EDWISOR DATA SCIENCE PROJECT

CAB FARE PREDICTION

PREDICTION OF FARE AMOUNT FOR A CAB RIDE IN THE CITY

ABSTRACT

This Case study is to predict the fare amount for a cab ride in the city based on historical data from a pilot project.

ABHISHEK VISHWAKARMA

8/4/2019

INDEX

OVERVIEW

Now a day's cab rental services are increasing at exponential rates. The flexibility and ease of using the service, gives customers a great experience at affordable prices.

CHAPTER 1: INTRODUCTION

1.1 PROBLEM STATEMENT

The objective of this Case is Predication of fare amount for a cab ride in the city.

The prediction is based on a successfully run pilot project and the historical data from your pilot project is being used to apply analytics for fare prediction.

1.2 DATA

Understanding data is the first and most important step to find solutions to any business problem.

Here in our case we have a data set with following features

, we need to go through each and every variable of it to understand and for better functioning.

Below mentioned is a list of all the variable names with their meanings:

Variables	***Description***
Fare_Amount	***Fare Amount***
Pickup_Datetime	*Cab Pickup Date With Time*
Pickup_Longitude	*Pickup Location Longitude*
Pickup_Latitude	*Pickup Location Latitude*
Dropoff_Longitude	*Drop Location Longitude*
Dropoff_Latitude	*Drop Location Latitude*
Passenger_Count	*Number Of Passengers Sitting In The Cab*

Table 1- Dataset variables

Size of Dataset: - 16067 rows, 7 Columns (includes dependent variable)
Missing Values: Yes
Outliers Present: Yes

Sample data:

fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
4.5	2009-06-15 17:26:21 UTC	-73.844311	40.721319	-73.84161	40.712278	1
16.9	2010-01-05 16:52:16 UTC	-74.016048	40.711303	-73.979268	40.782004	1
5.7	2011-08-18 00:35:00 UTC	-73.982738	40.76127	-73.991242	40.750562	2
7.7	2012-04-21 04:30:42 UTC	-73.98713	40.733143	-73.991567	40.758092	1
5.3	2010-03-09 07:51:00 UTC	-73.968095	40.768008	-73.956655	40.783762	1

Table 2- Sample Data

CHAPTER 2: METHODOLOGY FOLLOWED

2.1 PRE-PROCESSING

For a predictive model, before modelling the data should be checked and manipulated under ***Exploratory Data Analysis***. It includes following steps:

1. Data exploration and Cleaning
2. Missing values treatment
3. Outlier Analysis
4. Feature Selection
5. Features Scaling
    - Skewness and Log transformation
6. Visualization

2.2 MODELLING

After Pre-Processing the next step is modelling. Modelling helps to find out the good inferences from the data.

Choice of models depends upon the problem statement and data set. Based on our problem statement and dataset, following models are tested on our pre-processed data:

1. Linear regression
2. Decision Tree
3. Random forest,
4. Gradient Boosting

Post comparison of the output results we select the best suitable model for our problem.

We have also used hyper parameter tunings to check the parameters on which our model runs best. Following are two techniques of hyper parameter tuning we have used:

1. Random Search CV
2. Grid Search CV

2.3 MODEL SELECTION

The final step of our methodology is Model Selection.

It is based on the outputs and results shown by different models. Multiple parameters are also studied to further scrutinize the model feasibility for the problem statement.

