Prediction of Cab fare amount

Debayan Chakraborty

24th October 2019

Contents

Chapter 1

Introduction

1.1 Problem Statement

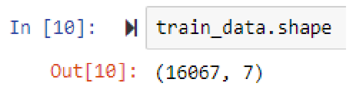
The objective of this project is to predict Cab Fare amount.
We are a cab rental start-up company. We have successfully run the pilot project and now want to launch your cab service across the country. We have collected the historical data from our pilot project and now have a requirement to apply analytics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

1.2 Data

The goal is to build regression models which will predict the fare amount of cab ride in the city with the help of given independent variables. Here in our case our company has provided a training data containing historical records of pilot project which we need for building our model and a test dataset where we need to test our model and predict the dependent variable. We need to go through each variable of it to understand and for better functioning.

For analysis of the given datasets and prediction of the target variable, we have chosen “Python” as our coding tool for data analysis and its results have been articulated in this project report.

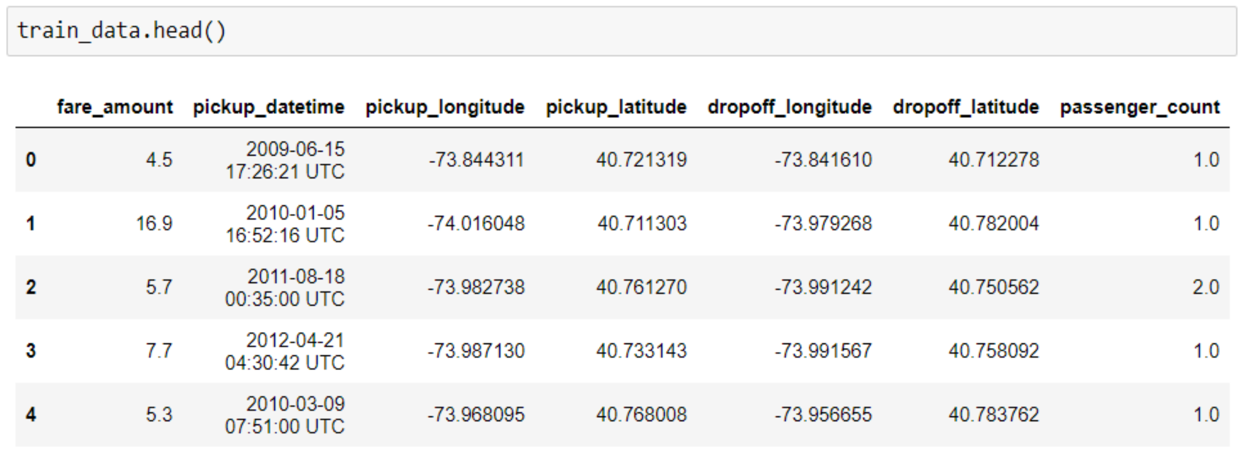
Structure of training dataset provided: 16067 observations & 7 Columns



```
In [10]:  train_data.shape

  Out[10]: (16067, 7)
```

Missing Values: Present

Table 1.1: “Train_cab.csv”



```
train_data.head()
```

	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

The details of variables in the dataset are mentioned as follows:

The dataset contains continuous as well as categorical variables.

a) **fare_amount:** Value of the fare amount.

b) **pickup_datetime:** Timestamp data indicating when the cab ride started.

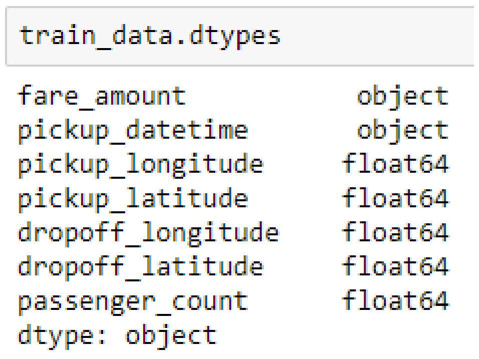
c) **pickup_longitude :** Longitude coordinate of where the cab ride started.

d) **pickup_latitude:** Latitude coordinate of where the cab ride started.

e) **dropoff_longitude:** Longitude coordinate of where the cab ride ended.

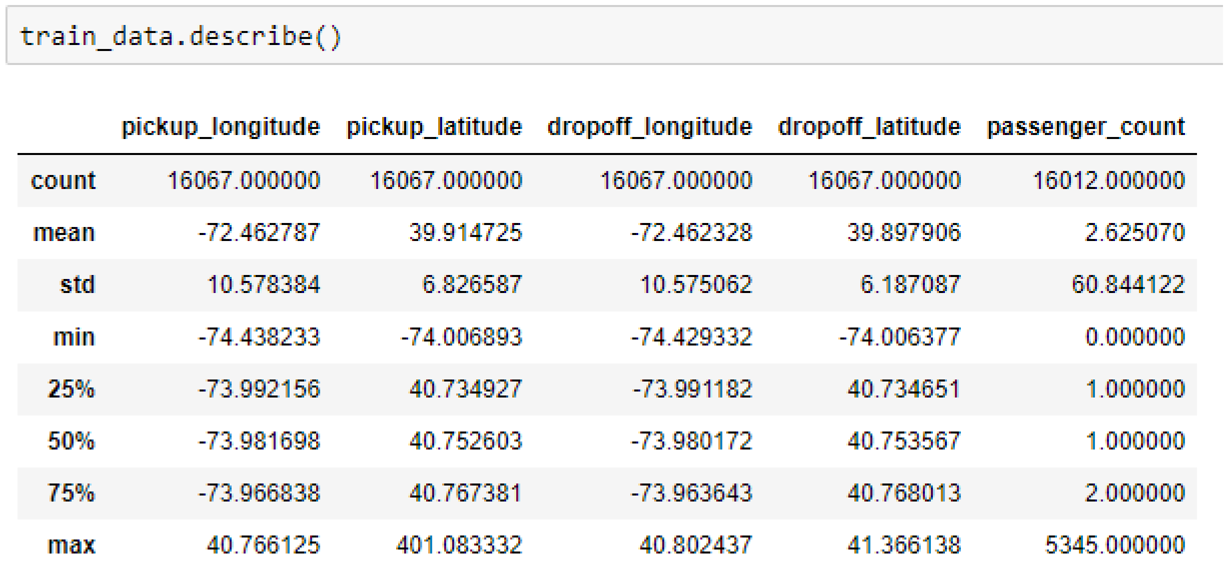
f) **dropoff_latitude:** Latitude coordinate of where the cab ride ended.

g) **passenger**_count: Data indicating the number of passengers in the cab ride.



```
train_data.dtypes
```

```
fare_amount           object
pickup_datetime       object
pickup_longitude     float64
pickup_latitude      float64
dropoff_longitude    float64
dropoff_latitude     float64
passenger_count      float64
dtype: object
```



```
train_data.describe()
```

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	16067.000000	16067.000000	16067.000000	16067.000000	16012.000000
mean	-72.462787	39.914725	-72.462328	39.897906	2.625070
std	10.578384	6.826587	10.575062	6.187087	60.844122
min	-74.438233	-74.006893	-74.429332	-74.006377	0.000000
25%	-73.992156	40.734927	-73.991182	40.734651	1.000000
50%	-73.981698	40.752603	-73.980172	40.753567	1.000000
75%	-73.966838	40.767381	-73.963643	40.768013	2.000000
max	40.766125	401.083332	40.802437	41.366138	5345.000000

Chapter 2

Methodology

2.1 Pre-Processing

Any predictive modelling requires that we look at the data before we start modelling. However, in data mining terms looking at data refers to so much more than just looking. Looking at data refers to exploring the data, cleaning the data as well as visualizing the data through graphs and plots. This is often called as **Exploratory Data Analysis**. To start this process, we will first convert the data variables into required datatypes and visualize the data of continuous variables using histogram and categorical variables using bar charts. Next feature engineering operations will include steps viz. missing value analysis, outlier analysis and correlation analysis.

2.1.1 Primary Visualizations

a) Histogram of continuous variables:

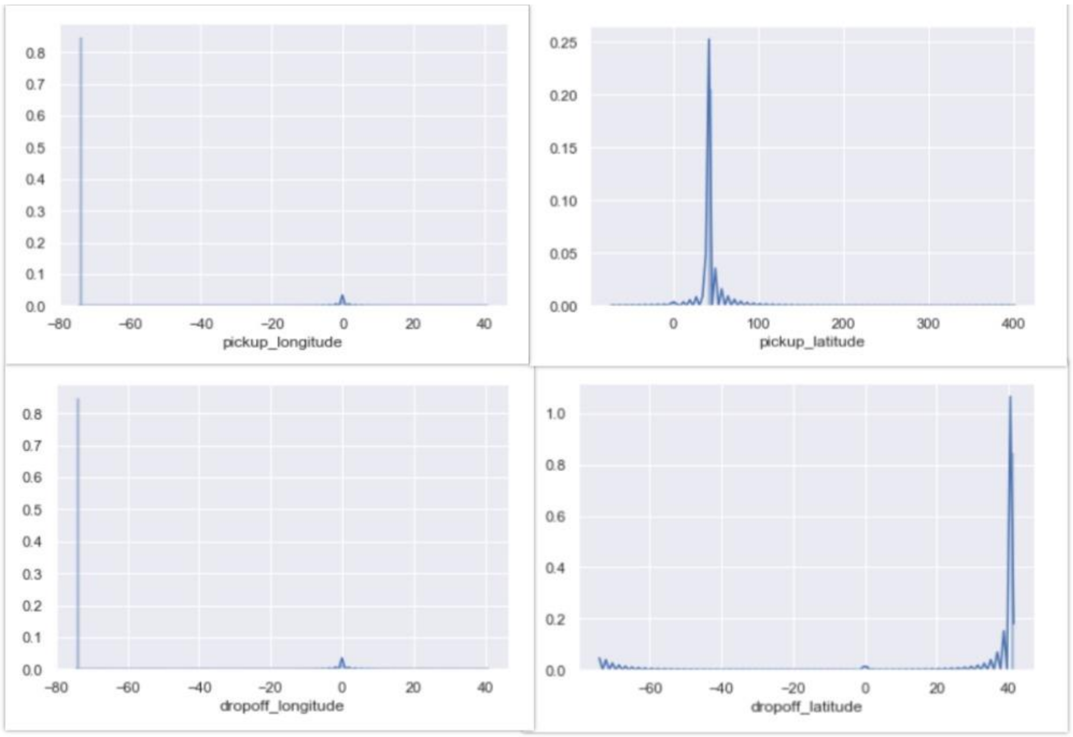


Fig. 1.1

In fig. 1.1 we have plotted the data distribution of independent continuous variables using distplot visualization under seaborn library of Python.

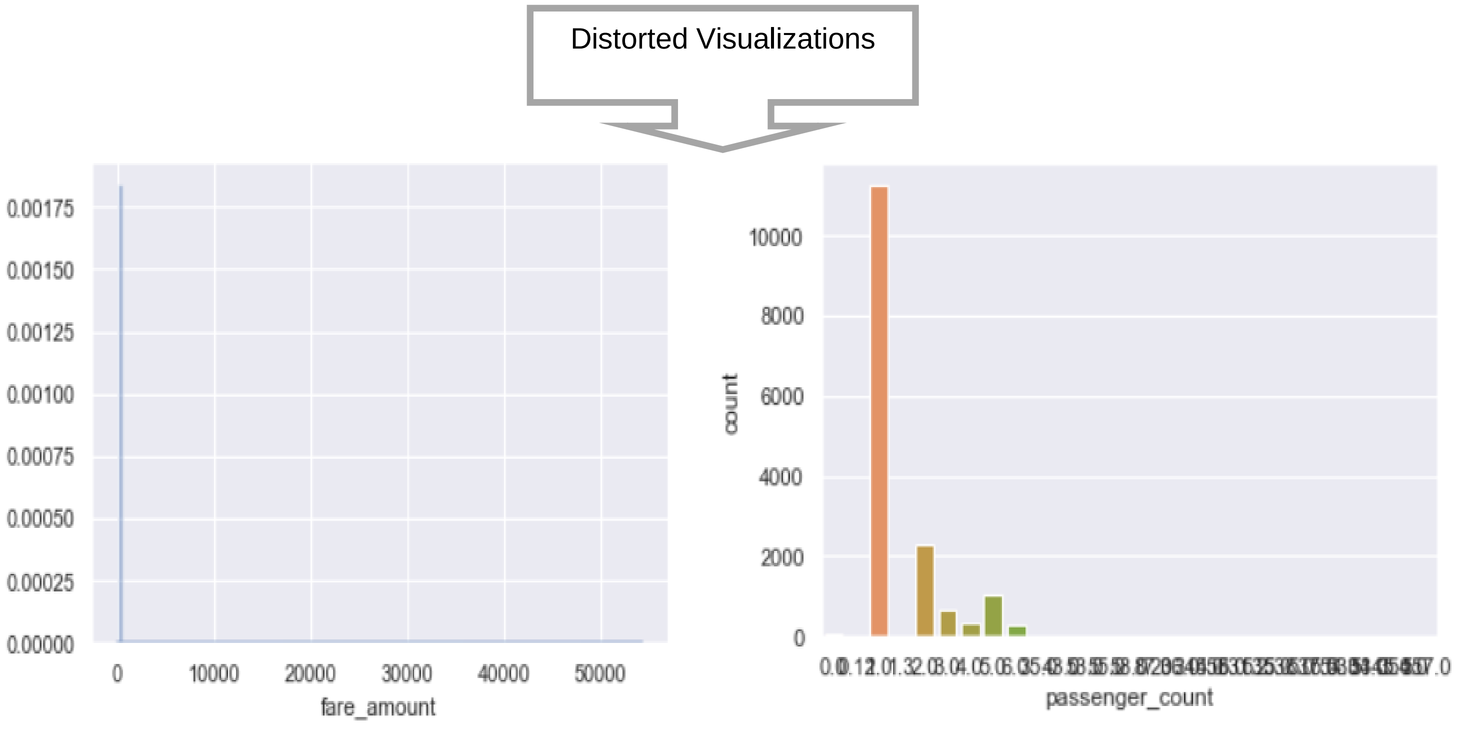


Fig. 1.2

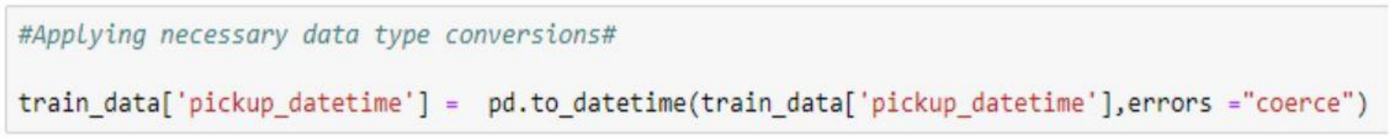
*In fig. 1.2 we have plotted the data distribution of other two independent variables, but the **visualizations obtained were rather distorted due to the presence of outliers in these variables.** We will be plotting a clean visualization free from outliers in section 2.1.5 (Secondary visualization). Moreover, distribution of fare amount needs to be normalized by feature scaling method as shown in section 2.1.6*

2.1.2 Feature Engineering

Feature engineering is a process of transforming the given data into a form which is easier to interpret. Here, we are interested in making it more transparent for a machine learning model, but some features can be generated so that the data visualization prepared for people without a data-related background can be more digestible. Here we will be driving meaningful features from two variables i.e. **pickup_datetime** and coordinate variables viz. **pickup_longitude, pickup_latitude, dropoff_longitude and dropoff_latitude**

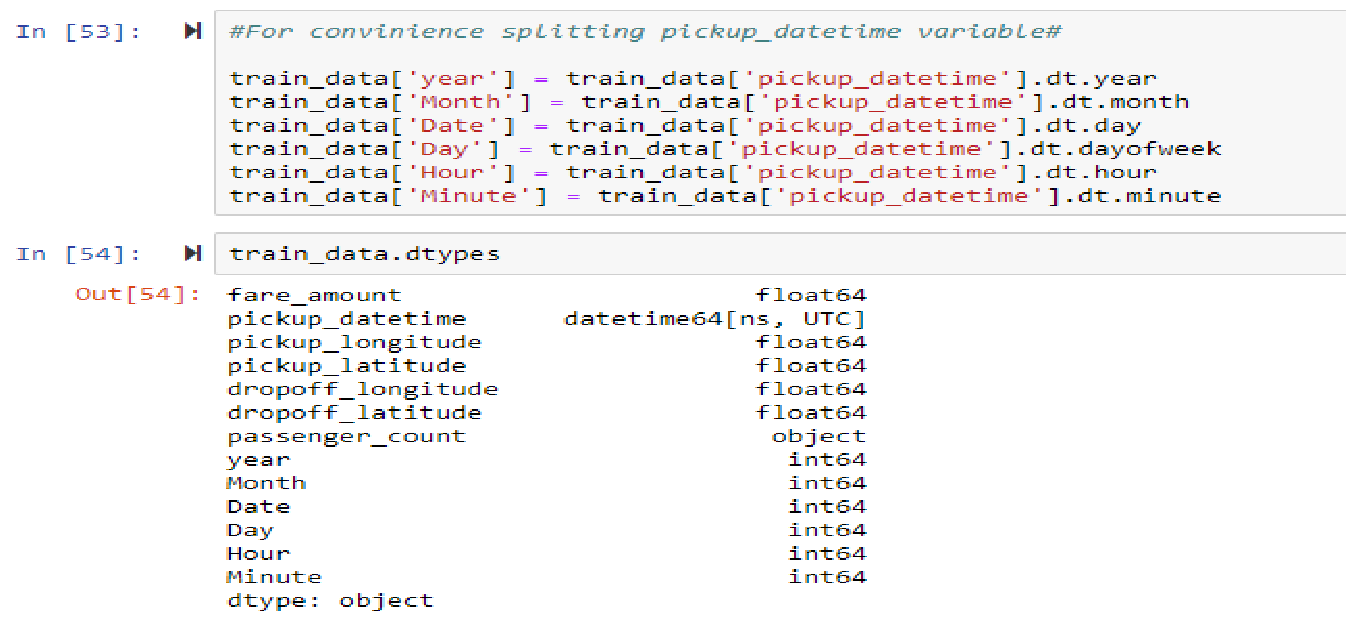
a) pickup_datetime is a time stamp data which contains all the six features i.e. **year, month, date, day, hour and minute** & hence could not be used to interpret the dependency of target variable with respect to year, month, date, day, hour, minute. Therefore, we need to split this variable into mentioned features.

