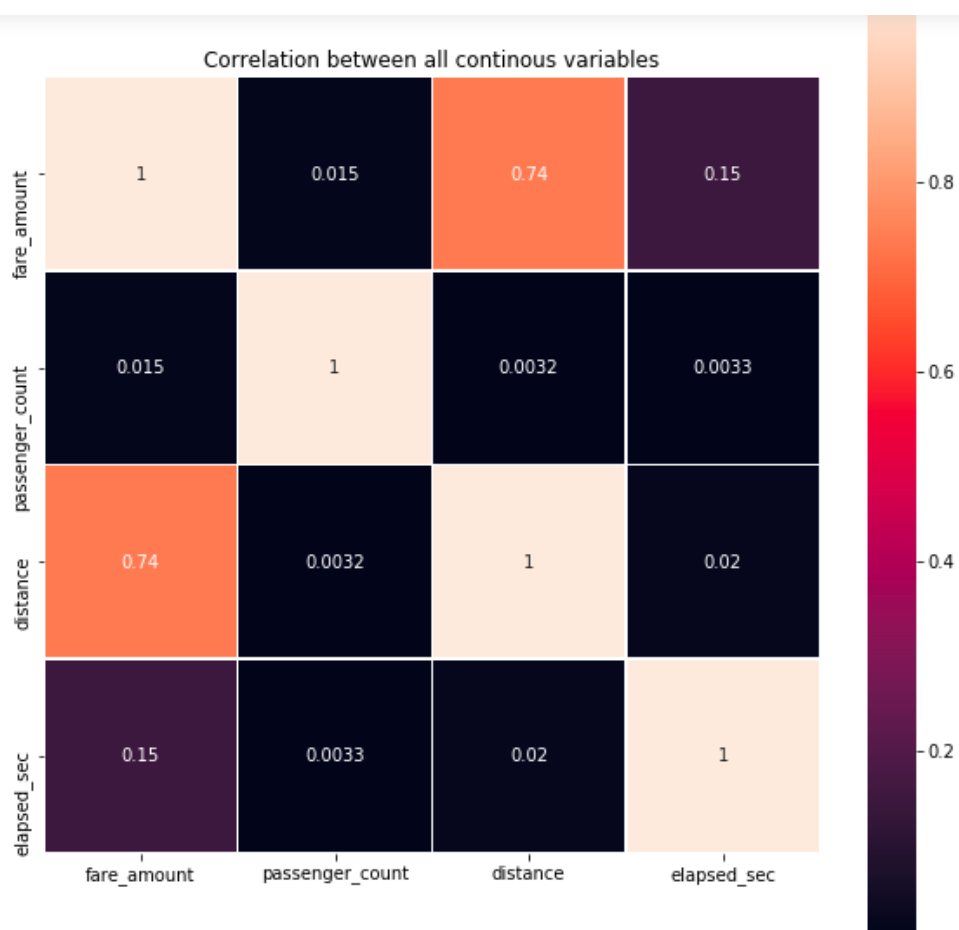
Feature Selection is the process where you automatically or manually select those features which contribute most to your prediction variable or output in which you are interested in.

Having irrelevant features in your data can decrease the accuracy of the models and make your model learn based on irrelevant features.

We perform two test for our feature selection process:

- i. Plotting heat map to check correlation between continuous variables.

First we plot heat map to check correlation.



**Fig.2.1.4.1 Heat Map for correlation.**

## **Cab Fare Prediction**

Each square shows the correlation between the variables on each axis. Correlation ranges from -1 to +1. Values closer to zero means there is no linear trend between the two variables. The closer to 1 the correlation is the more positively correlated they are; that is as one increases so does the other and the closer to -1 the stronger this relationship is. A correlation closer to -1 is similar, but instead of both increasing one variable will decrease as the other increases. The number and darker the color the higher the correlation between the two variables. The plot is also symmetrical about the diagonal since the same two variables are being paired together in those squares.

### **Conclusion:**

Above tests leads to following Conclusion.