CHAPTER 3: PRE-PROCESSING

3.1 DATA EXPLORATION & CLEANING (MISSING VALUES & OUTLIERS)

Following are the steps followed to explore and clean the data we have:

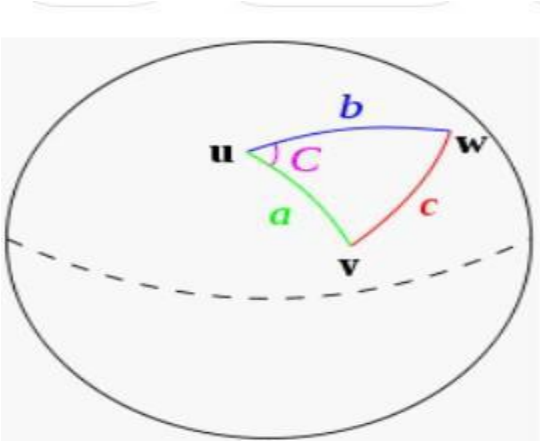
1. Combined variable separation.
2. Negative values in fare amount to be removed.
3. Passenger count cannot exceed 6 (SUV vehicle), the rows having passengers counts more than 6 and less than 1 to be removed.
4. Outlier figures in the fare (like top 3 values) to be removed.
5. Latitudes range from -90 to 90. Longitudes range from -180 to 180. To remove the rows beyond above ranges.

3.2 CREATING NEW VARIABLES FROM GIVEN VARIABLES

Here in our data set our variable name pickup_datetime contains date and time for pickup. So we tried to extract some important data in form of variables from pickup_datetime:

1. Year
2. Month
3. Date
4. Day of Week
5. Hour
6. Minute

Using the haversine formula we tried to find out the distance:
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.



So our new extracted variables are:

1. fare_amount
2. pickup_datetime
3. pickup_longitude
4. pickup_latitude
5. dropoff_longitude
6. dropoff_latitude
7. passenger_count
8. year
9. Month
10. Date
11. Day of Week
12. Hour
13. Minute
14. Distance

3.3 SELECTION OF VARIABLES

Now as we know that all above variables are of now use so we will drop the redundant variables:

1. pickup_datetime
2. pickup_longitude
3. pickup_latitude
4. dropoff_longitude
5. dropoff_latitude
6. Minute

Below is the data being used after removing not useful variables:

	fare_amount	passenger_count	year	Month	Date	Day	Hour	distance
0	4.5	1.0	2009.0	6.0	15.0	0.0	17.0	1.030764
1	16.9	1.0	2010.0	1.0	5.0	1.0	16.0	8.450134
2	5.7	2.0	2011.0	8.0	18.0	3.0	0.0	1.389525
3	7.7	1.0	2012.0	4.0	21.0	5.0	4.0	2.799270
4	5.3	1.0	2010.0	3.0	9.0	1.0	7.0	1.999157

Table 3 - Data after removing redundant variables

```
fare_amount        float64
passenger_count      int64
year                 int64
Month                int64
Date                 int64
Day                  int64
Hour                 int64
distance           float64
dtype: object
```

Table 4 - Data types of the variables

3.4 SOME MORE DATA EXPLORATION

In this report we are trying to predict the fare prices of a cab rental company. So here we have a data set of 16067 observations with 8 variables including one dependent variable.

Below are the names of Independent variables:
passenger_count, year, Month, Date, Day of Week, Hour, distance
Our Dependent variable is: **fare_amount**

3.4.1 Uniqueness in Variable

We need to look at the unique number in the variables which help us to decide whether the variable is categorical or numeric. So, by using python script ‘nunique’ we tried to find out the unique values in each variable. We have also added the table below:

Variable Name	Unique Counts
fare_amount	450
passenger_count	7
year	7
Month	12
Date	31
Day of Week	7
Hour	24
distance	15424

Table 5- Unique elements

3.4.2 Dividing the variables into two categories basis their data types:

Continuous variables - 'fare_amount', 'distance'.
Categorical Variables - 'year', 'Month', 'Date', 'Day of Week', 'Hour', 'passenger_count'

3.5 FEATURE SCALING

Skewness is asymmetry in a statistical distribution, in which the curve appears distorted or skewed either to the left or to the right.
Skewness can be quantified to define the extent to which a distribution differs from a normal distribution. Here by using **log transform** technique we tried to reduce the skewness of the same.