Before proceeding with splitting, we need to first convert **pickup_datetime** into datetime format.



```
#Applying necessary data type conversions#

train_data['pickup_datetime'] =  pd.to_datetime(train_data['pickup_datetime'],errors ="coerce")
```

Now performing the required operations of feature engineering:

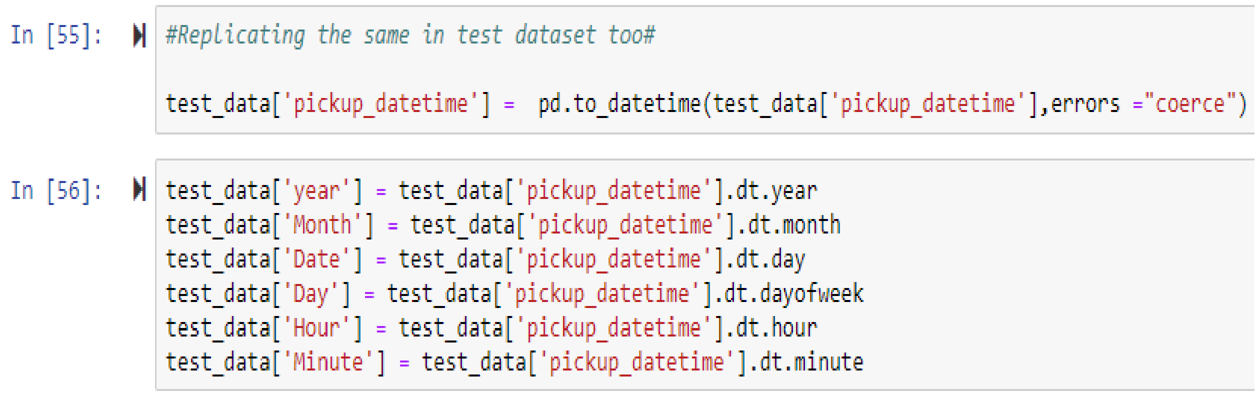


In [53]:
```
#For convinience splitting pickup_datetime variable#

train_data['year'] = train_data['pickup_datetime'].dt.year
train_data['Month'] = train_data['pickup_datetime'].dt.month
train_data['Date'] = train_data['pickup_datetime'].dt.day
train_data['Day'] = train_data['pickup_datetime'].dt.dayofweek
train_data['Hour'] = train_data['pickup_datetime'].dt.hour
train_data['Minute'] = train_data['pickup_datetime'].dt.minute
```

In [54]:
```
train_data.dtypes
```

Out[54]:
```
fare_amount                      float64
pickup_datetime      datetime64[ns, UTC]
pickup_longitude                 float64
pickup_latitude                  float64
dropoff_longitude                float64
dropoff_latitude                 float64
passenger_count                   object
year                               int64
Month                              int64
Date                               int64
Day                                int64
Hour                               int64
Minute                             int64
dtype: object
```

The same operation has been replicated in the test data too.*



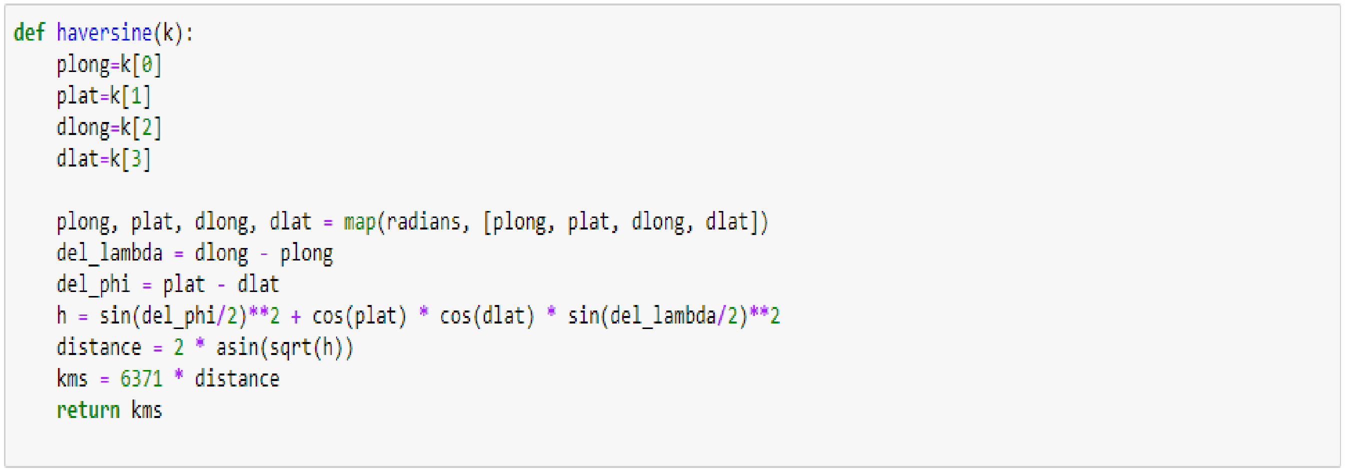
In [55]:
```
#Replicating the same in test dataset too#

test_data['pickup_datetime'] =  pd.to_datetime(test_data['pickup_datetime'],errors ="coerce")
```

In [56]:
```
test_data['year'] = test_data['pickup_datetime'].dt.year
test_data['Month'] = test_data['pickup_datetime'].dt.month
test_data['Date'] = test_data['pickup_datetime'].dt.day
test_data['Day'] = test_data['pickup_datetime'].dt.dayofweek
test_data['Hour'] = test_data['pickup_datetime'].dt.hour
test_data['Minute'] = test_data['pickup_datetime'].dt.minute
```

* **Note:** All the feature engineering steps has been replicated to test too, but the same is not shown for the consecutive steps so as to avoid the redundant presentation of the project report.

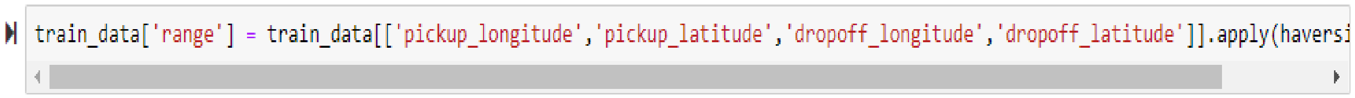
b) Coordinates data containing longitude and latitude values also does not give insight in a lucid term regarding the actual distance. Hence, it is required to interpret these variables in measurable distances. For doing the necessary feature engineering we need the help of 'Haversine formula' to calculate the distance using longitude and latitude values.

The **haversine formula** determines the great-circle distance between two points on a sphere given their longitudes and latitudes. Important in navigation, it is a special case of a more general **formula** in spherical trigonometry, the law of **haversines**, that relates the sides and angles of spherical triangles. (Check reference section)



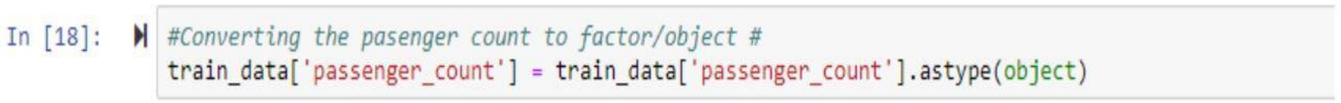
```
def haversine(k):
    plong=k[0]
    plat=k[1]
    dlong=k[2]
    dlat=k[3]

    plong, plat, dlong, dlat = map(radians, [plong, plat, dlong, dlat])
    del_lambda = dlong - plong
    del_phi = plat - dlat
    h = sin(del_phi/2)**2 + cos(plat) * cos(dlat) * sin(del_lambda/2)**2
    distance = 2 * asin(sqrt(h))
    kms = 6371 * distance
    return kms
```

By using the above formulae, we generate a new variable called **“range”** which describes the distance between pickup and drop-off point and which in turn will help us to predict our target variable in a more lucid manner.



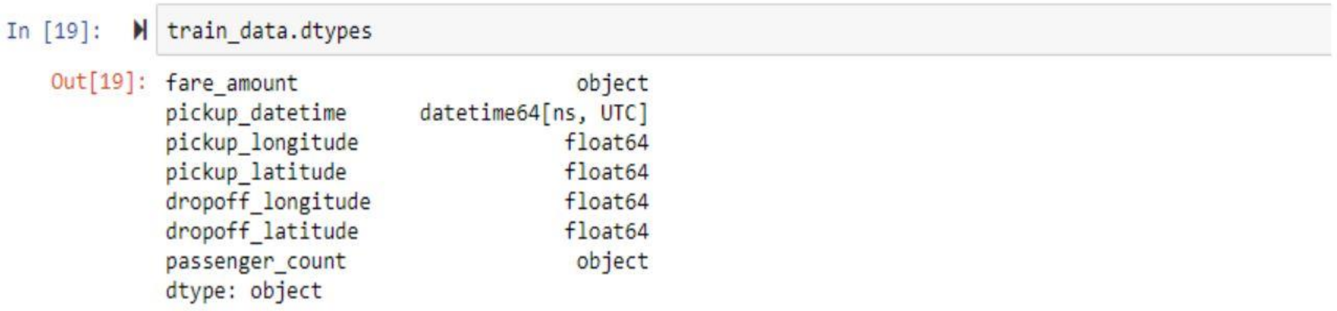
```
train_data['range'] = train_data[['pickup_longitude','pickup_latitude','dropoff_longitude','dropoff_latitude']].apply(haversi
```

c) passenger count: Data type conversion is required for passenger count variable since in this case number of passengers will be a factor only and count of the same in integer values is not possible.



```
In [18]:  #Converting the pasenger count to factor/object #
          train_data['passenger_count'] = train_data['passenger_count'].astype(object)
```

The datatypes of all the variable post data type conversion is shown below:



```
In [19]:  train_data.dtypes

Out[19]: fare_amount                       object
         pickup_datetime      datetime64[ns, UTC]
         pickup_longitude                 float64
         pickup_latitude                  float64
         dropoff_longitude                float64
         dropoff_latitude                 float64
         passenger_count                   object
         dtype: object
```

2.1.3 Outlier Analysis

We can clearly feel the presence of outliers and extreme values in our data from fig 1.2 where distorted visualizations can be seen to obtained while plotting.

One of the other steps of the analysis of the outliers in the data is using *boxplots*.

In figure 1.3 we have plotted the boxplot for fare_amount to see the presence of outliers.

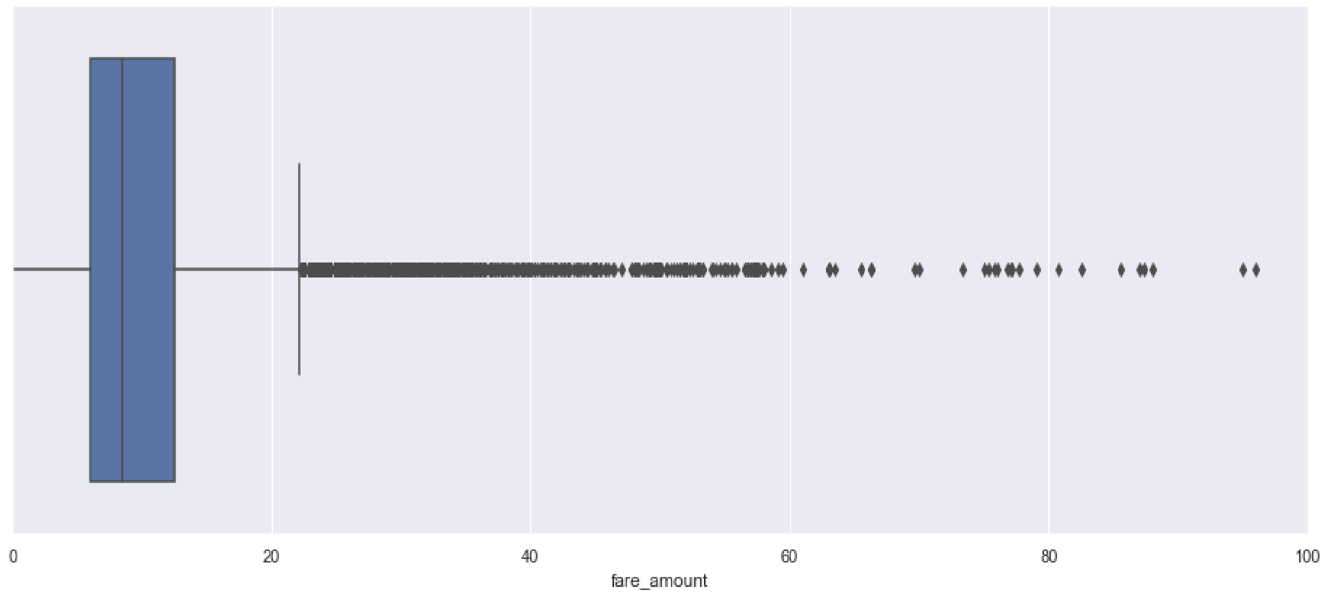
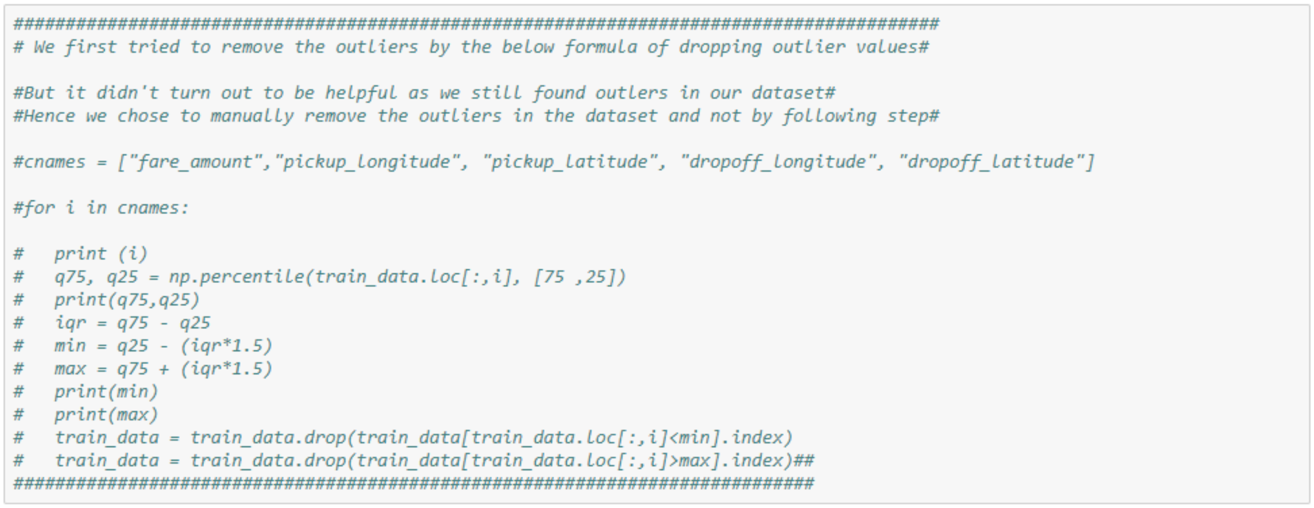


Fig. 1.3

Here, one-point needs clarification i.e. removal of outliers will not be performed by using the below method because issues were faced while executing the code I later stage where outliers were still present in the data even after execution of the below code.



```
###########################################################################################
# We first tried to remove the outliers by the below formula of dropping outlier values#

#But it didn't turn out to be helpful as we still found outlers in our dataset#
#Hence we chose to manually remove the outliers in the dataset and not by following step#

#cnames = ["fare_amount","pickup_longitude", "pickup_latitude", "dropoff_longitude", "dropoff_latitude"]

#for i in cnames:

#   print (i)
#   q75, q25 = np.percentile(train_data.loc[:,i], [75 ,25])
#   print(q75,q25)
#   iqr = q75 - q25
#   min = q25 - (iqr*1.5)
#   max = q75 + (iqr*1.5)
#   print(min)
#   print(max)
#   train_data = train_data.drop(train_data[train_data.loc[:,i]<min].index)
#   train_data = train_data.drop(train_data[train_data.loc[:,i]>max].index)##
#############################################################################
```

Hence in this project I have opted to remove the outliers manually for each variable separately.

a)Fare amount - The best way to remove the outliers in the data is to view the data in ascending and descending order to get an idea of extreme data points that might distort the dataset and then drop the values which are both out of range and below range. I have cross checked by viewing the data in descending order again and no outliers value were found post removal of the same.

Here I have implemented the same using the below code.

```
#We can see some absurd values as high as 50k and some negetive values too#
#we need to eliminate these outliers#

train_data = train_data.drop(train_data[train_data["fare_amount"]<1].index, axis=0)
```

```
train_data = train_data.drop(train_data[train_data["fare_amount"]>453].index, axis=0)
```

```
train_data["fare_amount"].sort_values(ascending=False)
```

b) passenger count: Here the outlier removal has been done manually by simply screening the data between 1 to 6. As we have already converted the datatype from integer to factor, this is also known that minimum passenger for any vehicle type will be 1 and maximum passenger count that can be accommodated considering an SUV is 6. Hence, simply screening the data and dropping all misfit values has done the needful of removing the outliers here.

```
#By performing the sort operation we found that there are way too more extreme values that we imagined#
#By viewing the data on descending order we found there are few entries having value more than 6#
#By viewing the data on ascending order we found many entries having 0 value#
#Many NA values were also found#

train_data = train_data.drop(train_data[train_data['passenger_count']<1].index, axis=0)
```

```
train_data = train_data.drop(train_data[train_data['passenger_count']>6].index, axis=0)
```

c) range: The boxplots for predicting the outliers were first plotted for one of the coordinates i.e. pickup latitude, so as to get an idea of the presence of outliers in the coordinate data. But the very same idea was dropped as the concept of coordinates to measure the distance is not known to many.