- 1) No Negative co relation between variables.
- 2) Strong Positive correlation between target variable and distance
- 3) Little positive correlation between other variables.

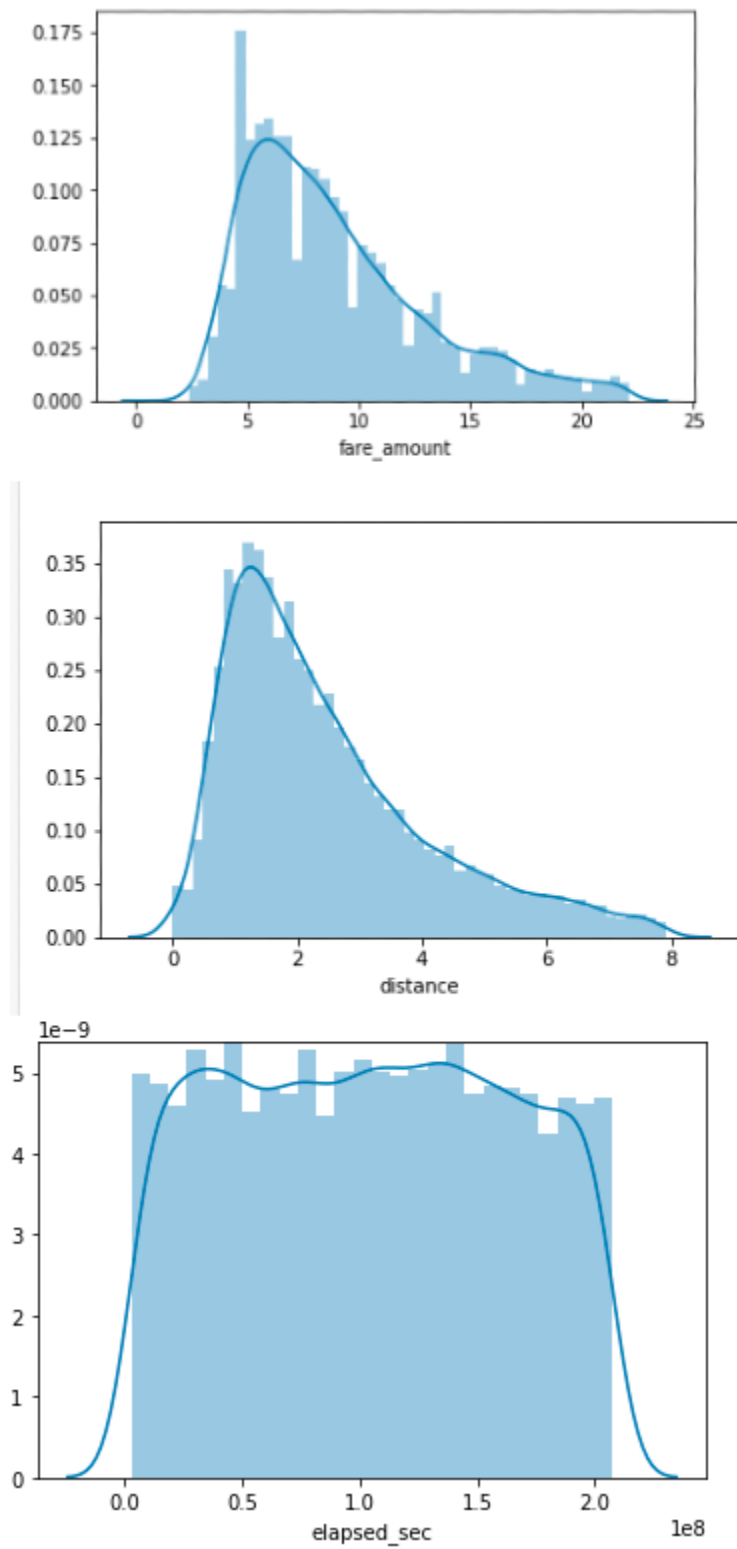
## Cab Fare Prediction

### 2.1.5 Feature Scaling

Data scaling methods are used when we want our variables in data to scale on common ground. It is performed only on continuous variables.