Below mentioned graphs shows the probability distribution plot to check distribution before log transformation:

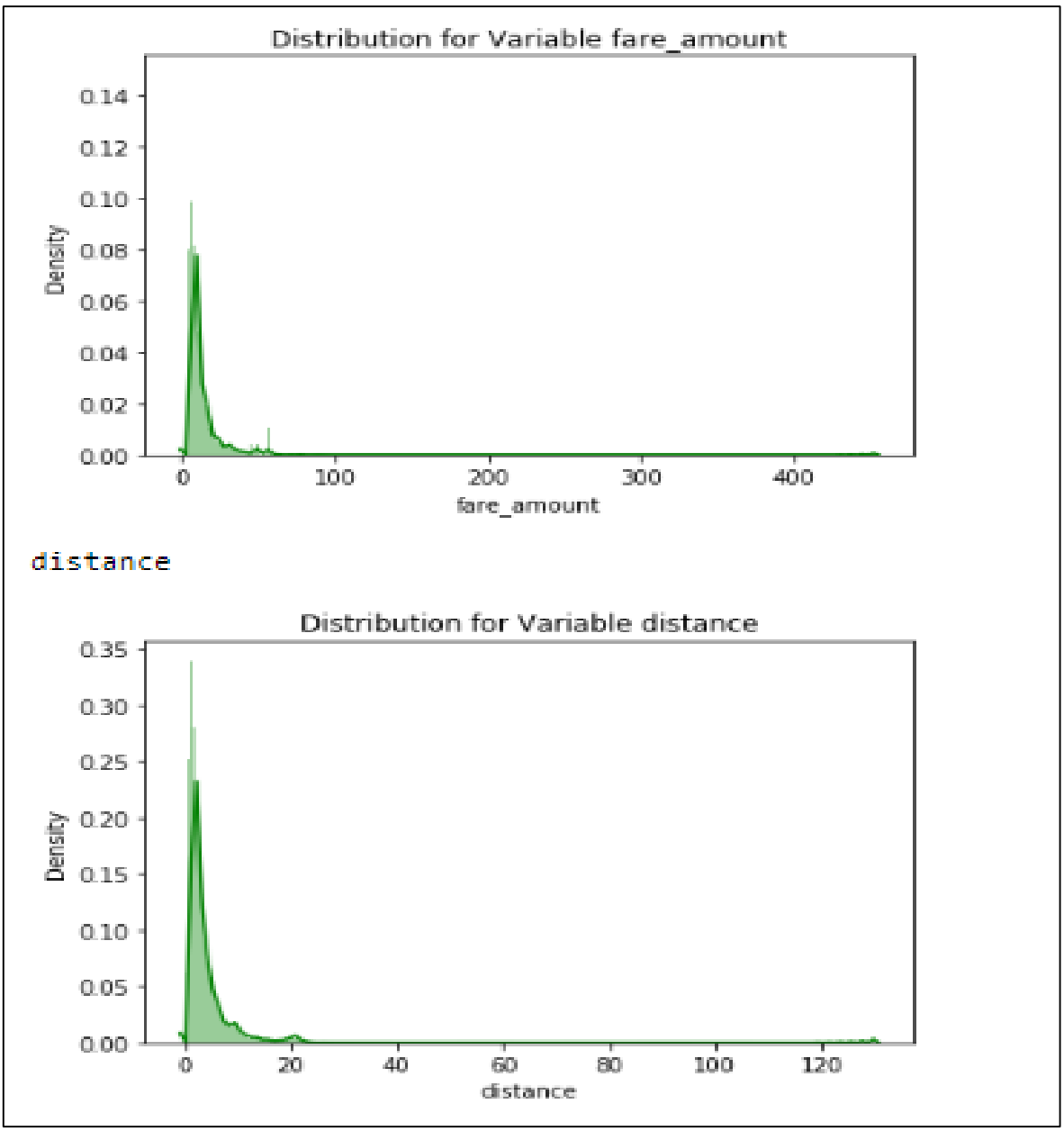


Figure 1 - Probability distribution before log transformation

Below mentioned graphs shows the probability distribution plot to check distribution after log transformation:

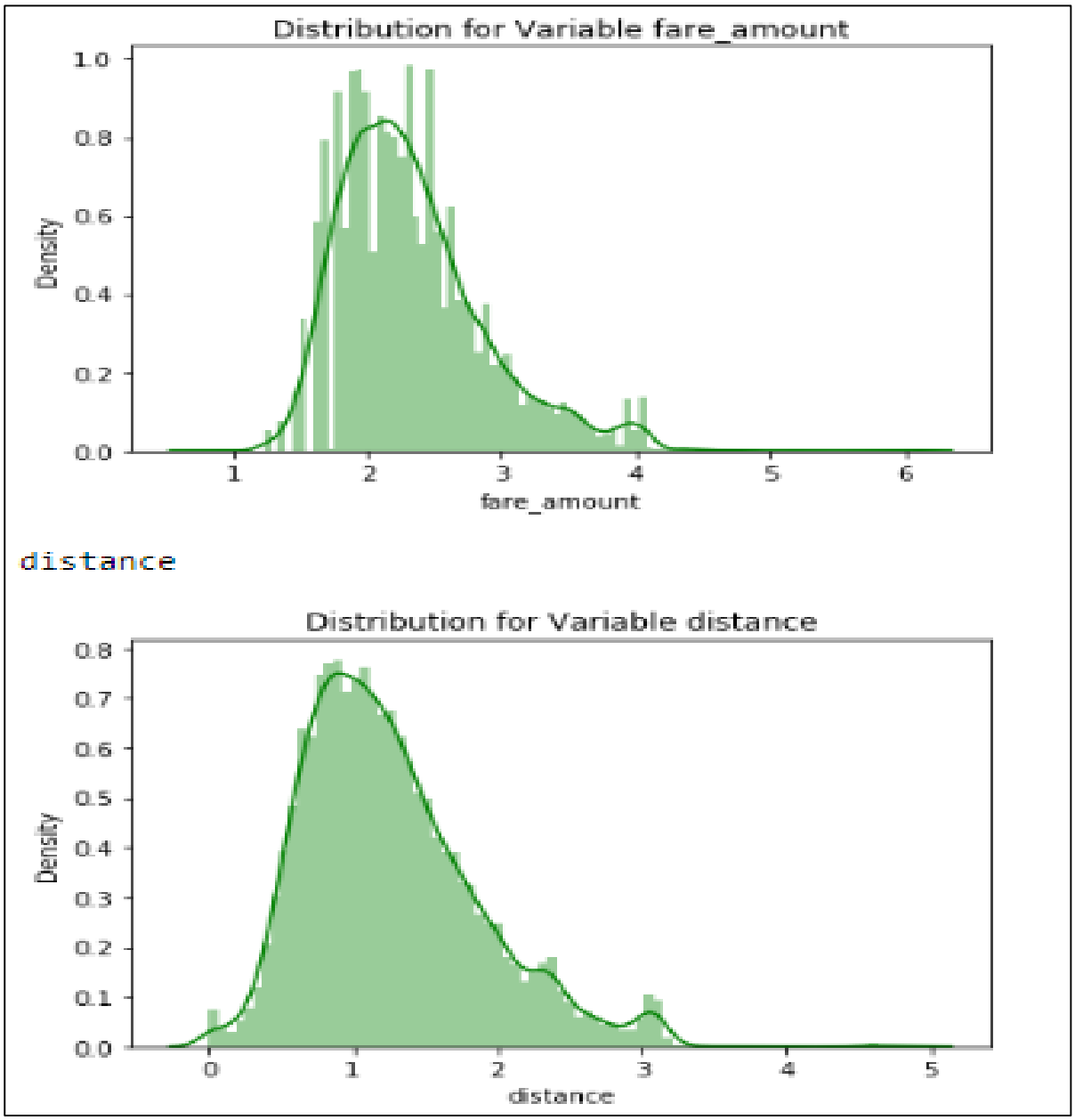


Figure 2-Probability distribution after log transformation

As our continuous variables appears to be normally distributed so we don't need to use feature scaling techniques like normalization and standardization for the same.