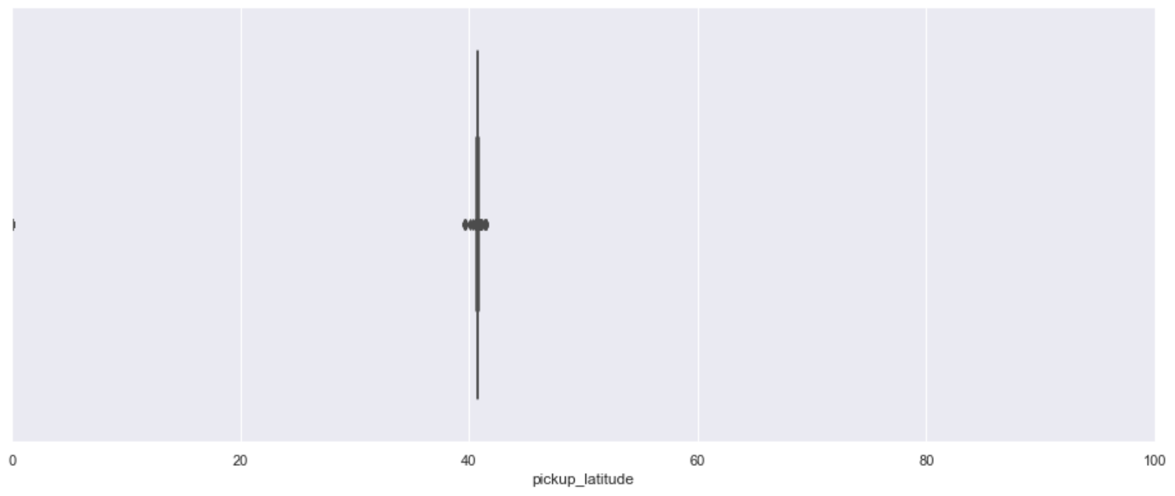


Fig. 1.4

Hence, it was decided to remove the outliers in the new variable 'range' obtained after the feature engineering process, as it is easier to detect outliers when he value is in a measurable metric i.e. 'kilometres'

The boxplot is plotted below for ‘range’ to detect the outliers in the data.

```
#Checking outliers in range#
plt.figure(figsize=(16,6))
plt.xlim(0,150)
sns.boxplot(x=train_data['range'],data=train_data)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x5472b5ae10>
```

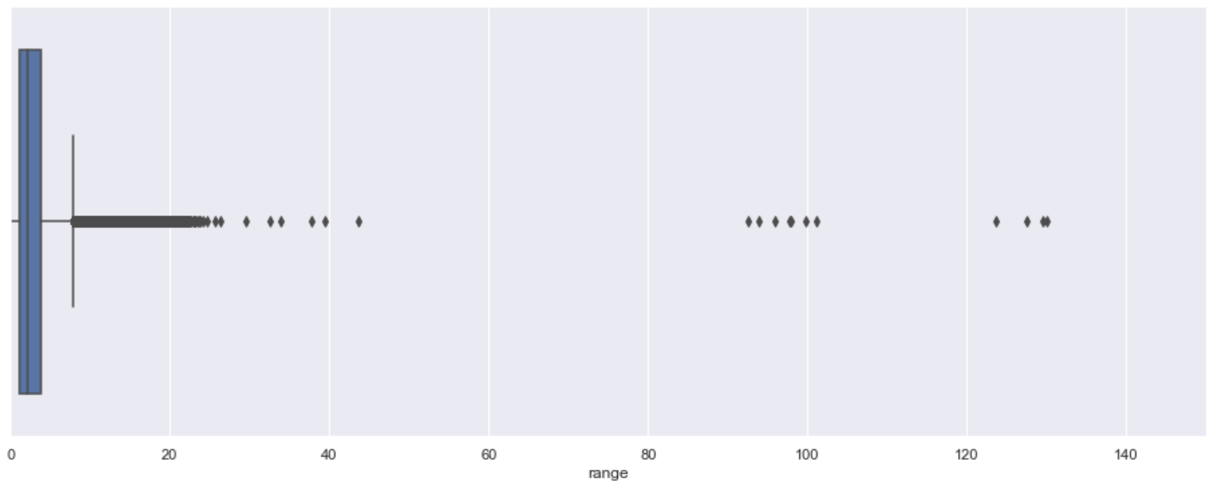


Fig. 1.5

The outlier as detected has been dropped manually as shown below.

```
#too many outlers in range#
train_data["range"].sort_values(ascending=False)
```

```
train_data = train_data.drop(train_data[train_data['range']< 0.1].index, axis=0)
```

```
train_data = train_data.drop(train_data[train_data['range']== 0].index, axis=0)
```

```
train_data = train_data.drop(train_data[train_data['range'] > 150 ].index, axis=0)
```

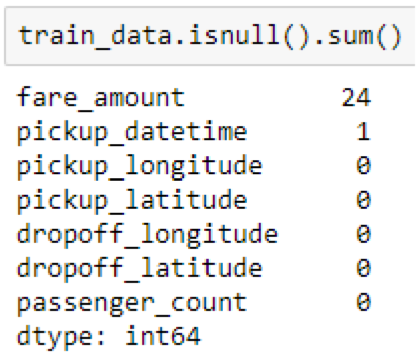
2.1.4 Missing value analysis

The detection and deletion of missing values has been done variable wise post outlier removal. Though, the removal of missing value while coding was done separately for each variable. In this report, we have compiled the snips of outlier removal operation as follows.

```
train_data = train_data.drop(train_data[train_data['passenger_count'].isnull()].index, axis=0)
```

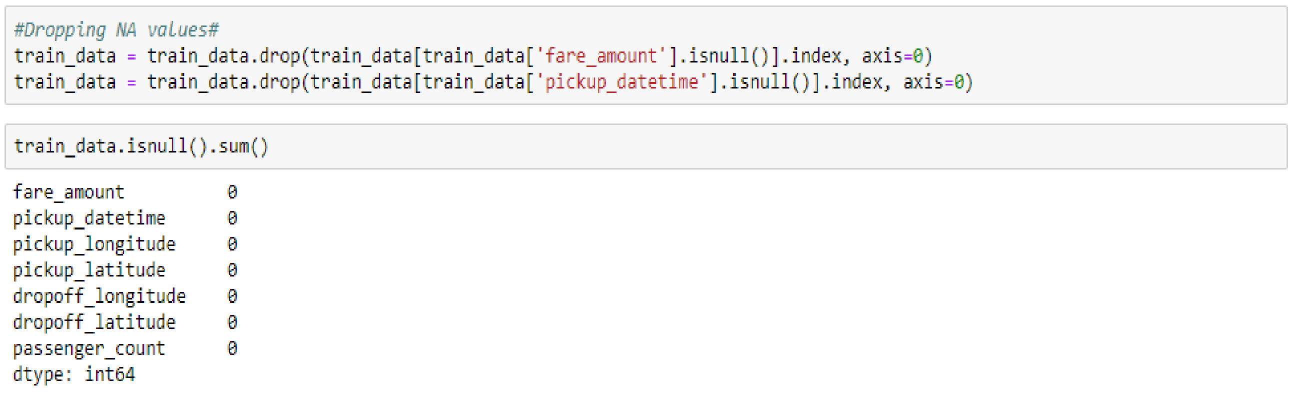
```
print(train_data['passenger_count'].isnull().sum())
```

```
0
```



```
train_data.isnull().sum()
```

```
fare_amount          24
pickup_datetime       1
pickup_longitude      0
pickup_latitude       0
dropoff_longitude     0
dropoff_latitude      0
passenger_count       0
dtype: int64
```

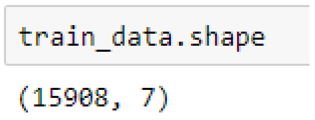


```
#Dropping NA values#
train_data = train_data.drop(train_data[train_data['fare_amount'].isnull()].index, axis=0)
train_data = train_data.drop(train_data[train_data['pickup_datetime'].isnull()].index, axis=0)
```

```
train_data.isnull().sum()
```

```
fare_amount          0
pickup_datetime      0
pickup_longitude     0
pickup_latitude      0
dropoff_longitude    0
dropoff_latitude     0
passenger_count      0
dtype: int64
```

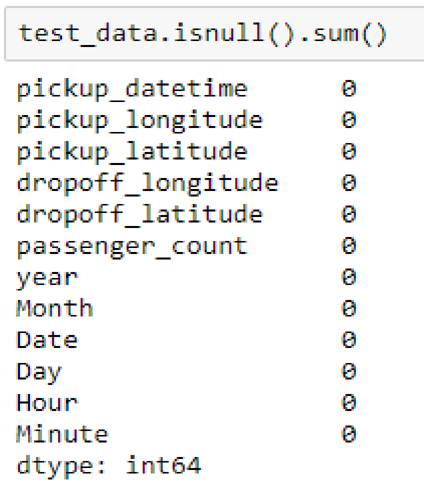
The shape of train data post missing value clearance is shown below:



```
train_data.shape
```

```
(15908, 7)
```

The same is replicated for test data too.



```
test_data.isnull().sum()
```

```
pickup_datetime      0
pickup_longitude     0
pickup_latitude      0
dropoff_longitude    0
dropoff_latitude     0
passenger_count      0
year                 0
Month                0
Date                 0
Day                  0
Hour                 0
Minute               0
dtype: int64
```

2.1.5 Secondary Visualizations

After outlier removal and feature engineering steps we have plotted the below visualizations that can show the relationship with the target variable (fare amount).

Bar chart distribution of passenger count:

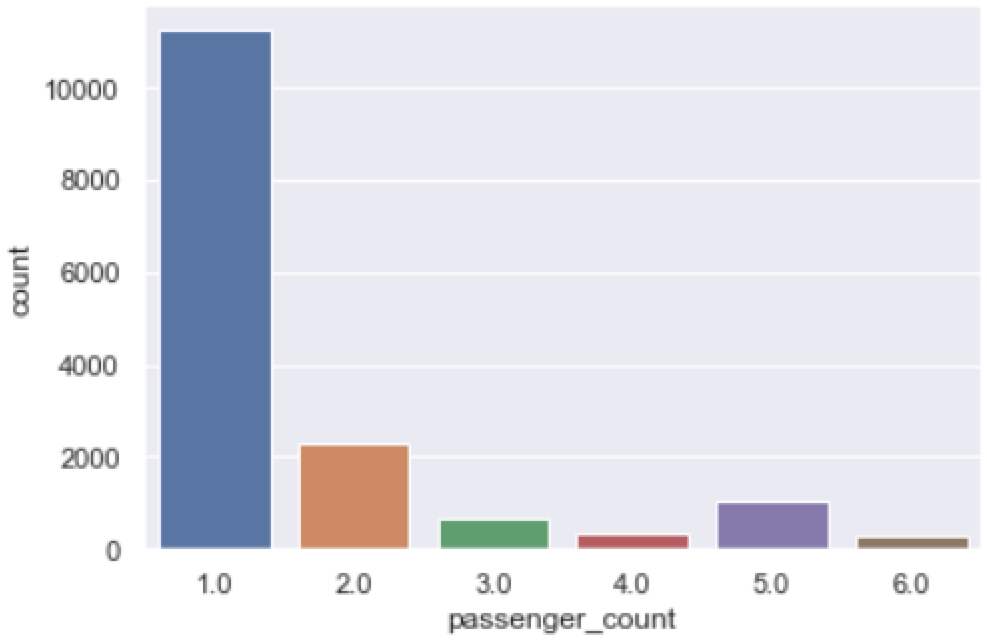


Fig – 1.6

Figure 1.7 shows us a stark difference and the extent to which data cleaning methods if executed correctly can be effective in deriving inference just by visualization. This is in sharp contrast to what we got in fig. 1.2 when outliers were present in the data.

Bar chart distribution of passenger count with respect to month:

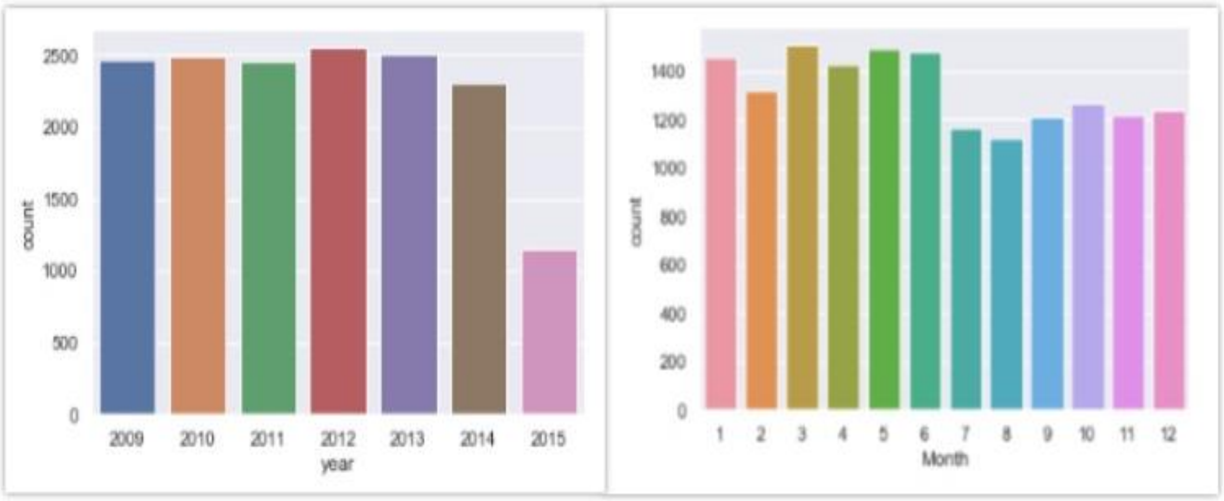


Fig – 1.7

Bar chart distribution of passenger count according to date of the month and day of week:

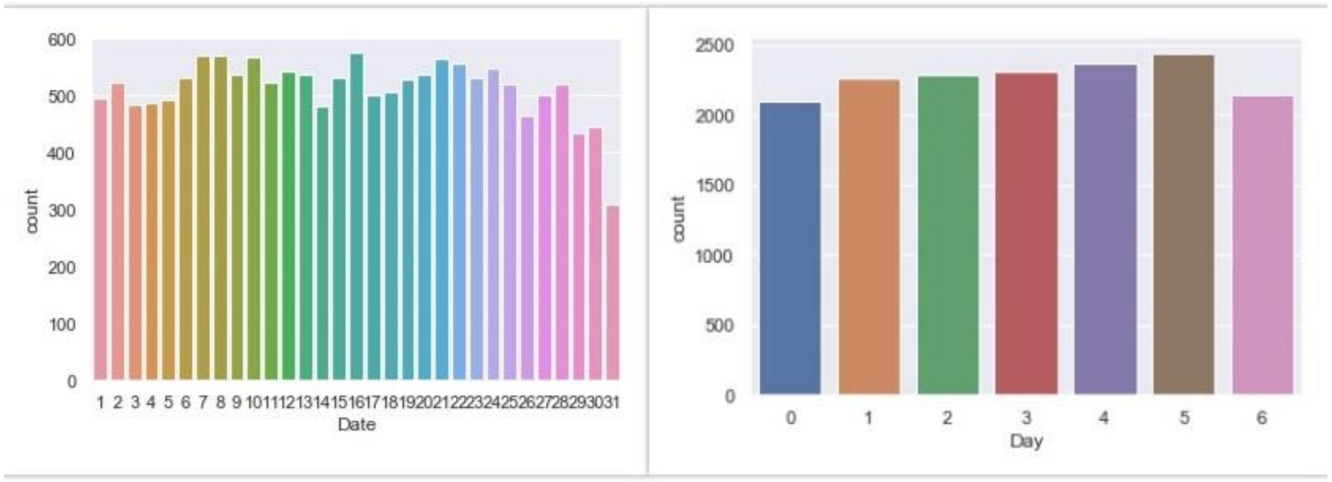


Fig – 1.8

Bar chart distribution of passenger count according to hour and minute:

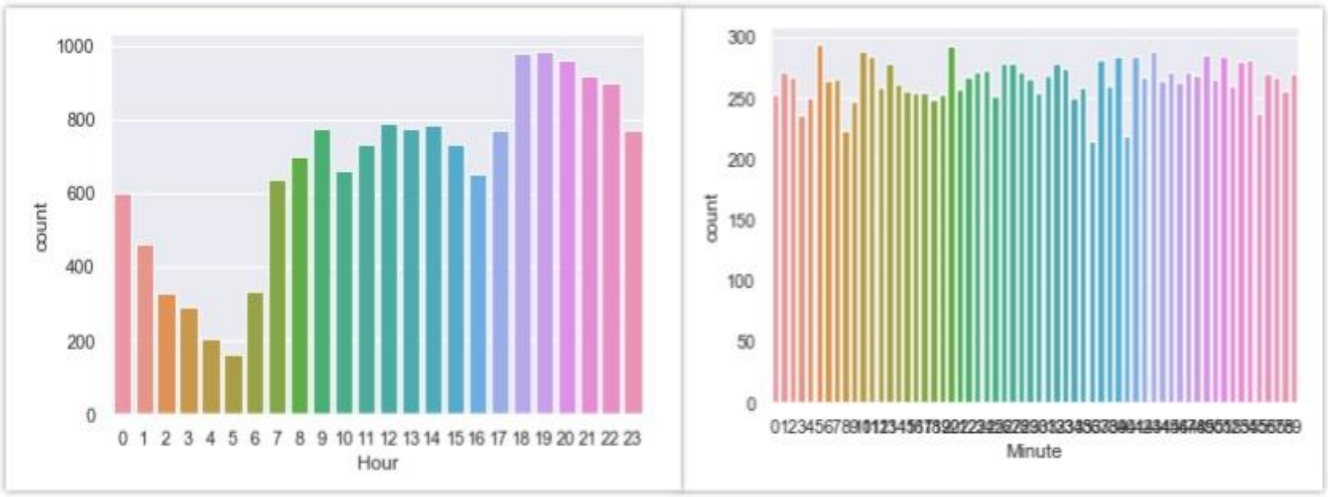


Fig – 1.9

Interpretation:

- In fig. 1.7 it can be understood that most of the cab bookings are done by the solo riders.
- It can be inferred from the data distribution of Month with respect to passenger (fig. 1.8) count that the count of passengers is comparatively lower than in the second half of the year.
- From fig. 1.9, it can be concluded that the last days of the month (30th and 31st) as well as Saturdays registers lowest cab bookings.
- In fig. 1.10 we get a very needful information as on which ‘hour’ shall the cab company implement the price surge. Minute trend is however is not much useful in deriving the required prediction, hence we can remove this variable in feature selection process.