- **Normalization:** Normalization refers to the dividing of a vector by its length. Normalization normalizes the data in the range of 0 to 1. It is generally used when we are planning to use distance method for our model development purpose such as KNN. Normalizing the data improves convergence of such algorithms. Normalization of data scales the data to a very small interval, where outliers can be loosed.
- **Standardization:** Standardization refers to the subtraction of mean from individual point and then dividing by its SD. Z is negative when the raw score is below the mean and Z is positive when above mean. When the data is distributed normally you should go for standardization.
- High variance will affect the accuracy of the model. So, we want to normalize that variance.
- We will perform Normalization on “distance” variable.
- We will perform Normality check
- We will perform skewness test on the continuous Variables.

## Cab Fare Prediction



**Fig.2.1.5.1 Density Plot for Normality Check.**



## Cab Fare Prediction

As we can see that the distribution is not normal for distance and fare\_amount

```
#Lets check the skewness
for var in clean_train_df.columns:
    print("Skewness of", var, ":", clean_train_df[var].skew(), "\n")
```

Skewness of fare\_amount : 1.025278015372948

Skewness of passenger\_count : 2.0712905923822933

Skewness of distance : 1.1126410119951426

Skewness of elapsed\_sec : 0.012727027937341207

**Fig.2.1.5.2 Skewness Test.**

The rule of thumb seems to be:

- 1) If skewness is less than -1 or greater than 1, the distribution is highly skewed.
- 2) If skewness is between -1 and -0.5 or between 0.5 and 1, the distribution is moderately skewed.
- 3) If skewness is between -0.5 and 0.5, the distribution is approximately symmetric.

The skewness for a normal distribution is zero, and any symmetric data should have a skewness near zero. Negative values for the skewness indicate data that are skewed left and positive values for the skewness indicate data that are skewed right.

We will do log transformation on distance and fare\_amount.

```
#since skewness of target variable is high, apply log transform to reduce the skewness-
clean_train_df['fare_amount'] = np.log1p(clean_train_df['fare_amount'])
#since skewness of distance variable is high, apply log transform to reduce the skewness-
clean_train_df['distance'] = np.log1p(clean_train_df['distance'])
```

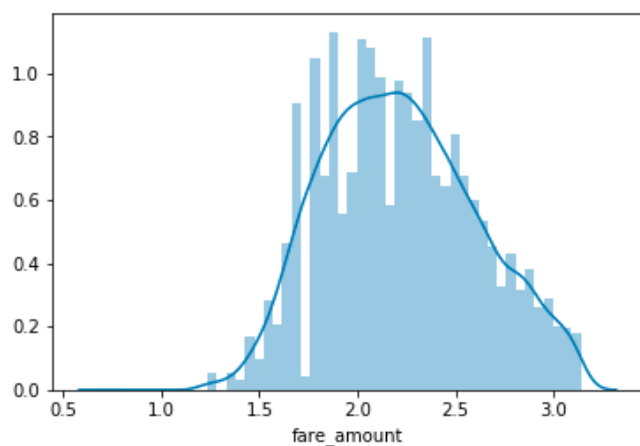
**Fig.2.1.5.3 Log Transformation.**

```
test_df.describe()
```

	passenger_count	distance	elapsed_sec
count	9914.000000	9914.000000	9914.000000
mean	1.671273	1.270879	0.514862
std	1.278747	0.610763	0.272658
min	1.000000	0.000000	0.000000
25%	1.000000	0.832160	0.289004
50%	1.000000	1.168577	0.472238
75%	2.000000	1.618457	0.774353
max	6.000000	4.615081	1.000000