CHAPTER 4: MODELLING

After a thorough pre-processing, we will use some regression models on our processed data to predict the target variable.

Following are the models which we have built: –
Linear Regression
Decision Tree
Random Forest
Gradient Boosting

Note : 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. Below is the snipped image of the split of train test.

We need to split our train data into two parts

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
```

```
##train test split for further modelling
X_train, X_test, y_train, y_test = train_test_split( train_df.iloc[:, train_df.columns != 'fare_amount'],
                      train_df.iloc[:, 0], test_size = 0.20, random_state = 1)
```

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

```
(12339, 7)
(3085, 7)
```

Figure 3- Code snippet

4.1 LINEAR REGRESSION

Multiple regression is an extension of simple linear regression. It is used as a predictive analysis, when we want to predict the value of a variable based on the value of two or more other variables. The variable we want to predict is called the dependent variable (or sometimes, the outcome, target or criterion variable).

Below is the screenshot of the query we executed and the result shown

Linear Regression Model

```
# Building model on top of training dataset
fit_LR = LinearRegression().fit(X_train , y_train)

#prediction on train data
pred_train_LR = fit_LR.predict(X_train)

#prediction on test data
pred_test_LR = fit_LR.predict(X_test)

##calculating RMSE for test data
RMSE_test_LR = np.sqrt(mean_squared_error(y_test, pred_test_LR))

##calculating RMSE for train data
RMSE_train_LR= np.sqrt(mean_squared_error(y_train, pred_train_LR))

print("Root Mean Squared Error For Training data = "+str(RMSE_train_LR))
print("Root Mean Squared Error For Test data = "+str(RMSE_test_LR))

##calculate R^2 for train data
from sklearn.metrics import r2_score
r2_score(y_train, pred_train_LR)
```

```
Root Mean Squared Error For Training data = 0.27531100179673135
Root Mean Squared Error For Test data = 0.24540661786977466

0.7495502651880406
```

Figure 4- Linear regression

4.2 DECISION TREE

A tree has many analogies in real life, and turns out that it has influenced a wide area of machine learning, covering both classification and regression. In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. As the name goes, it uses a tree-like model of decisions.
Below is the screenshot of the query we executed and the result shown

Decision tree Model :

```
fit_DT = DecisionTreeRegressor(max_depth = 2).fit(X_train,y_train)

#prediction on train data
pred_train_DT = fit_DT.predict(X_train)

#prediction on test data
pred_test_DT = fit_DT.predict(X_test)

##calculating RMSE for train data
RMSE_train_DT = np.sqrt(mean_squared_error(y_train, pred_train_DT))

##calculating RMSE for test data
RMSE_test_DT = np.sqrt(mean_squared_error(y_test, pred_test_DT))

print("Root Mean Squared Error For Training data = "+str(RMSE_train_DT))
print("Root Mean Squared Error For Test data = "+str(RMSE_test_DT))

## R^2 calculation for train data
r2_score(y_train, pred_train_DT)
```

```
Root Mean Squared Error For Training data = 0.29962109020770195
Root Mean Squared Error For Test data = 0.28674606171586153

0.7033678616157002
```

Figure 5- Decision tree

4.3 RANDOM FOREST

Random forests or random decision forests are an ensemble learning method for classification, regression and other task, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set.

To say it in simple words: Random forest builds multiple decision trees and merges them together to get a more accurate and stable prediction.

Below is a screenshot of the model we build and its output:

Random Forest Model : ¶

```
fit_RF = RandomForestRegressor(n_estimators = 200).fit(X_train,y_train)

#prediction on train data
pred_train_RF = fit_RF.predict(X_train)

#prediction on test data
pred_test_RF = fit_RF.predict(X_test)