Scatter plots for relationship between fare amount

Scatter plot for relationship between fare amount:

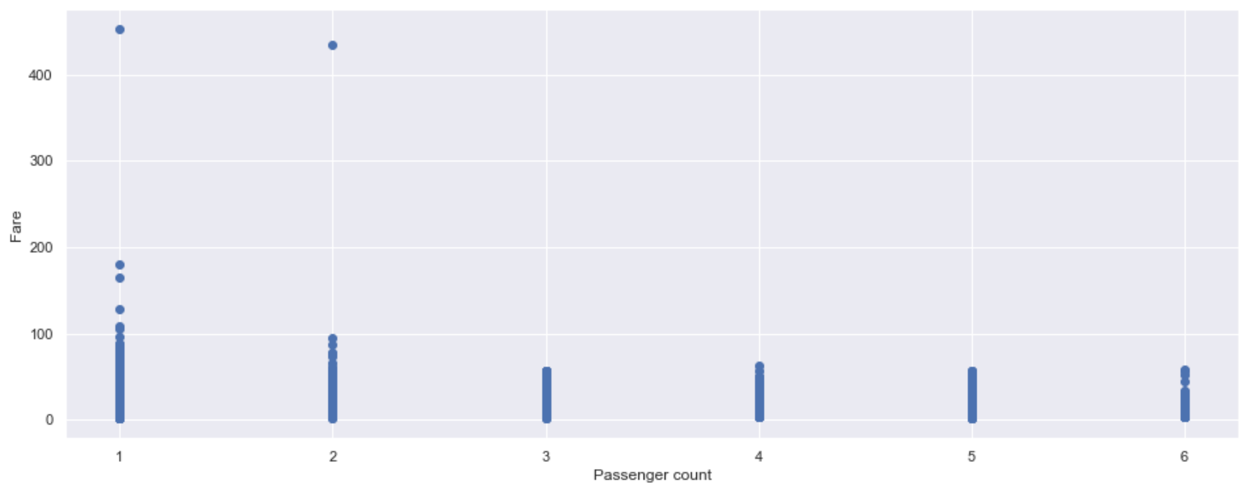


Fig – 1.10

Scatter plot for relationship between fare amount:

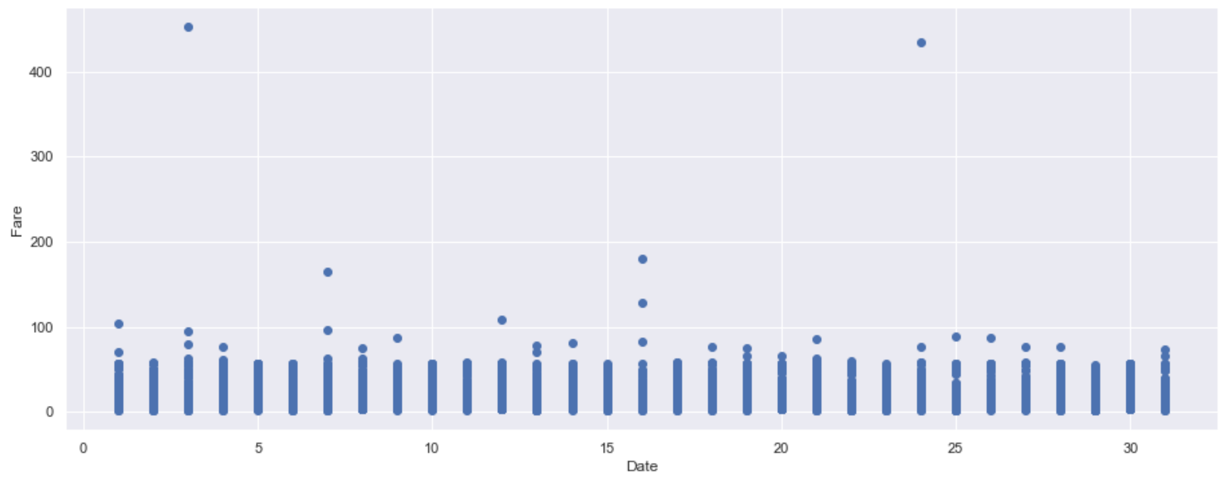


Fig – 1.11

Scatter plot for relationship between fare amount:

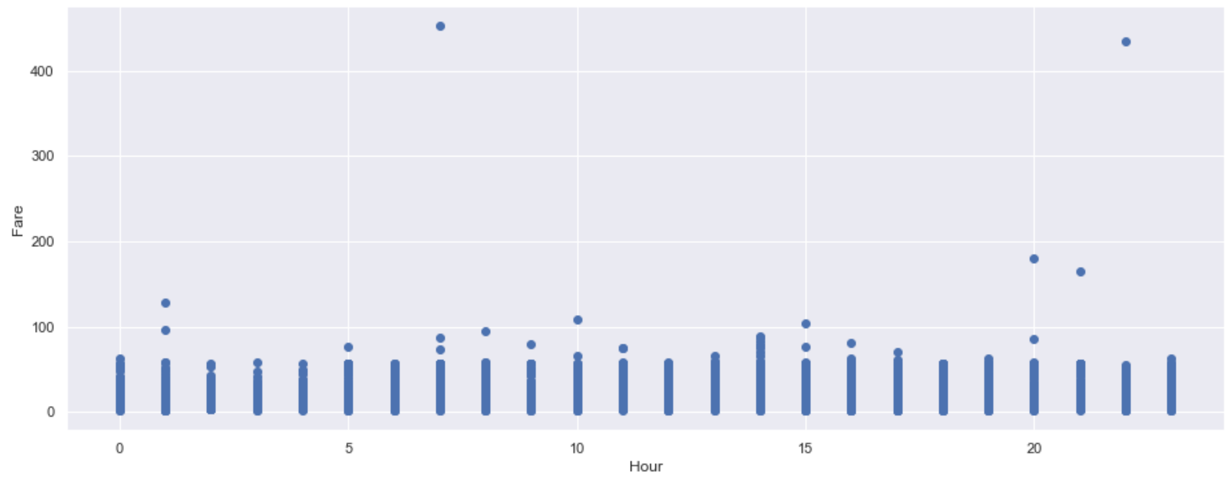


Fig – 1.12

2.1.6 Feature Selection

Correlation analysis

Before performing any type of modelling, we need to assess the importance of each predictor variable in our analysis. There is a possibility that many variables in our analysis are not important at all and we need to select only those selective variables which will have very high correlation with the target variable and thereby contribute to predict the dependent variable. Moreover, it will help us detect and eliminate the variables that can cause multicollinearity problem. In this step we drop certain variables which contains the same information and can increase the complexity of the of the model on which we are going to predict the target variable, hence performing feature selection will help us to remove irrelevant features from the dataset.

We have plotted the below correlation plot of continuous variables to determine the possible multicollinearity between variables.

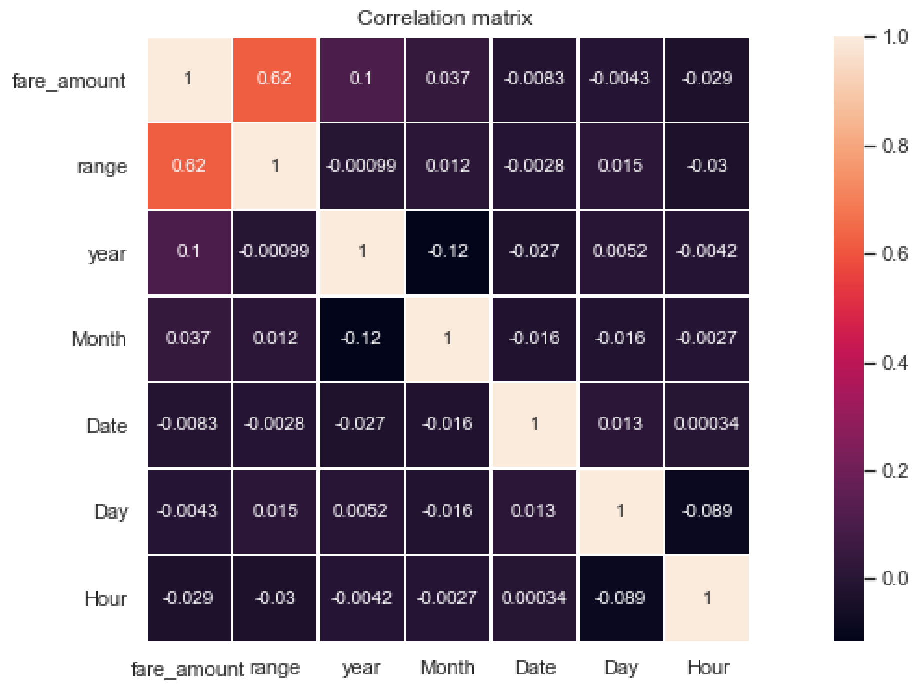
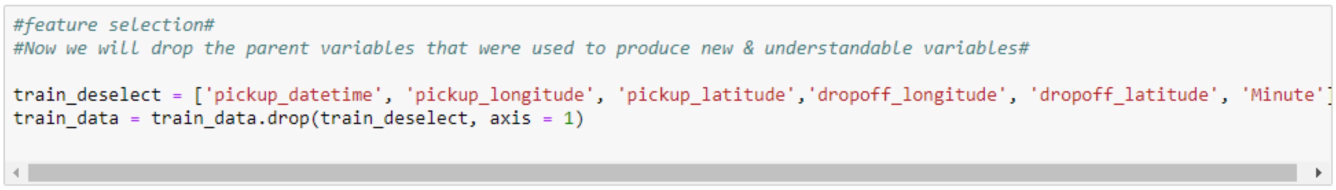


Fig. 1.13

Interpretation:

- Here we can see that the fare amount is positively correlated with ‘range’
- Here we will be mainly dropping those values which were used to create new features in feature engineering step i.e. coordinate variables containing longitude and latitude values and time stamp variable(pickup_datetime)
- We are also removing ‘Minute’ as discussed in last section w.r.t. fig. 1.10.

We are doing the needful as per below code:



```
#feature selection#
#Now we will drop the parent variables that were used to produce new & understandable variables#

train_deselect = ['pickup_datetime', 'pickup_longitude', 'pickup_latitude','dropoff_longitude', 'dropoff_latitude', 'Minute']
train_data = train_data.drop(train_deselect, axis = 1)
```

2.1.6 Feature Scaling

Data Scaling methods are used when we want our variables in data to scaled on common ground. It is performed only on continuous variables. Normalization is used for feature scaling when the data is not normally distributed. In this case, 'range' which is our independent variable here is not normally distributed.

a) We have checked the variance in dataset before normalisation. A distplot graph is plotted before normalization is performed.

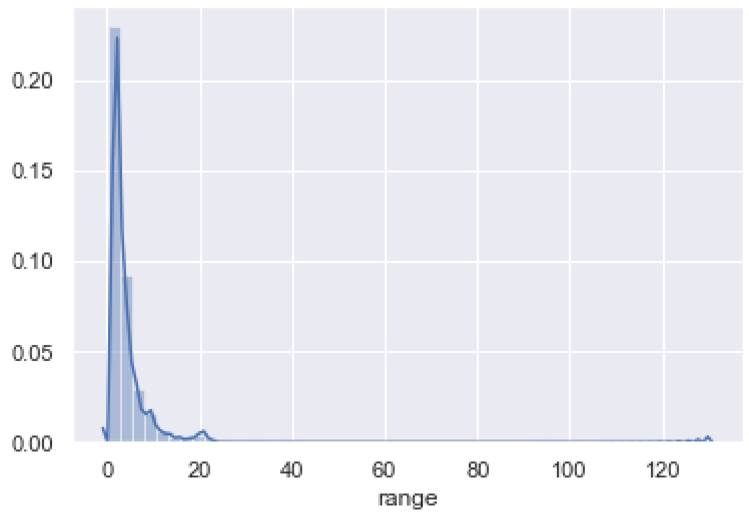


Fig. 1.14

b) High variance will affect the accuracy of the model. So, we must normalise that variance as shown in the distplot graph below:

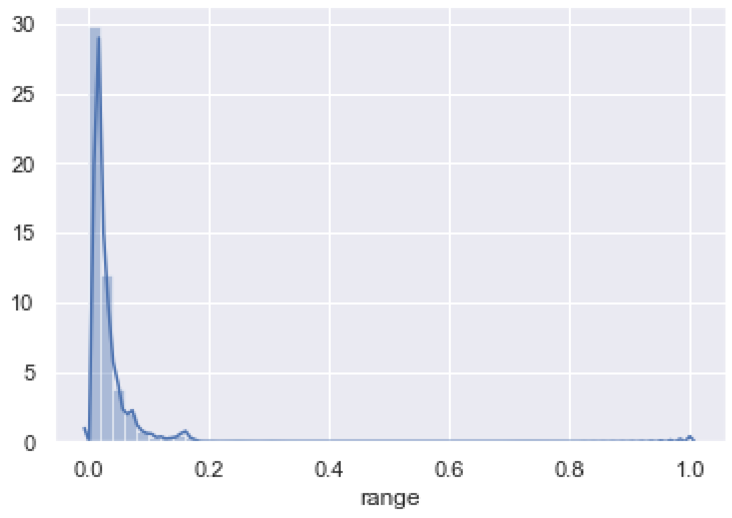


Fig. 1.15

Chapter 3

Modelling

3.1 Model Selection & Evaluation

Since our target variable is continuous and we want to know to know how well the model predicts the new data and we will select the following regression models:

1. Multiple Linear Regression

2. Decision Tree

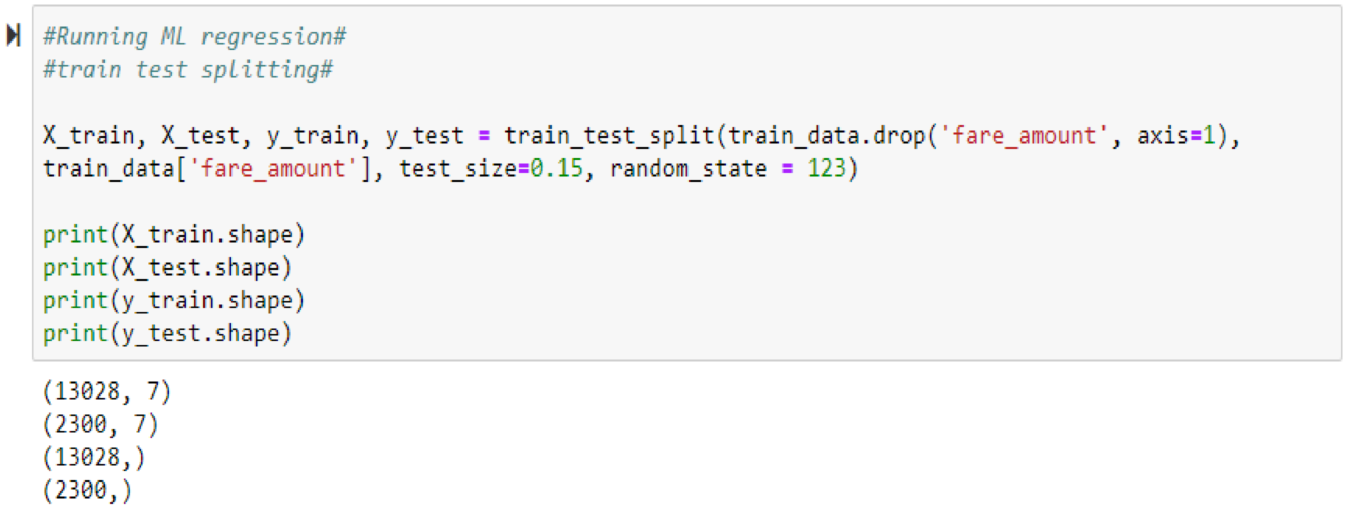
3. Random Forest

Train Test Splitting:

Before running any model, we will split our data into two parts which is train and test data. Here in our case we have taken 80% of the data as our train data as shown below. We will test the performance of model on validation dataset.

The model which performs best will be chosen to perform on test dataset provided along with original train dataset.

X_train, y_train are training subset & X_test, y_test are validation subset.