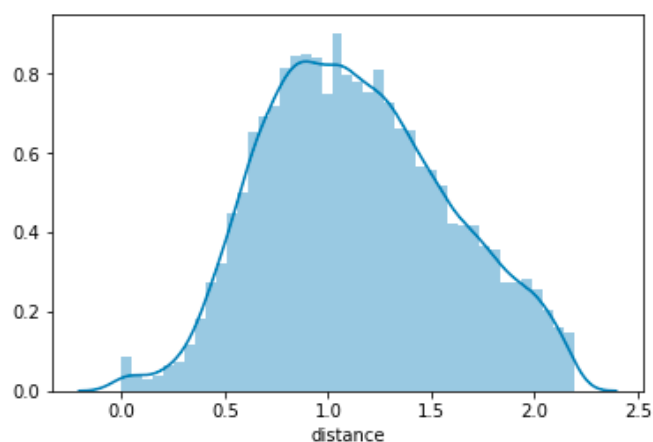
**Fig.2.1.5.4 Data after Log Transformation.**

## Cab Fare Prediction



```
0]: sns.distplot(clean_train_df['distance'])
```

```
0]: <matplotlib.axes._subplots.AxesSubplot at 0x1cd12b9a6a0>
```



**Fig.2.1.5.5 Data after Log Transformation.**

```
for var in ['fare_amount', 'distance']:
    print("Skewness of", var, ":", clean_train_df[var].skew(), "\n")

#All good now
```

```
Skewness of fare_amount : 0.21527327089853524
```

```
Skewness of distance : 0.20009538679423
```

**Fig.2.1.5.6 Skewness after Log Transformation.**

## Cab Fare Prediction

## 2.2 Model Selection

### RandomSearchCV

This uses a random set of hyper parameters. Useful when there are many hyper parameters, so the search space is large. It can be used if you have a prior belief on what the hyper parameters should be.

### 2.2.1 Decision Tree

A decision tree is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements.

A decision tree is a flowchart-like structure in which each internal node represents a “test” on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules.

#### Decision Tree Model

```
DT_model = DecisionTreeRegressor(max_depth=2).fit(train_feature_variables,train_target_variable)
```

```
prediction = DT_model.predict(test_feature_variables)
```

```
prediction = prediction.reshape((3085,1))
```

```
#calculate Error metrics on test data
print("R2 Score:",r2_score(test_target_variable,prediction))
print("Mean Absolute Error:",mean_absolute_error(test_target_variable,prediction))
print("Mean Squared Error:",mean_squared_error(test_target_variable,prediction))
print("Root Mean Squared Error:",np.sqrt(mean_squared_error(test_target_variable,prediction)))
print("MAPE:",mean_absolute_percentage_error(test_target_variable,prediction))
print("Accuracy:",(100 - mean_absolute_percentage_error(test_target_variable,prediction)))
```

```
R2 Score: 0.56502904032977
Mean Absolute Error: 0.19290872262939546
Mean Squared Error: 0.06602757000319132
Root Mean Squared Error: 0.25695830401680214
MAPE: fare_amount      8.780484
dtype: float64
Accuracy: fare_amount    91.219516
dtype: float64
```

**Fig. 2.2.1.1 Decision Tree**

## Cab Fare Prediction

### 2.2.2 Linear Regression

Linear regression is used for finding linear relationship between target and one or more predictors.

#### Linear Regression

```
#Building Linear Regression model
LR_model = LinearRegression().fit(train_feature_variables, train_target_variables)

prediction = LR_model.predict(test_feature_variables)

#calculate Error metrics on test data
print("R2 Score:", r2_score(test_target_variable, prediction))
print("Mean Absolute Error:", mean_absolute_error(test_target_variable, prediction))
print("Mean Squared Error:", mean_squared_error(test_target_variable, prediction))
print("Root Mean Squared Error:", np.sqrt(mean_squared_error(test_target_variable, prediction)))
print("MAPE:", mean_absolute_percentage_error(test_target_variable, prediction))
print("Accuracy:", (100 - mean_absolute_percentage_error(test_target_variable, prediction)))

R2 Score: 0.0048132984386068944
Mean Absolute Error: 0.3196589020711661
Mean Squared Error: 0.15106700376826834
Root Mean Squared Error: 0.38867338958085146
MAPE: fare_amount      14.899055
dtype: float64
Accuracy: fare_amount      85.100945
dtype: float64
```

**Fig.2.2.2.2 Linear Search**

## Cab Fare Prediction

### 2.2.3 Random Forest

Random forest is a supervised learning algorithm. The "forest" it builds, is an ensemble of decision trees, usually trained with the “bagging” method. The general idea of the bagging method is that a combination of learning models increases the overall result.