##calculating RMSE for train data
RMSE_train_RF = np.sqrt(mean_squared_error(y_train, pred_train_RF))

##calculating RMSE for test data
RMSE_test_RF = np.sqrt(mean_squared_error(y_test, pred_test_RF))

print("Root Mean Squared Error For Training data = "+str(RMSE_train_RF))
print("Root Mean Squared Error For Test data = "+str(RMSE_test_RF))

## calculate R^2 for train data
r2_score(y_train, pred_train_RF)
```

```
Root Mean Squared Error For Training data = 0.09588764987823241
Root Mean Squared Error For Test data = 0.23496810817523822

0.9696192289485864
```

Figure 6- Random forest

4.4 GRADIENT BOOSTING

Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

Below is a screenshot of the model we build and its output:

Gradient Boosting :

```
fit_GB = GradientBoostingRegressor().fit(X_train, y_train)

#prediction on train data
pred_train_GB = fit_GB.predict(X_train)

#prediction on test data
pred_test_GB = fit_GB.predict(X_test)

##calculating RMSE for train data
RMSE_train_GB = np.sqrt(mean_squared_error(y_train, pred_train_GB))

##calculating RMSE for test data
RMSE_test_GB = np.sqrt(mean_squared_error(y_test, pred_test_GB))

print("Root Mean Squared Error For Training data = "+str(RMSE_train_GB))
print("Root Mean Squared Error For Test data = "+str(RMSE_test_GB))

#calculate R^2 for test data
r2_score(y_test, pred_test_GB)
```

```
Root Mean Squared Error For Training data = 0.22754316149645537
Root Mean Squared Error For Test data = 0.22754218213548194

0.8131869525110339
```

4.5 HYPER PARAMETERS TUNINGS FOR OPTIMIZING THE RESULTS

Model hyperparameters are set by the data scientist ahead of training and control implementation aspects of the model. The weights learned during training of a linear regression model are parameters while the number of trees in a random forest is a model hyperparameter because this is set by the data scientist.

Hyperparameters can be thought of as model settings. These settings need to be tuned for each problem because the best model hyperparameters for one particular dataset will not be the best across all datasets. The process of hyperparameter tuning (also called hyperparameter optimization) means finding the combination of hyperparameter values for a machine learning model that performs the best - as measured on a validation dataset - for a problem.

Here we have used two hyper parameters tuning techniques

1. **Random Search CV**: This algorithm set 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 time/resources.

2. **Grid Search CV**: This algorithm set up a grid of hyperparameter values and for each combination, train a model and score on the validation data. In this approach, every single combination of hyperparameters values is tried which can be very inefficient.

Check results after using Random Search CV on Random forest and gradient boosting model.

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

RRF = 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(RRF, param_distributions = rand_grid, n_iter = 5, cv = 5, random_state=0)
randomcv_rf = randomcv_rf.fit(X_train,y_train)
predictions_RRF = randomcv_rf.predict(X_test)

view_best_params_RRF = randomcv_rf.best_params_

best_model = randomcv_rf.best_estimator_

predictions_RRF = best_model.predict(X_test)

#R^2
RRF_r2 = r2_score(y_test, predictions_RRF)
#Calculating RMSE
RRF_rmse = np.sqrt(mean_squared_error(y_test,predictions_RRF))

print('Random Search CV Random Forest Regressor Model Performance:')
print('Best Parameters = ',view_best_params_RRF)
print('R-squared = {:0.2}.'.format(RRF_r2))
print('RMSE = ',RRF_rmse)
```

```
Random Search CV Random Forest Regressor Model Performance:
Best Parameters =  {'n_estimators': 15, 'max_depth': 9}
R-squared = 0.8.
RMSE =  0.23730781853507238
```

```
##Random Search CV on gradient boosting model

gb = GradientBoostingRegressor(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_gb = RandomizedSearchCV(gb, param_distributions = rand_grid, n_iter = 5, cv = 5, random_state=0)
randomcv_gb = randomcv_gb.fit(X_train,y_train)
predictions_gb = randomcv_gb.predict(X_test)

view_best_params_gb = randomcv_gb.best_params_

best_model = randomcv_gb.best_estimator_

predictions_gb = best_model.predict(X_test)

#R^2
gb_r2 = r2_score(y_test, predictions_gb)
#Calculating RMSE
gb_rmse = np.sqrt(mean_squared_error(y_test,predictions_gb))

print('Random Search CV Gradient Boosting Model Performance:')
print('Best Parameters = ',view_best_params_gb)
print('R-squared = {:0.2}.'.format(gb_r2))
print('RMSE = ', gb_rmse)
```

```
Random Search CV Gradient Boosting Model Performance:
Best Parameters =  {'n_estimators': 15, 'max_depth': 9}
R-squared = 0.77.
RMSE =  0.25199340493550487
```