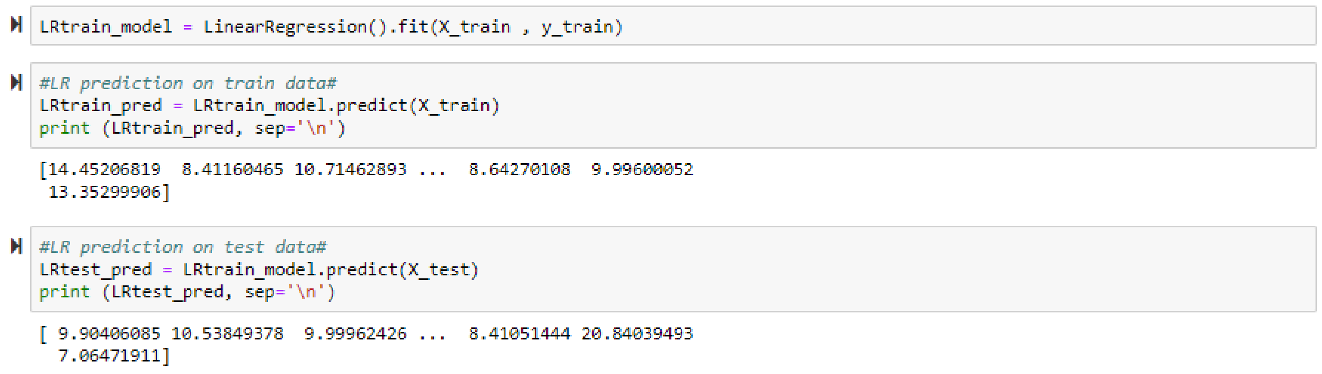
```
#Running ML regression#
#train test splitting#

X_train, X_test, y_train, y_test = train_test_split(train_data.drop('fare_amount', axis=1),
train_data['fare_amount'], test_size=0.15, random_state = 123)

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(13028, 7)
(2300, 7)
(13028,)
(2300,)
```

3.1.1 Multiple Linear regression

Multiple linear regression is used to explain the relationship between one dependent carriable with multiple independent variables. Our dependent variable here is a continuous one.

The Linear regression model as built using Python 3 is shown below:



```
LRtrain_model = LinearRegression().fit(X_train , y_train)
```

```
#LR prediction on train data#
LRtrain_pred = LRtrain_model.predict(X_train)
print (LRtrain_pred, sep='\n')
```

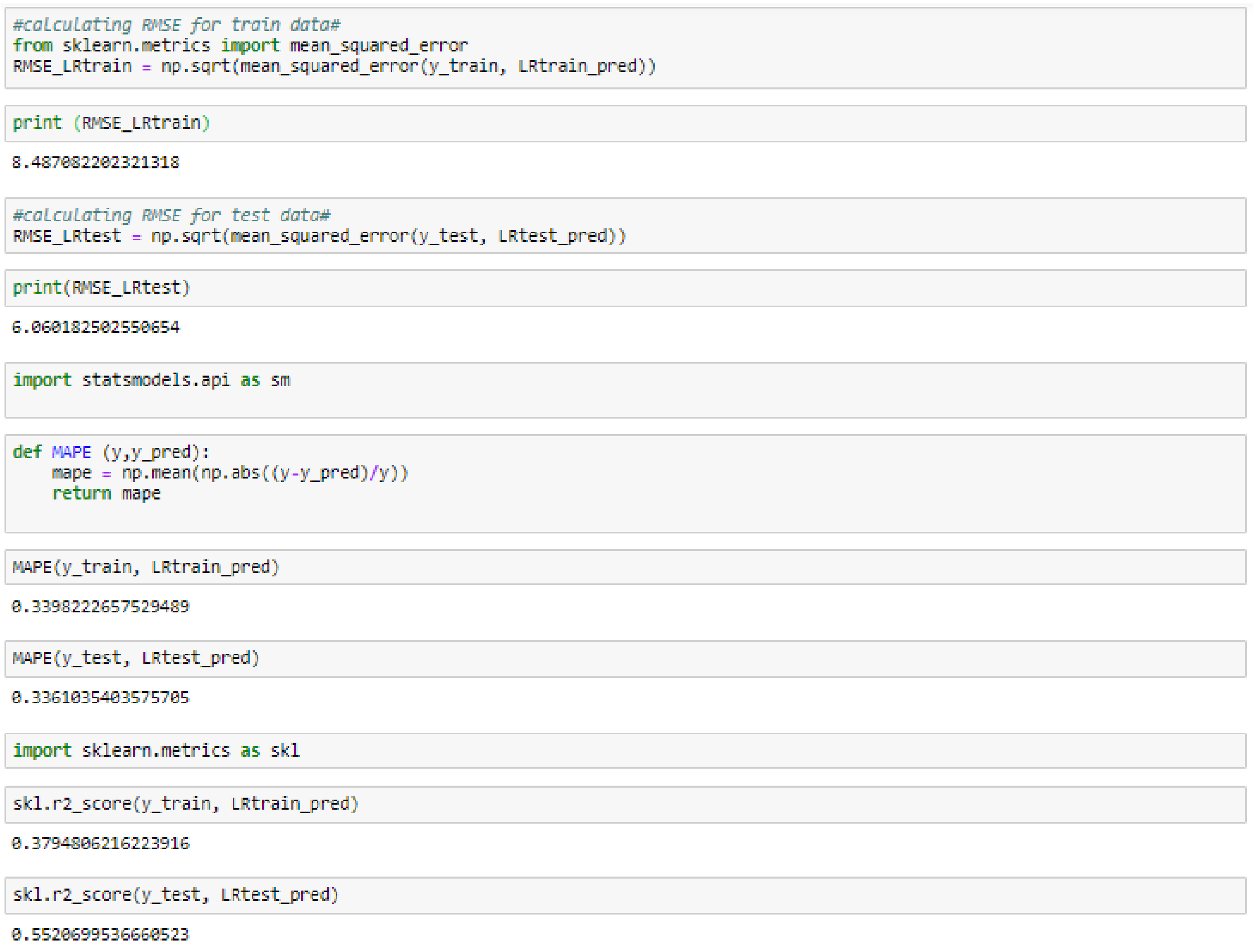
```
[14.45206819  8.41160465 10.71462893 ...  8.64270108  9.99600052
 13.35299906]
```

```
#LR prediction on test data#
LRtest_pred = LRtrain_model.predict(X_test)
print (LRtest_pred, sep='\n')
```

```
[ 9.90406085 10.53849378  9.99962426 ...  8.41051444 20.84039493
  7.06471911]
```

Model Evaluation:

The necessary error metrics used for model evaluation of linear regression model is described as below:



```
#calculating RMSE for train data#
from sklearn.metrics import mean_squared_error
RMSE_LRtrain = np.sqrt(mean_squared_error(y_train, LRtrain_pred))
```

```
print (RMSE_LRtrain)
```

```
8.487082202321318
```

```
#calculating RMSE for test data#
RMSE_LRtest = np.sqrt(mean_squared_error(y_test, LRtest_pred))
```

```
print(RMSE_LRtest)
```

```
6.060182502550654
```

```
import statsmodels.api as sm
```

```
def MAPE (y,y_pred):
    mape = np.mean(np.abs((y-y_pred)/y))
    return mape
```

```
MAPE(y_train, LRtrain_pred)
```

```
0.3398222657529489
```

```
MAPE(y_test, LRtest_pred)
```

```
0.3361035403575705
```

```
import sklearn.metrics as skl
```

```
skl.r2_score(y_train, LRtrain_pred)
```

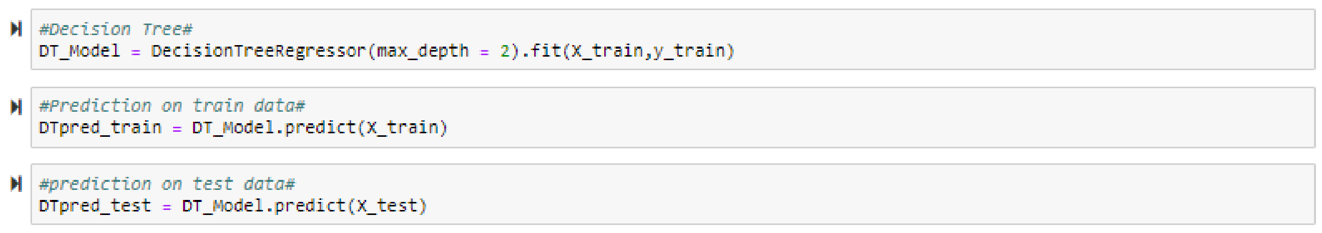
```
0.3794806216223916
```

```
skl.r2_score(y_test, LRtest_pred)
```

```
0.5520699536660523
```

3.1.2 Decision Tree

Now we will try and use another regression model known as decision tree to predict the fare amount of cab ride.



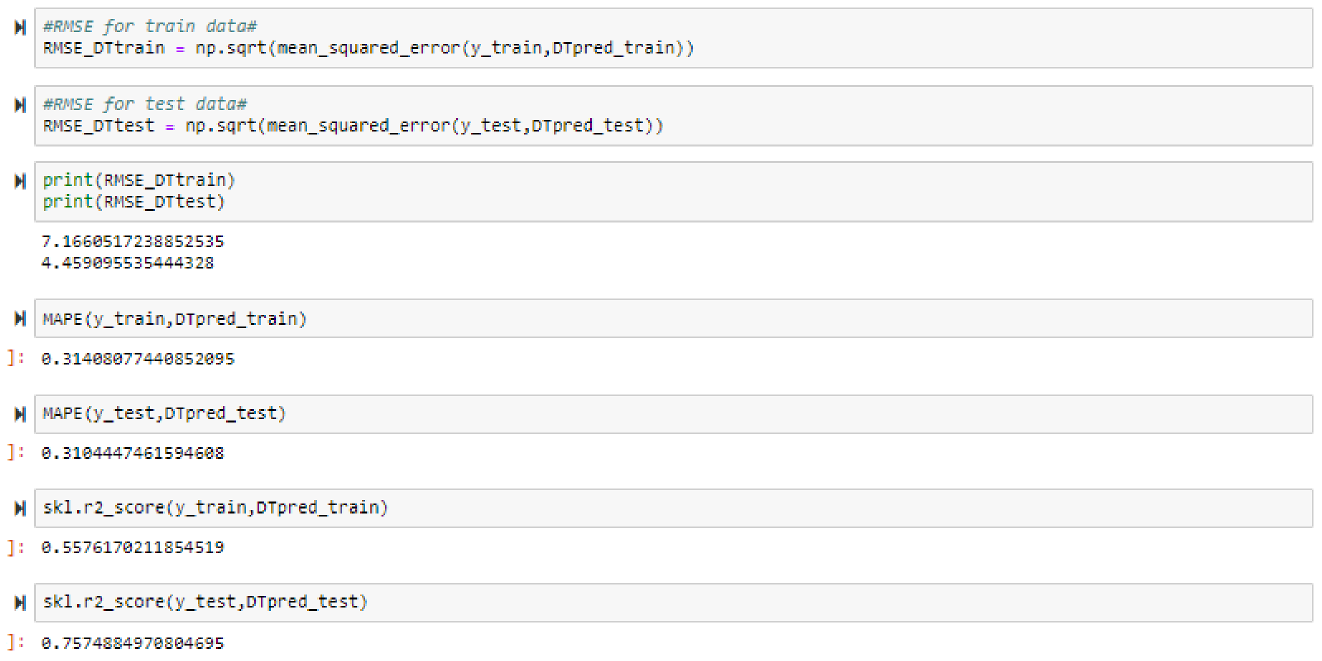
```
#Decision Tree#
DT_Model = DecisionTreeRegressor(max_depth = 2).fit(X_train,y_train)
```

```
#Prediction on train data#
DTpred_train = DT_Model.predict(X_train)
```

```
#prediction on test data#
DTpred_test = DT_Model.predict(X_test)
```

Model Evaluation:

We have calculated the following error metrices, where we find the MAE and RMSE to be error metrices for the evaluation of the model.



```
#RMSE for train data#
RMSE_DTtrain = np.sqrt(mean_squared_error(y_train,DTpred_train))
```

```
#RMSE for test data#
RMSE_DTtest = np.sqrt(mean_squared_error(y_test,DTpred_test))
```

```
print(RMSE_DTtrain)
print(RMSE_DTtest)

7.1660517238852535
4.459095535444328
```

```
MAPE(y_train,DTpred_train)

]: 0.31408077440852095
```

```
MAPE(y_test,DTpred_test)

]: 0.3104447461594608
```

```
skl.r2_score(y_train,DTpred_train)

]: 0.5576170211854519
```

```
skl.r2_score(y_test,DTpred_test)

]: 0.7574884970804695
```

Root mean square of errors for training dataset is 7.16
Root mean square of errors for test dataset is 4.45

MAPE for both dataset was found to be 31%

R^2 value for training dataset is 0.55
R^2 value for test dataset is 0.75

3.1.3 Random Forest

Now we will try our final model for regression called random forest. Random forest is an ensemble that consist of many decision trees.

```
#Random Forest#
RF_Model = RandomForestRegressor(n_estimators = 200).fit(X_train,y_train)
```

```
#prediction on train data#
RFpred_train = RF_Model.predict(X_train)
```

```
#prediction on test data#
RFpred_test = RF_Model.predict(X_test)
```

Model Evaluation:

We have calculated the following error metrices, where we find the MAPE and RMSE to be error metrices for the evaluation of the model.

```
RMSE_RFtrain = np.sqrt(mean_squared_error(y_train,RFpred_train))
```

```
RMSE_RFtest = np.sqrt(mean_squared_error(y_test,RFpred_test))
```

```
print (RMSE_RFtrain)
print (RMSE_RFtest)
```

```
2.7521480327255574
3.8246022079990913
```

```
MAPE(y_train,RFpred_train)
```

```
0.0828879272666313
```

```
MAPE(y_test,RFpred_test)
```

```
0.20958007765297532
```

```
skl.r2_score(y_train,RFpred_train)
```

```
0.9347497602608128
```

```
skl.r2_score(y_test,RFpred_test)
```

```
0.8215932295373434
```

Root mean square of errors for training dataset is 2.75
Root mean square of errors for test dataset is 3.82

MAPE for training dataset is 8%
MAPE for test dataset is 21%

R^2 value for training dataset is 0.93
R^2 value for test dataset is 0.82

Chapter 4

Hyper parameter tuning

If we optimize the model for the training data, then our model will score very well on the training dataset but will not be able to generalize to new data, such as in a test dataset.

Evaluating each model only on the training set can lead to one of the most fundamental problems in machine learning: overfitting.

An overfit model may look impressive on the training dataset but will be useless in a real application. Therefore, the standard procedure for hyperparameter optimization accounts for overfitting through cross validation (CV).

In this section, we will focus on optimizing the random forest model in Python using Scikit-Learn tools. Here we have used the two most popular techniques of hyper parameter tuning viz.

i) Random search CV
ii) Grid search CV

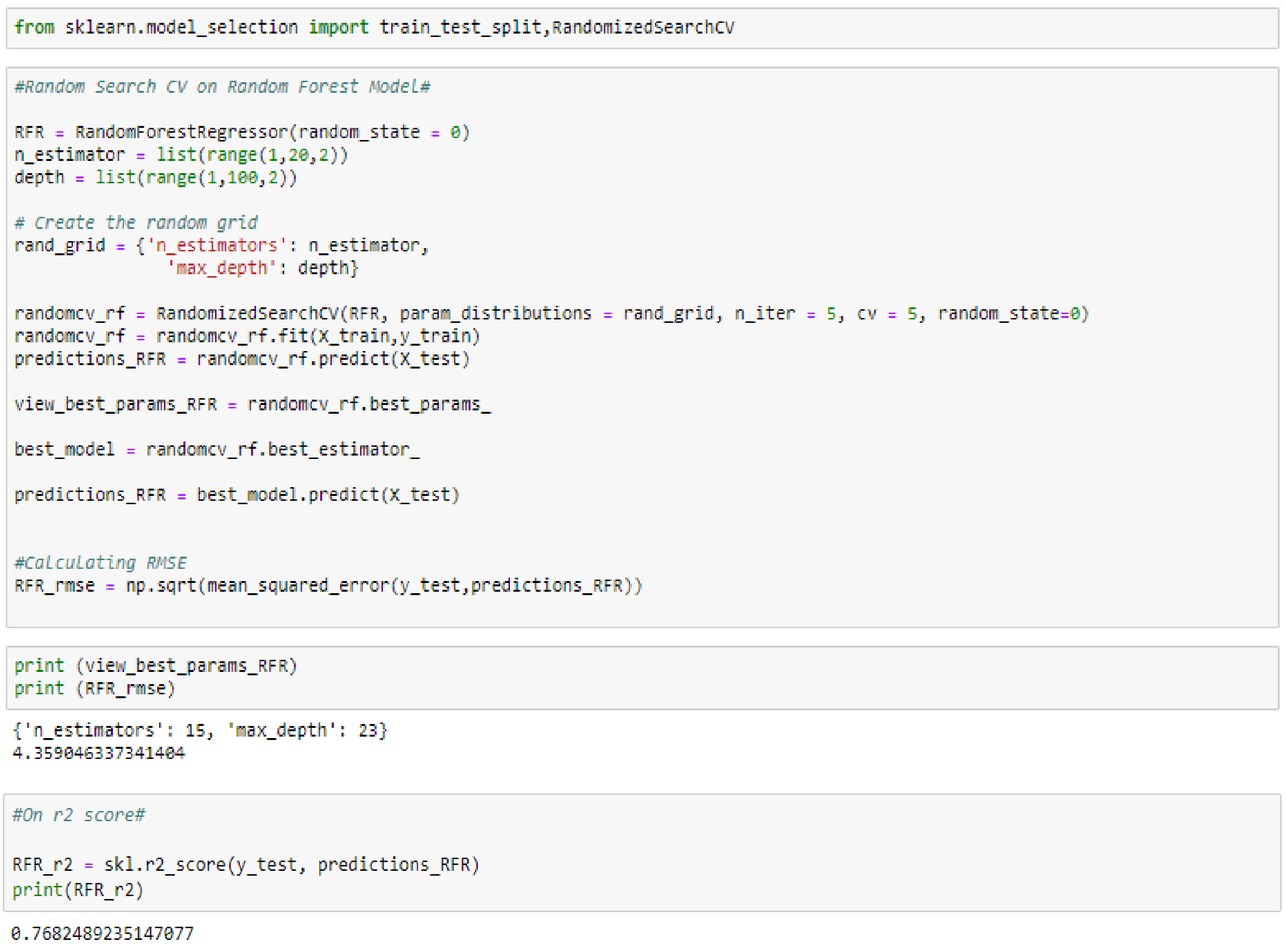
4.1 Random Search CV

Random Search CV sets up a grid of hyperparameter values and select random combinations to train the model and score. The number of search iterations is set based on resources.

```
#Hyper parameter tuning with Randomsearch CV#

rf = RandomForestRegressor(random_state = 45)
from pprint import pprint

#parameters used by our current forest#

print('Parameters currently in use')
pprint(rf.get_params())
```

```
Parameters currently in use
{'bootstrap': True,
 'criterion': 'mse',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 'warn',
 'n_jobs': None,
 'oob_score': False,
 'random_state': 45,
 'verbose': 0,
 'warm_start': False}
```