Put simply: random forest builds multiple decision trees and merges them together to get a more accurate and stable prediction.

#### Random Forest

```

]: RF_model = RandomForestRegressor(n_estimators = 200).fit(train_feature_variables,train_target_variable.values.ravel())

]: prediction = RF_model.predict(test_feature_variables)

]: prediction = prediction.reshape((3085,1))

]: #calculate Error metrics on test data
print("R2 Score:",r2_score(test_target_variable,prediction))
print("Mean Absolute Error:",mean_absolute_error(test_target_variable,prediction))
print("Mean Squared Error:",mean_squared_error(test_target_variable,prediction))
print("Root Mean Squared Error:",np.sqrt(mean_squared_error(test_target_variable,prediction)))
print("MAPE:",mean_absolute_percentage_error(test_target_variable,prediction))
print("Accuracy:",(100 - mean_absolute_percentage_error(test_target_variable,prediction)))

R2 Score: 0.5965764851516375
Mean Absolute Error: 0.18113399117552217
Mean Squared Error: 0.06123874197895518
Root Mean Squared Error: 0.24746462773284422
MAPE: fare_amount    8.272189
dtype: float64
Accuracy: fare_amount    91.727811
dtype: float64

```

**Fig.2.2.3.3 Random Forest**

## Cab Fare Prediction

### 2.2.4 XGBoost

**XGBoost** is an optimized distributed gradient boosting library designed to be highly **efficient**, **flexible** and **portable**. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way.

```
# Create the random grid
para = {'n_estimators': range(100,500,100),
        'max_depth': range(3,10,1),
        'reg_alpha': np.logspace(-4, 0, 50),
        'subsample': np.arange(0.1,1,0.2),
        'colsample_bytree': np.arange(0.1,1,0.2),
        'colsample_bylevel': np.arange(0.1,1,0.2),
        'colsample_bynode': np.arange(0.1,1,0.2),
        'learning_rate': np.arange(.05, 1, .05)}

# Instantiate a Decision Forest classifier: Forest
Xgb = XGBRegressor()

# Instantiate the gridSearchCV object: Forest_cv
xgb_cv = RandomizedSearchCV(Xgb, para, cv=5)

xgb_cv.fit(X,y)
print("Tuned Xgboost Parameters: {}".format(xgb_cv.best_params_))
print("Best score is {}".format(xgb_cv.best_score_))

Tuned Xgboost Parameters: {'subsample': 0.30000000000000004, 'reg_alpha': 0.013257113655901081, 'n_estimators': 300, 'max_depth': 3, 'learning_rate': 0.1, 'colsample_bytree': 0.5000000000000001, 'colsample_bynode': 0.5000000000000001, 'colsample_bylevel': 0.1}
Best score is 0.6299374738481539

Xgb = XGBRegressor(subsample=0.30000000000000004, reg_alpha=0.013257113655901081, n_estimators=300, max_depth=3, learning_rate=0.1, cv=5)

# Fit the regressor to the data
Xgb.fit(train_feature_variables, train_target_variable)
```

**Fig.2.2.4.1 XgBoosting**

```
prediction = Xgb.predict(test_feature_variables)

prediction = prediction.reshape((3085,1))

#calculate Error metrics on test data
print("R2 Score:", r2_score(test_target_variable, prediction))
print("Mean Absolute Error:", mean_absolute_error(test_target_variable, prediction))
print("Mean Squared Error:", mean_squared_error(test_target_variable, prediction))
print("Root Mean Squared Error:", np.sqrt(mean_squared_error(test_target_variable, prediction)))
print("MAPE:", mean_absolute_percentage_error(test_target_variable, prediction))
print("Accuracy:", (100 - mean_absolute_percentage_error(test_target_variable, prediction)))

R2 Score: 0.6405187604035649
Mean Absolute Error: 0.16716349710968018
Mean Squared Error: 0.05456840780883999
Root Mean Squared Error: 0.2335988180809997
MAPE: fare_amount 7.61149
dtype: float64
Accuracy: fare_amount 92.38851
dtype: float64
```

**Fig.2.2.4.2 XgBoosting**

## 3. Conclusion

### 3.1 Model Evaluation

The dependent variable in our model is a continuous variable i.e., Count of bike rentals. Hence the models that we choose are Linear Regression, Decision Tree, Random Forest and Gradient Boosting.