The result of using Random search CV on random forest is shown below:



```
from sklearn.model_selection import train_test_split,RandomizedSearchCV
```

```
#Random Search CV on Random Forest Model#

RFR = RandomForestRegressor(random_state = 0)
n_estimator = list(range(1,20,2))
depth = list(range(1,100,2))

# Create the random grid
rand_grid = {'n_estimators': n_estimator,
               'max_depth': depth}

randomcv_rf = RandomizedSearchCV(RFR, param_distributions = rand_grid, n_iter = 5, cv = 5, random_state=0)
randomcv_rf = randomcv_rf.fit(X_train,y_train)
predictions_RFR = randomcv_rf.predict(X_test)

view_best_params_RFR = randomcv_rf.best_params_

best_model = randomcv_rf.best_estimator_

predictions_RFR = best_model.predict(X_test)


#Calculating RMSE
RFR_rmse = np.sqrt(mean_squared_error(y_test,predictions_RFR))
```

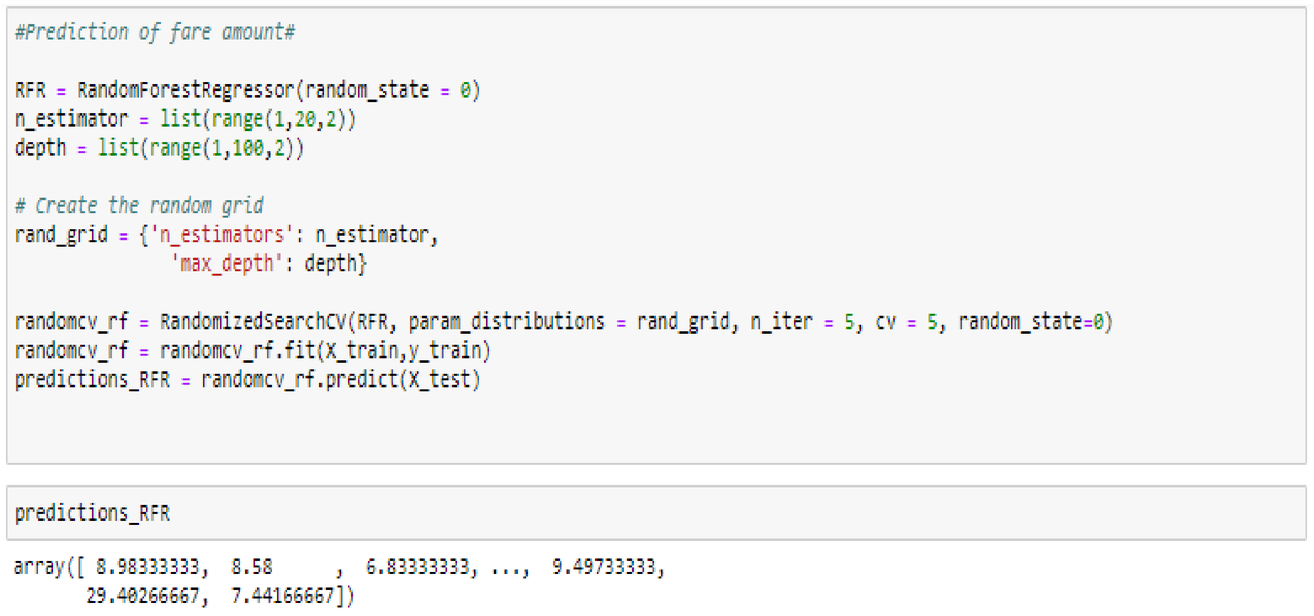
```
print (view_best_params_RFR)
print (RFR_rmse)
```

```
{'n_estimators': 15, 'max_depth': 23}
4.359046337341404
```

```
#On r2 score#

RFR_r2 = skl.r2_score(y_test, predictions_RFR)
print(RFR_r2)
```

```
0.7682489235147077
```

Random Search CV hyper parameter tuning is applied to predict the fare amount in test dataset. The predictions for Random Search CV will be merged in test data frame in coming steps.



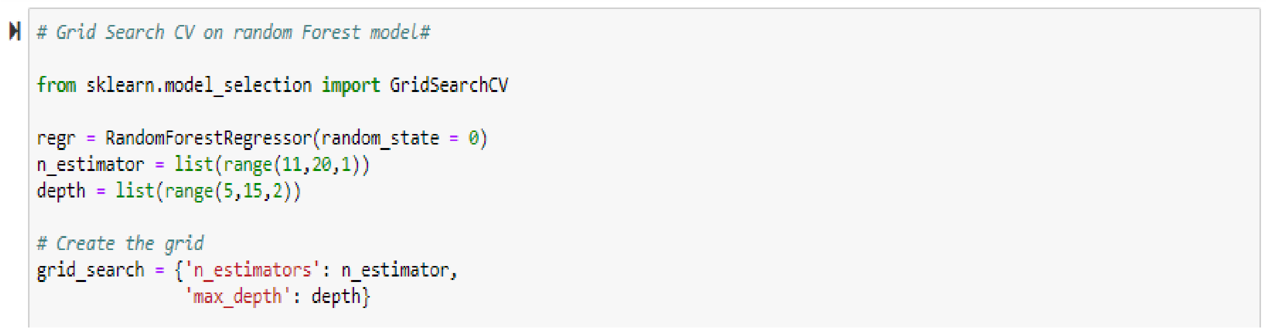
```
#Prediction of fare amount#

RFR = RandomForestRegressor(random_state = 0)
n_estimator = list(range(1,20,2))
depth = list(range(1,100,2))

# Create the random grid
rand_grid = {'n_estimators': n_estimator,
               'max_depth': depth}

randomcv_rf = RandomizedSearchCV(RFR, param_distributions = rand_grid, n_iter = 5, cv = 5, random_state=0)
randomcv_rf = randomcv_rf.fit(X_train,y_train)
predictions_RFR = randomcv_rf.predict(X_test)
```

```
predictions_RFR
```

```
array([ 8.98333333,  8.58      ,  6.83333333, ...,  9.49733333,
       29.40266667,  7.44166667])
```

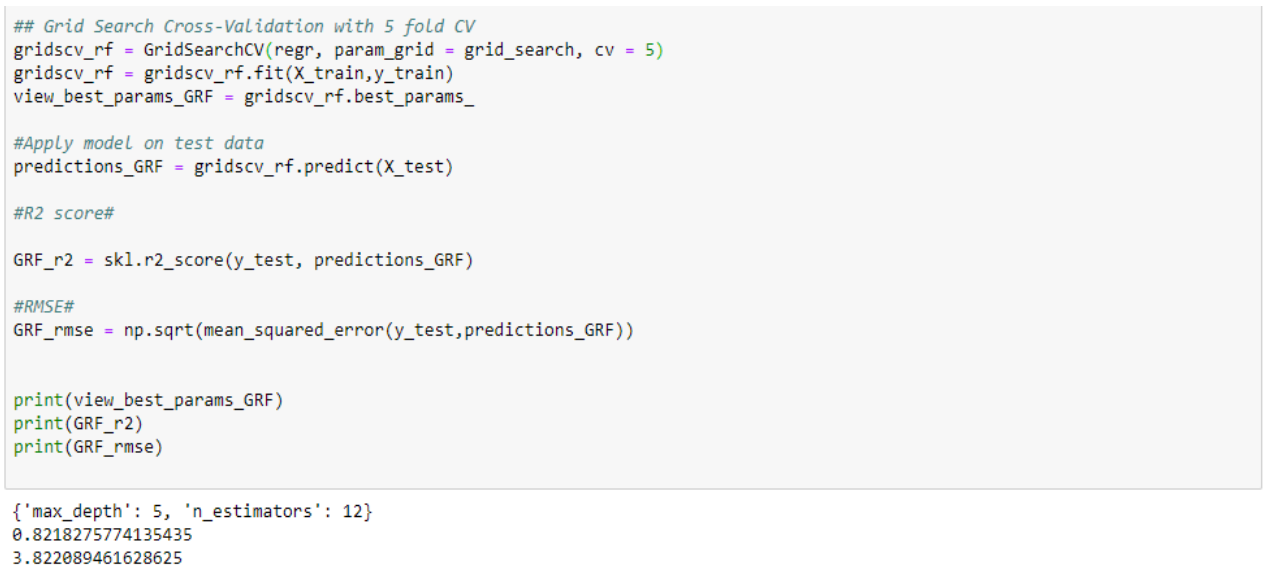
4.2 Grid Search CV

Grid search CV is arguably the most basic hyperparameter tuning method. With this technique, we simply build a model for each possible combination of all the hyperparameter values provided, evaluating each model, and selecting the architecture which produces the best results.



```
# Grid Search CV on random Forest model#

from sklearn.model_selection import GridSearchCV

regr = RandomForestRegressor(random_state = 0)
n_estimator = list(range(11,20,1))
depth = list(range(5,15,2))

# Create the grid
grid_search = {'n_estimators': n_estimator,
               'max_depth': depth}
```

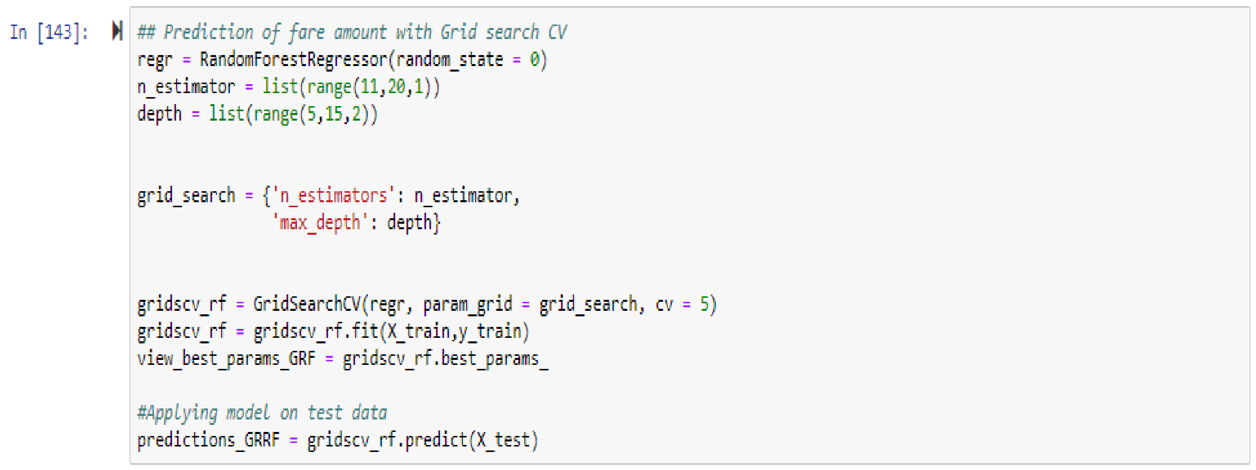
The result of using Grid search CV on random forest model is shown below:



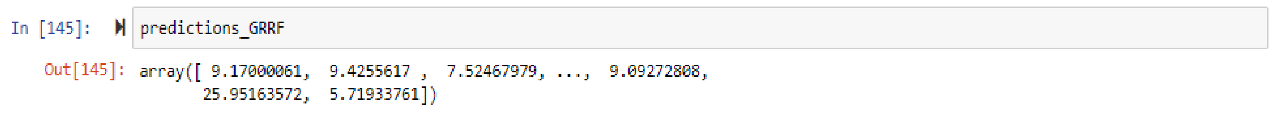
```
## Grid Search Cross-Validation with 5 fold CV
gridscv_rf = GridSearchCV(regr, param_grid = grid_search, cv = 5)
gridscv_rf = gridscv_rf.fit(X_train,y_train)
view_best_params_GRF = gridscv_rf.best_params_

#Apply model on test data
predictions_GRF = gridscv_rf.predict(X_test)

#R2 score#

GRF_r2 = skl.r2_score(y_test, predictions_GRF)

#RMSE#
GRF_rmse = np.sqrt(mean_squared_error(y_test,predictions_GRF))


print(view_best_params_GRF)
print(GRF_r2)
print(GRF_rmse)
```

```
{'max_depth': 5, 'n_estimators': 12}
0.8218275774135435
3.822089461628625
```

Prediction of fare amount using Grid search CV

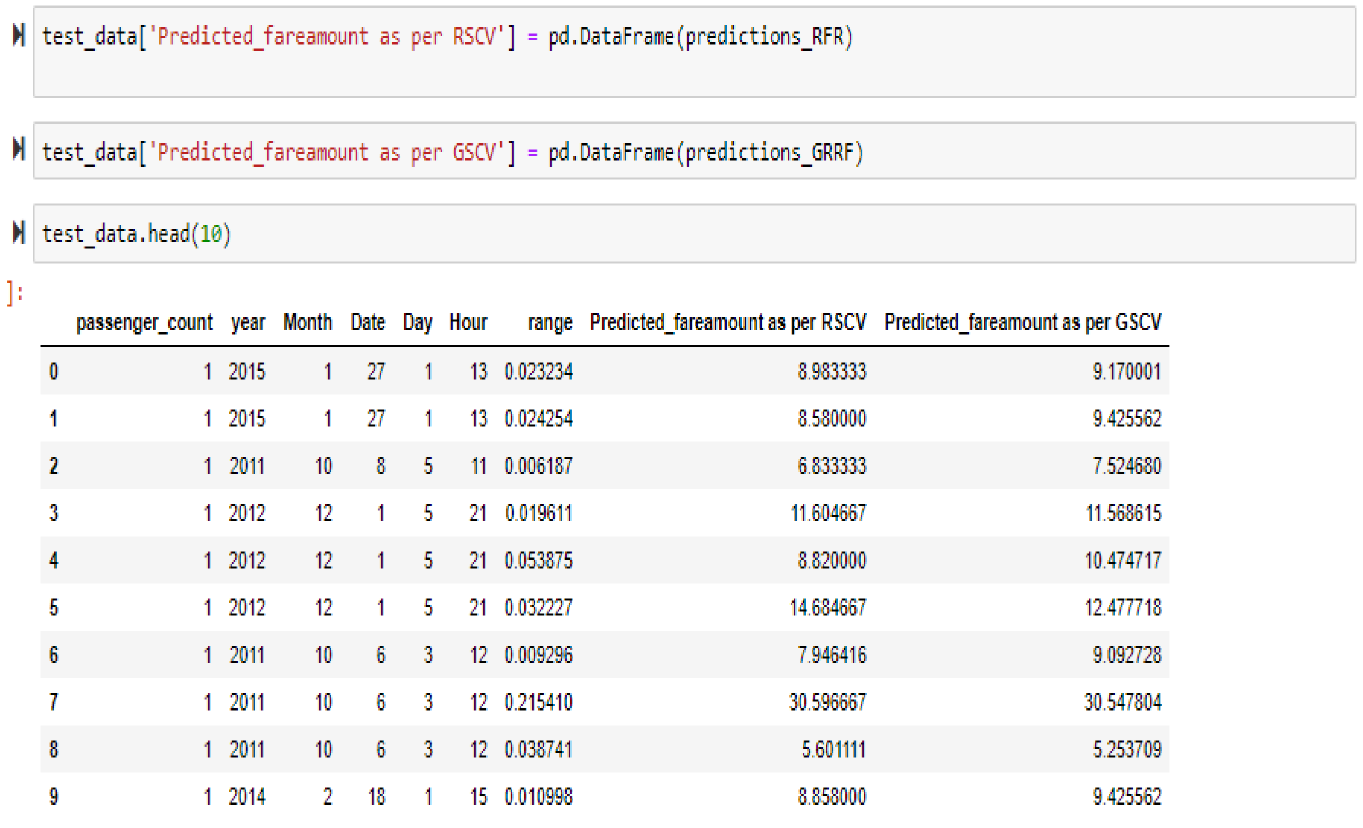


```
In [143]:  ## Prediction of fare amount with Grid search CV
           regr = RandomForestRegressor(random_state = 0)
           n_estimator = list(range(11,20,1))
           depth = list(range(5,15,2))


           grid_search = {'n_estimators': n_estimator,
                          'max_depth': depth}


           gridscv_rf = GridSearchCV(regr, param_grid = grid_search, cv = 5)
           gridscv_rf = gridscv_rf.fit(X_train,y_train)
           view_best_params_GRF = gridscv_rf.best_params_

           #Applying model on test data
           predictions_GRRF = gridscv_rf.predict(X_test)
```



```
In [145]:  predictions_GRRF

Out[145]: array([ 9.17000061,  9.4255617 ,  7.52467979, ...,  9.09272808,
                 25.95163572,  5.71933761])
```

The fare amount in test dataset has been predicted by two hyper tuning methods and their results are added in separate columns in test data frame.



```
test_data['Predicted_fareamount as per RSCV'] = pd.DataFrame(predictions_RFR)
```

```
test_data['Predicted_fareamount as per GSCV'] = pd.DataFrame(predictions_GRRF)
```

```
test_data.head(10)
```

]:

	passenger_count	year	Month	Date	Day	Hour	range	Predicted_fareamount as per RSCV	Predicted_fareamount as per GSCV
0	1	2015	1	27	1	13	0.023234	8.983333	9.170001
1	1	2015	1	27	1	13	0.024254	8.580000	9.425562
2	1	2011	10	8	5	11	0.006187	6.833333	7.524680
3	1	2012	12	1	5	21	0.019611	11.604667	11.568615
4	1	2012	12	1	5	21	0.053875	8.820000	10.474717
5	1	2012	12	1	5	21	0.032227	14.684667	12.477718
6	1	2011	10	6	3	12	0.009296	7.946416	9.092728
7	1	2011	10	6	3	12	0.215410	30.596667	30.547804
8	1	2011	10	6	3	12	0.038741	5.601111	5.253709
9	1	2014	2	18	1	15	0.010998	8.858000	9.425562

Key Observations post prediction by hyperparameter tuning techniques:

- The randomized search and the grid search explore the same space of parameters. The result in parameter settings is quite similar, while the run time for randomized search is drastically lower.

- Post evaluation and getting the values of RMSE and R^2, the performance is slightly pale for the randomized search. One possible reason might be due the random combinations of hyperparameters, incase of random search CV.

- Grid search gives the best combination, but it took a lot of time.

Hence, it can be concluded that when having a small/medium dataset it is always advisable to use Grid search CV as we simply build a model for each possible combination of all of the hyperparameter values provided, evaluating each model, and selecting the architecture which produces the best results.

On the other hand, when handling huge datasets and where the computational time is a factor, it is recommended to use Random search CV technique and to yield better results comparatively.

Chapter 5

Conclusion

Now that we have run three different regression models for predicting the target variable, we need to decide which one to choose. There are several criteria that exist for evaluating and comparing models.

In section 3.1 we have evaluated all the three models that were selected for prediction of fare amount. Out of the three error metrices we have selected RMSE and R squared value for model evaluation and selection.

On the basis RMSE and R Squared results a good model should have least RMSE and max R Squared value. As it is a time series data we will give more importance to the RMSE value.

5.1 RMSE

RMSE is one of the error measures used to calculate the predictive performance of the model. RMSE calculated for Training and test dataset using three different models:

Linear Regression Model: Train = 8.48, Test = 6.06
Decision Tree: Train = 7.16, Test = 4.46
Random Forest: Train = 2.75, Test = 3.82

Post applying hyper tuning (Random Search CV & Grid Search CV) on “test” dataset

Random Search CV on Random Forest = 4.35
Grid Search CV on Random Forest = 3.82

5.1 R^2 value

R^2 is one of the error measures used to calculate the predictive performance of the model. R^2 calculated for three different models:

Linear Regression Model: Train = 0.38, Test = 0.55
Decision Tree: Train = 0.55, Test = 0.75
Random Forest: Train = 0.93, Test = 0.82

Post applying hyper tuning (Random Search CV & Grid Search CV) on “test” dataset

Random Search CV on Random Forest = 4.35
Grid Search CV on Random Forest = 0.82

Based on the above error metrics, Random Forest is the best model for our analysis as its error metrices are also supported by Grid search CV hyper parameter tuning method.

Hence Random Forest is chosen as the model for prediction of cab fare amount.

APPENDIX A – FIGURES

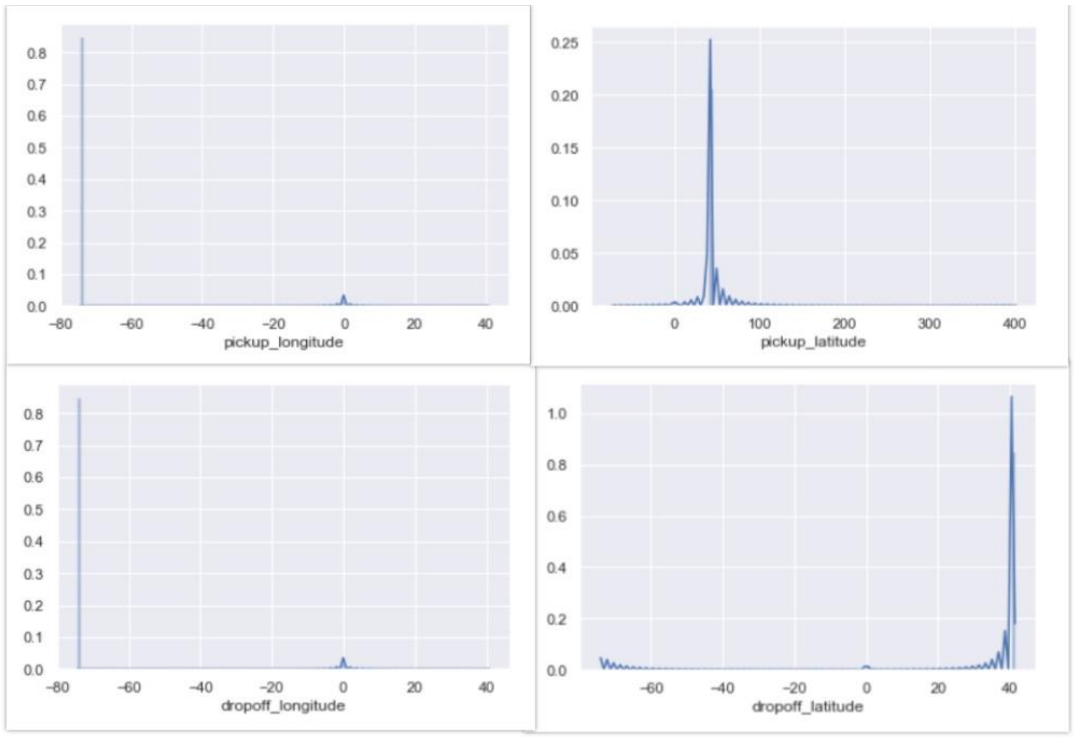


Fig. 1.1

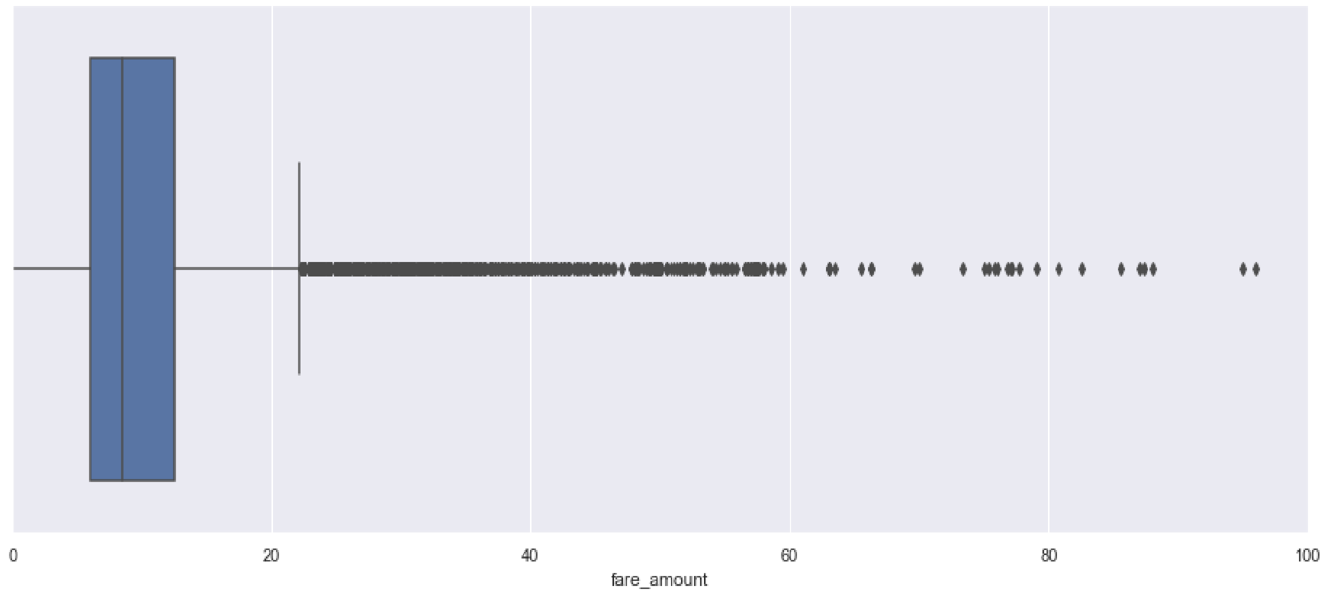


Fig. 1.3

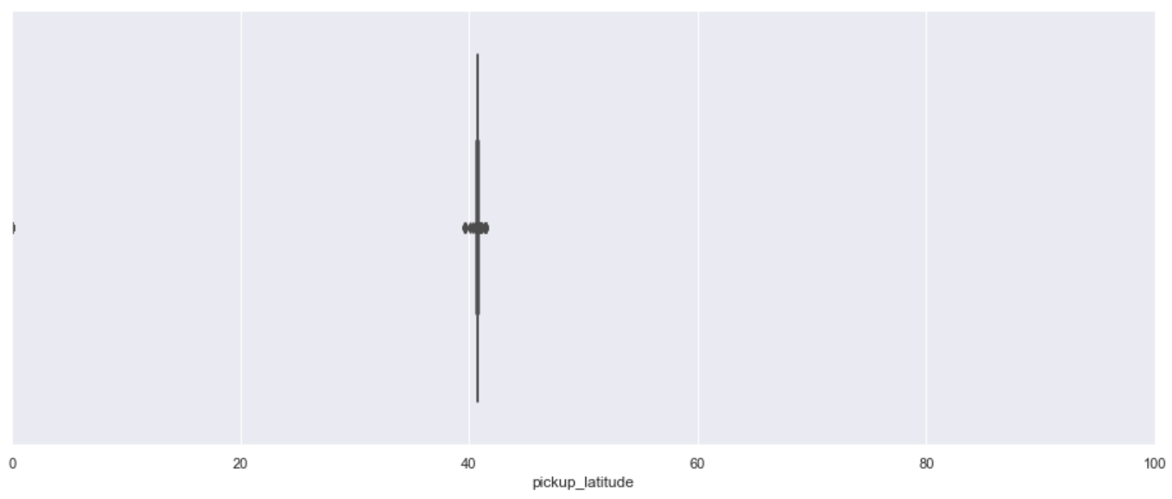


Fig. 1.4

```
#Checking outliers in range#
plt.figure(figsize=(16,6))
plt.xlim(0,150)
sns.boxplot(x=train_data['range'],data=train_data)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x5472b5ae10>
```

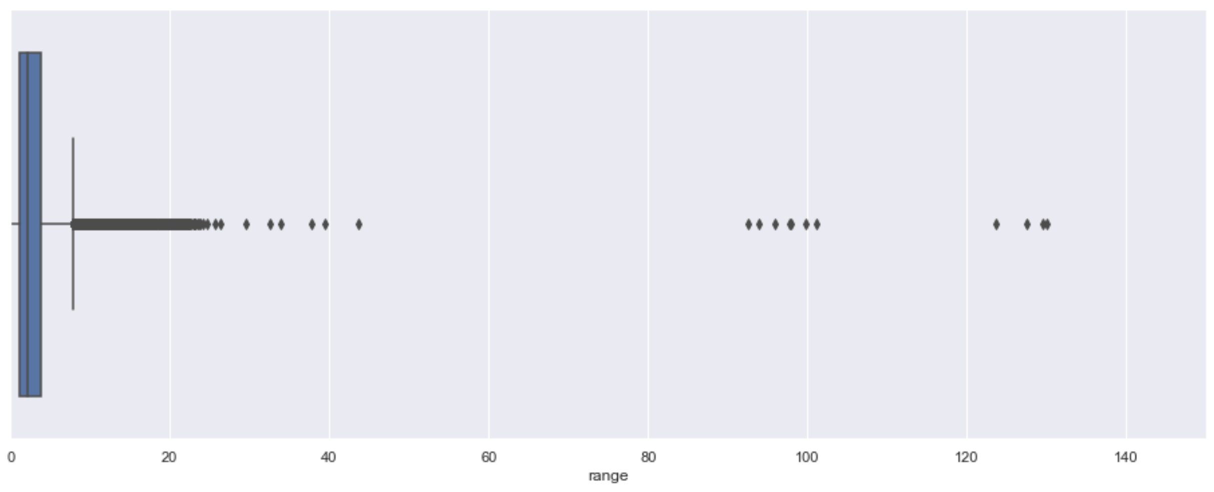


Fig. 1.5

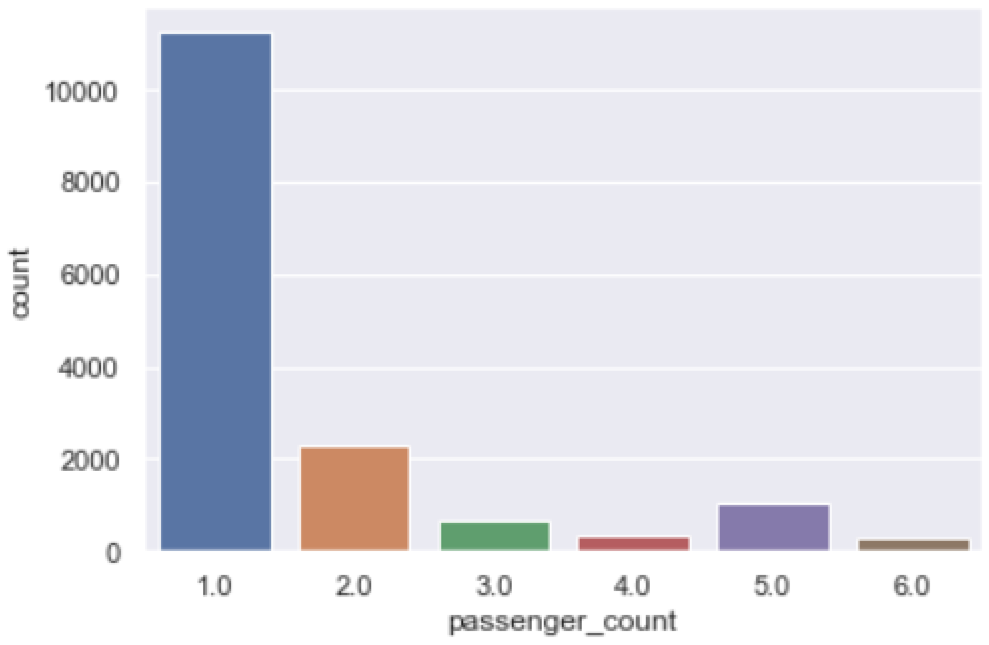

count
10000
8000
6000
4000
2000
0
1.0
2.0
3.0
4.0
5.0
6.0
passenger_count

Fig – 1.6

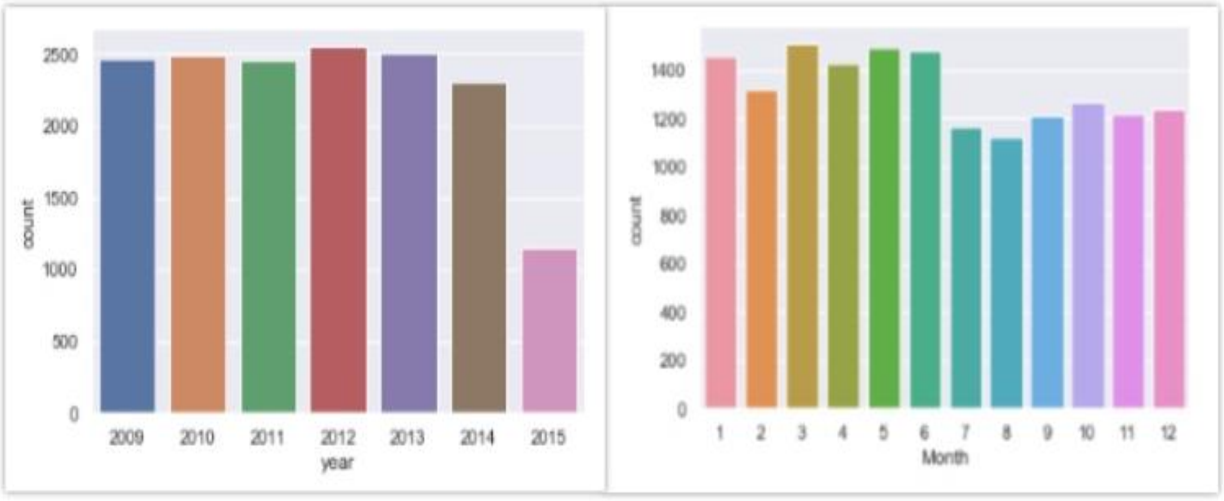

count
2500
2000
1500
1000
500
0
2009
2010
2011
2012
2013
2014
2015
year
count
1400
1200
1000
800
600
400
200
0
1
2
3
4
5
6
7
8
9
10
11
12
Month