The Following error metrics are chosen for the problem statement:

i. **Root Mean Squared Error:**

RMSE is a quadratic scoring rule that also measures the average magnitude of the error. It's the square root of the average of squared differences between prediction and actual observation.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2}$$

ii. **Mean Absolute Error:**

MAE measures the average magnitude of the errors in a set of predictions, without considering their direction. It's the average over the test sample of the absolute differences between prediction and actual observation where all individual differences have equal weight.

$$\text{MAE} = \frac{1}{n} \sum_{j=1}^n |y_j - \hat{y}_j|$$

## Cab Fare Prediction

### iii. R – Square:

R-squared ( $R^2$ ) is a statistical measure that represents the proportion of the variance for a dependent variable that's explained by an independent variable or variables in a regression model. Whereas correlation explains the strength of the relationship between an independent and dependent variable, R-squared explains to what extent the variance of one variable explains the variance of the second variable. So, if the  $R^2$  of a model is 0.50, then approximately half of the observed variation can be explained by the model's inputs.

$$R^2 = 1 - \frac{\text{Unexplained Variation}}{\text{Total Variation}}$$

### iv. Mean Absolute Percentage Error (MAPE):

The mean absolute percentage error (MAPE) is a statistical measure of how accurate a forecast system is. It measures this accuracy as a percentage, and can be calculated as the average absolute percent error for each time period minus actual values divided by actual values. Where  $A_t$  is the actual value and  $F_t$  is the forecast value, this is given by:

$$M = \frac{1}{n} \sum_{t=1}^n \left| \frac{A_t - F_t}{A_t} \right|$$

The mean absolute percentage error (MAPE) is the most common measure used to forecast error, and works best if there are no extremes to the data (and no zeros).

### v. Accuracy:

Accuracy =  $1 - \text{MAPE}$  or  $100 - \text{MAPE}$  (if it is in percentage).



## Cab Fare Prediction

### 3.2 Model Selection

We will select that model which has low MAPE or high accuracy and whose r-squared value is high.

In our case,

**XgBoosting Model is providing best results having R-squared = 0.65 and Accuracy = 92.388%**

## Appendix – R Code

```
train = read.csv("train_cab.csv", header = T, na.strings = c(" ", "", "NA"))
test = read.csv("test.csv")
# structure of data
```

**Fig.A.1: Reading data**

```
# Changing the data types of variables
train$fare_amount = as.numeric(as.character(train$fare_amount))
train$passenger_count = as.integer(train$passenger_count)
train$pickup_datetime = as.POSIXct(train$pickup_datetime, format="%Y-%m-%d %H:%M:%S", tz="UTC")
test$pickup_datetime = as.POSIXct(test$pickup_datetime, format="%Y-%m-%d %H:%M:%S", tz="UTC")
```

**Fig.A.2: Conversion of Variables Type**

```
# fare amount cannot be less than one and considering 453 as max because only 2 observations are there greater than 453
# passenger count range 1-6
# Latitudes range from -90 to 90, and longitudes range from -180 to 180.
train=subset(train, !(train$fare_amount<1))
train=subset(train,!(train$fare_amount>453))
train=subset(train,!(train$passenger_count<1))
train=subset(train,!(train$passenger_count>6))
sum(train$pickup_latitude>90) #..1
sum(train$pickup_latitude < (-90))#..0
sum(train$pickup_longitude>180) #..0
sum(train$pickup_longitude<(-180)) #..0

sum(train$dropoff_latitude>90) #..0
sum(train$dropoff_latitude < (-90))#..0
sum(train$dropoff_longitude>180) #..0
sum(train$dropoff_longitude<(-180)) #..0

train=subset(train,!(train$pickup_latitude>90))
#Zero degrees latitude is the line designating the Equator and divides the Earth into two equal hemispheres (north and south). Zero degrees longitude
for(i in list('pickup_longitude','pickup_latitude','dropoff_longitude','dropoff_latitude')){
  print(paste0(i," equal to 0 = ",sum(train[[i]]==0)))
}

#removing rows
for(i in list('pickup_longitude','pickup_latitude','dropoff_longitude','dropoff_latitude')){
  train=subset(train,!(train[[i]]==0))
}

#eliminate rows where the pickup and drop location points are same
train=subset(train,!(train$pickup_longitude==train$dropoff_longitude & train$pickup_latitude==train$dropoff_latitude))
```

**Fig.A.3: Data cleaning.**

## Cab Fare Prediction

```
# checking for missing values.
sum(is.na(train))
sum(is.na(test))
train=na.omit(train)#remove missing values
```

**Fig.A.4: Missing values analysis.**

```
#calculate distance for train
train$dist = distHaversine(cbind(train$pickup_longitude,train$pickup_latitude),cbind(train$dropoff_longitude,train$dropoff_latitude))
#the output is in metres, change it to kms
train$dist=as.numeric(train$dist)/1000

#calculate distance for test
test$dist = distHaversine(cbind(test$pickup_longitude,test$pickup_latitude),cbind(test$dropoff_longitude,test$dropoff_latitude))
#the output is in metres, change it to kms
test$dist=as.numeric(test$dist)/1000

#Now we know the starting date in our dataset. we will take a date 1 month before our starting date and take the difference for each row and create
sort(train$pickup_datetime,decreasing = FALSE)
reference_date = as.POSIXct("2008-12-01 01:31:49",format="%Y-%m-%d %H:%M:%S",tz="UTC")
#reset rownames
row.names(train) <- NULL

train$elapsed_sec = difftime(train$pickup_datetime,reference_date,units = "secs")
test$elapsed_sec = difftime(test$pickup_datetime,reference_date,units = "secs")

train$elapsed_sec = as.numeric(as.character(train$elapsed_sec))
test$elapsed_sec = as.numeric(as.character(test$elapsed_sec))
```

**Fig.A.5: Feature Engineering.**

```
##Outliers Analysis
for(i in 1:length(continuous_variables)){
  assign(paste0("gn",i), ggplot(data = train, aes_string(x = "fare_amount", y = continuous_variables[i]))+
    stat_boxplot(geom = "errorbar",width = 0.5)+
    geom_boxplot(outlier.colour = 'red',fill='grey',outlier.shape = 18,outlier.size = 4)+
    labs(y=continuous_variables[i],x='cnt')+
    ggtitle(paste("Box plot of",continuous_variables[i])))
}

gridExtra::grid.arrange(gn1,gn2,ncol=2)
gridExtra::grid.arrange(gn3,gn4,ncol=2)

df_train = train

##Treating outliers
outliers_vars = c('fare_amount','dist', 'elapsed_sec')
for(i in outliers_vars){
  temp = df_train[,i][df_train[,i] %in% boxplot.stats(df_train[,i])$out]
  df_train[,i][df_train[,i] %in% temp] = NA
}

# check for missing values
apply(df_train,2, function(x){ sum(is.na(x))})

#we found 1342 outliers in fare_amount and 1334 in dist
df_train = knnImputation(data = df_train, k = 3)
```

**Fig.A.6: Outliers Analysis.**

## Cab Fare Prediction

```
#Let's Normalize the data
df_train$elapsed_sec = (df_train$elapsed_sec - min(df_train$elapsed_sec)) / (max(df_train$elapsed_sec) - min(df_train$elapsed_sec))
test$elapsed_sec = (test$elapsed_sec - min(test$elapsed_sec)) / (max(test$elapsed_sec) - min(test$elapsed_sec))

#Let's perform log transformation to reduce skewness
df_train$dist = log1p(df_train$dist)
df_train$fare_amount = log1p(df_train$fare_amount)
test$dist = log1p(test$dist)
```