Fig – 1.7

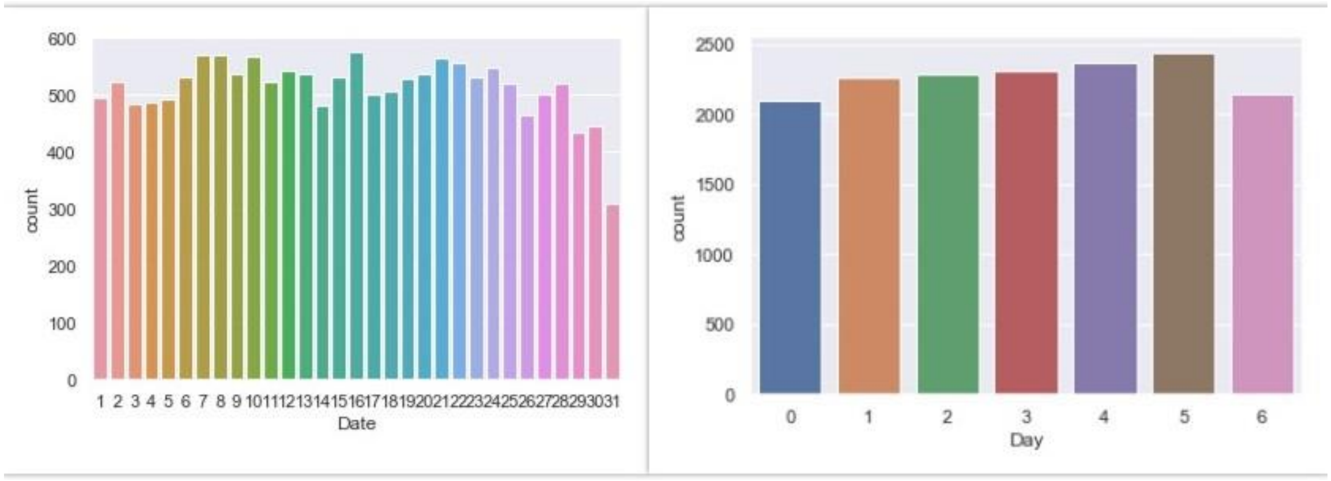


Fig – 1.8

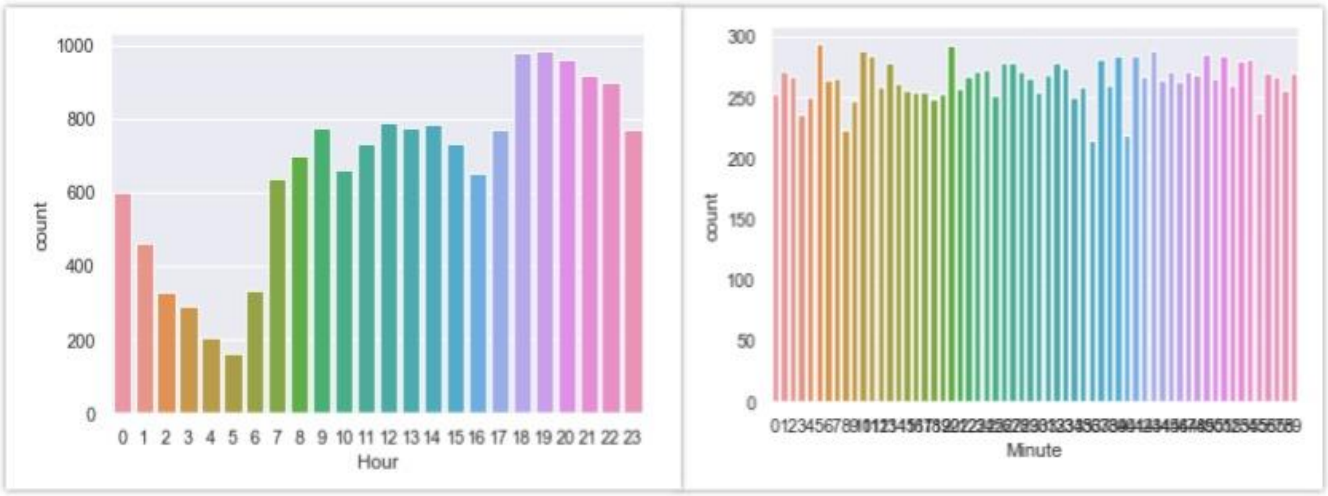


Fig – 1.9

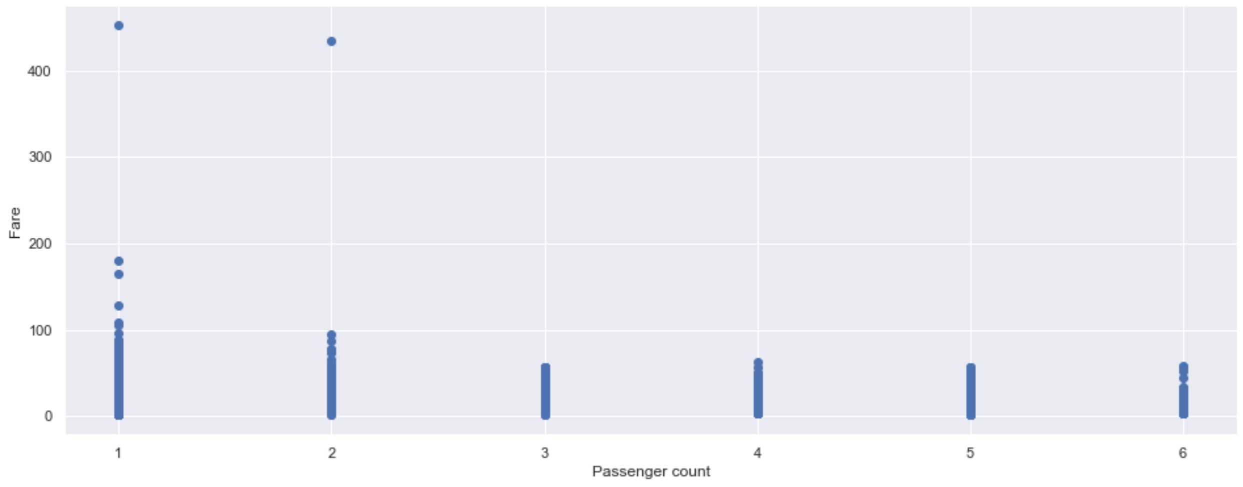


Fig – 1.10

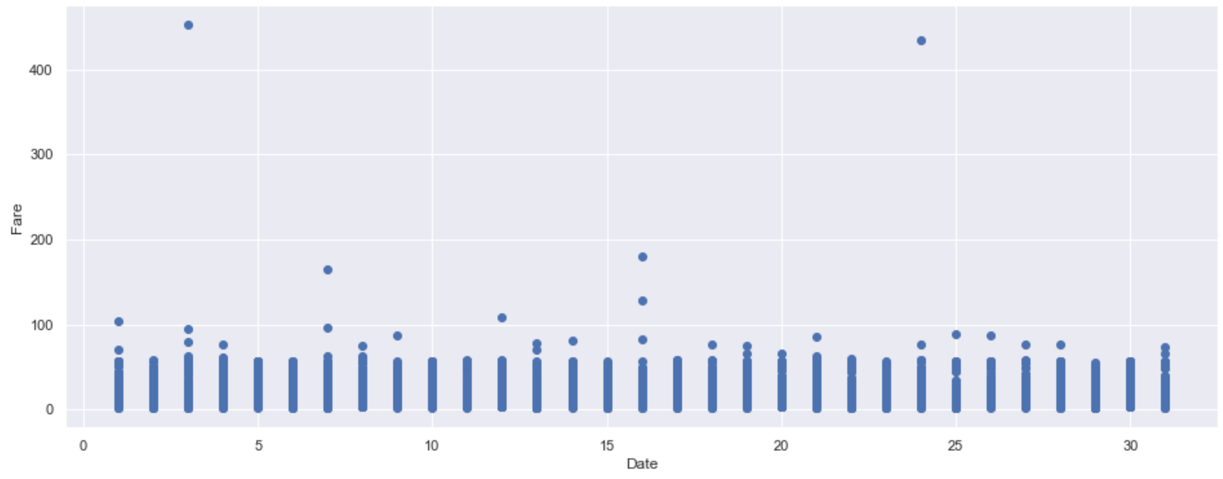


Fig – 1.11

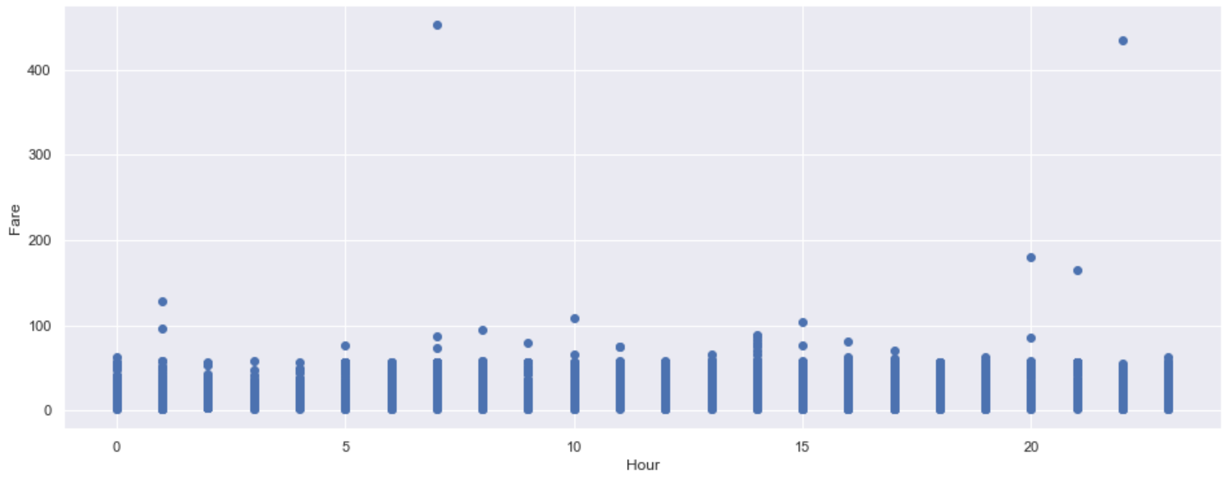
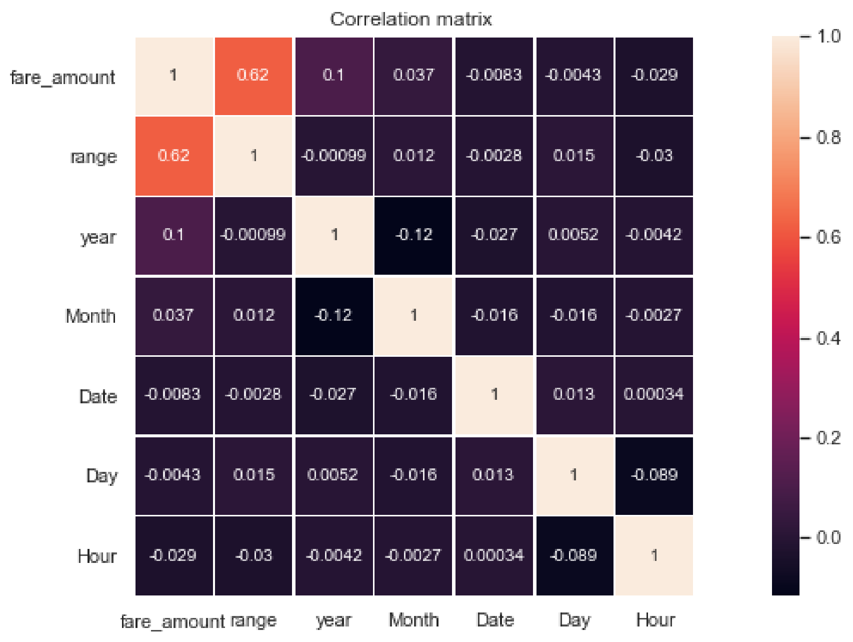


Fig – 1.12



	fare_amount	range	year	Month	Date	Day	Hour
fare_amount	1	0.62	0.1	0.037	-0.0083	-0.0043	-0.029
range	0.62	1	-0.00099	0.012	-0.0028	0.015	-0.03
year	0.1	-0.00099	1	-0.12	-0.027	0.0052	-0.0042
Month	0.037	0.012	-0.12	1	-0.016	-0.016	-0.0027
Date	-0.0083	-0.0028	-0.027	-0.016	1	0.013	0.00034
Day	-0.0043	0.015	0.0052	-0.016	0.013	1	-0.089
Hour	-0.029	-0.03	-0.0042	-0.0027	0.00034	-0.089	1

Fig – 1.13

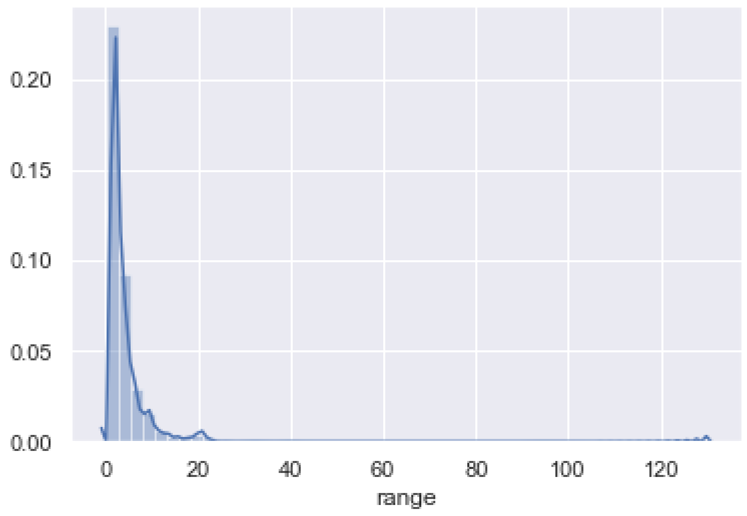


Fig. 1.14

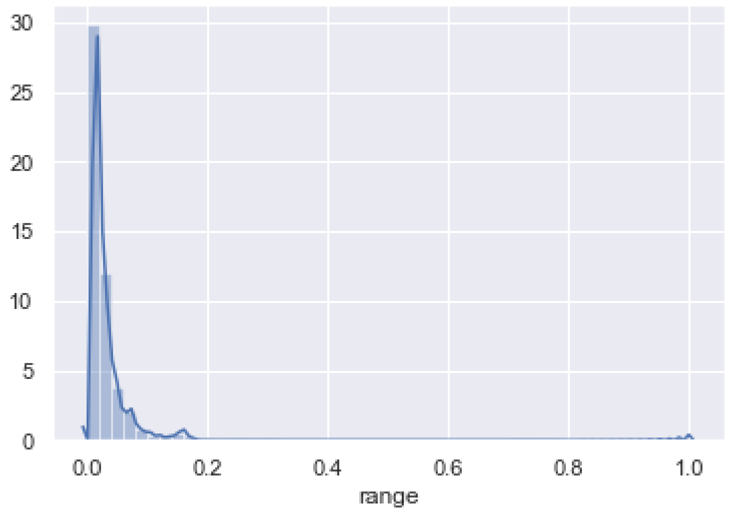
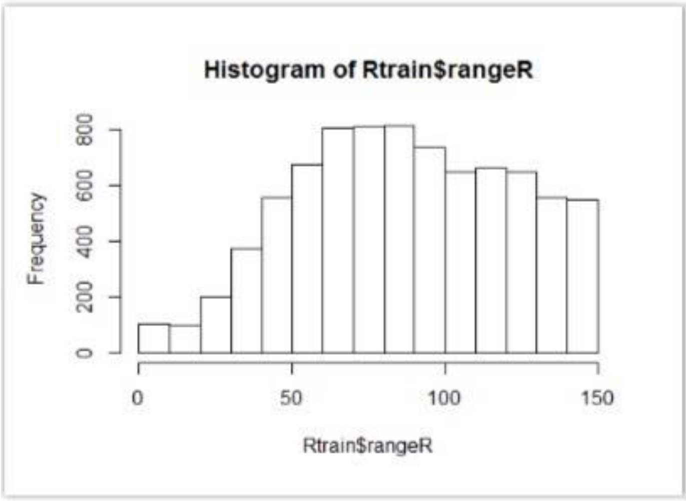
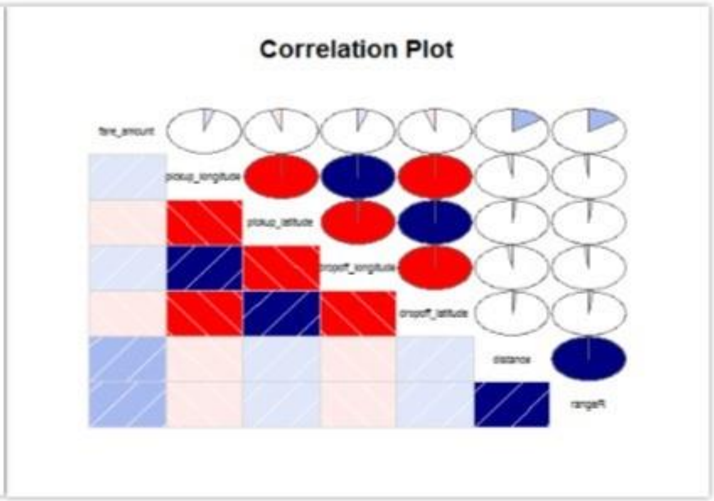


Fig. 1.15

<u>Some useful plots obtained from R</u>





APPENDIX B – Python code

The Python code implemented here is given in a separate pdf file, **'ANNEXURE A'**, attached herewith the project.

References

1. Edwisor lecture videos by Muquayyar Ahmed

2. https://en.wikipedia.org/wiki/Haversine_formula

3. https://www.kdnuggets.com/2018/12/feature-engineering-explained.html

4. https://seaborn.pydata.org/tutorial/distributions.html

5. https://www.geeksforgeeks.org/ml-hyperparameter-tuning/

6. https://www.jeremyjordan.me/hyperparameter-tuning/

7. https://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html

8. https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

9. https://towardsdatascience.com/how-to-select-the-right-evaluation-metric-for-machine-learning-models-part-1-regrression-metrics-3606e25beae0

10. https://towardsdatascience.com/how-to-select-the-right-evaluation-metric-for-machine-learning-models-part-2-regression-metrics-d4a1a9ba3d74