**Fig.A.7: Normalization and log Transformation.**

```
####splitting the data
set.seed(12345)
train_index = createDataPartition(df_train$fare_amount, p = 0.8, list=FALSE)
new_train = df_train[train_index,]
new_test = df_train[-train_index,]
```

**Fig.A.8: Splitting data into train, test.**

```
##### Linear Regression #####
# Linear Regression
linear_regressor = lm(fare_amount ~.,data = new_train)
summary(linear_regressor)
pred = predict(linear_regressor,new_test[,1])
regr.eval(test[,1],preds = pred)

#   mae      mse      rmse     mape
#1.2412741  2.0523051  1.4325868  0.9565344

#calculate R-Squared value
rsq(fitobj = linear_regressor,adj = TRUE,data = new_train)
#0.732751
```

---

**Fig.A.9: Linear Regression Model.**

## Cab Fare Prediction

```
### Decision Tree
tree = rpart(fare_amount ~ ., data=new_train, method = "anova")
summary(tree)

pred_dt = predict(tree, new_test[, -1])

regr.eval(new_test[, 1], preds = pred_dt)

#      mae      mse      rmse      mape
#0.16549514  0.04579052  0.21398719  0.07753618

rss_dt = sum((pred_dt - new_test$fare_amount) ^ 2)
tss_dt = sum((new_test$fare_amount - mean(new_test$fare_amount)) ^ 2)
rsq_dt = 1 - rss_dt/tss_dt
#0.7000909
```

**Fig.A.10: Decision Tree Model.**

```
##Random Forest
rf_model = randomForest(fare_amount ~., data=new_train, importance = TRUE, ntree=500)
summary(rf_model)

pred_rm = predict(rf_model, new_test[, -1])

regr.eval(new_test[, 1], preds = pred_rm)
#      mae      mse      rmse      mape
# 0.16099518  0.04287104  0.20705322  0.07593300

# calculate R-Square value
rss_rf = sum((pred_rm - new_test$fare_amount) ^ 2)
tss_rf = sum((new_test$fare_amount - mean(new_test$fare_amount)) ^ 2)
rsq_rf = 1 - rss_rf/tss_rf
#0.7192123
```

**Fig.A.11: Random Forest Model.**

## Cab Fare Prediction

```
##XgBoost

train_data_matrix = as.matrix(sapply(new_train[-1],as.numeric))
test_data_matrix = as.matrix(sapply(new_test[-1],as.numeric))

xgb = xgboost(data = train_data_matrix,label = new_train$fare_amount, nrounds = 13,verbose = TRUE)

pred_xgb = predict(xgb,test_data_matrix)

regr.eval(new_test[,1],preds = pred_xgb)
#   mae      mse      rmse     mape
#0.14578432  0.03761938  0.19395717  0.06733712

#calculate R-Square value
rss_xgb = sum((pred_xgb - new_test$fare_amount) ^ 2)
tss_xgb = sum((new_test$fare_amount - mean(new_test$fare_amount)) ^ 2)
rsq_xgb = 1 - rss_xgb/tss_xgb
#0.7536085
```

**Fig.A.12: XGboost Model.**

## References

- 1) <https://machinelearningmastery.com/how-to-tune-algorithm-parameters-with-scikit-learn/>
- 2) <https://www.r-bloggers.com/great-circle-distance-calculations-in-r/>
- 3) <https://www.analyticsvidhya.com/blog/2016/01/xgboost-algorithm-easy-steps/>
- 4) <http://benalexkeen.com/gradient-boosting-in-python-using-scikit-learn/>
- 5) <https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/>
- 6) <https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74> .
- 7) <https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f>
- 8) <https://campus.datacamp.com/courses/supervised-learning-with-scikit-learn/fine-tuning-your-model?ex=10>.
- 9) <https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/>
- 10) <https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/>

Thank You