Check results on using Grid Search CV on Random forest and gradient boosting model:

```
## Grid Search CV for random Forest model
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}

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

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

#R^2
GRF_r2 = r2_score(y_test, predictions_GRF)
#Calculating RMSE
GRF_rmse = np.sqrt(mean_squared_error(y_test,predictions_GRF))

print('Grid Search CV Random Forest Regressor Model Performance:')
print('Best Parameters = ',view_best_params_GRF)
print('R-squared = {:0.2}.'.format(GRF_r2))
print('RMSE = ',(GRF_rmse))
```

```
Grid Search CV Random Forest Regressor Model Performance:
Best Parameters =  {'max_depth': 7, 'n_estimators': 18}
R-squared = 0.8.
RMSE =  0.23637990451376567
```

```
## Grid Search CV for gradinet boosting
gb = GradientBoostingRegressor(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}

## Grid Search Cross-Validation with 5 fold CV
gridcv_gb = GridSearchCV(gb, param_grid = grid_search, cv = 5)
gridcv_gb = gridcv_gb.fit(X_train,y_train)
view_best_params_Ggb = gridcv_gb.best_params_

#Apply model on test data
predictions_Ggb = gridcv_gb.predict(X_test)

#R^2
Ggb_r2 = r2_score(y_test, predictions_Ggb)
#Calculating RMSE
Ggb_rmse = np.sqrt(mean_squared_error(y_test,predictions_Ggb))

print('Grid Search CV Gradient Boosting regression Model Performance:')
print('Best Parameters = ',view_best_params_Ggb)
print('R-squared = {:0.2}.'.format(Ggb_r2))
print('RMSE = ',(Ggb_rmse))
```

```
Grid Search CV Gradient Boosting regression Model Performance:
Best Parameters =  {'max_depth': 5, 'n_estimators': 19}
R-squared = 0.8.
RMSE =  0.23724212611002213
```

CHAPTER 5: CONCLUSION

5.1 Model Evaluation

The main concept is residuals or difference between our predictions f(x[I,]) and actual outcomes y[i].

In general, we scientists use two methods to evaluate the performance of the model:

5.1.1 RMSE (Root Mean Square Error):

It is a frequently used measure of the difference between values predicted by a model and the values actually observed from the environment that is being modelled.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (X_{obs,i} - X_{mo\ del,i})^2}{n}}$$

5.1.2 R Squared(R^2):

It is a statistical measure of how close the data are to the fitted regression line. It is also known as the coefficient of determination, or the coefficient of multiple determination for multiple regression. In other words, we can say it explains as to how much of the variance of the target variable is explained.

Below table shows the model results before applying hyper tuning:

Model Name	RMSE		R Squared	
	Train	Test	Train	Test
`Linear Regression	0.27	0.25	0.74	0.77
Decision Tree	0.30	0.28	0.70	0.70
Random Forest model	0.09	0.23	0.96	0.79
Gradient Boosting	0.22	0.22	0.82	0.81

Table 6- Pre Hyper tuning

Below table shows results post using hyper parameter tuning techniques:

Model Name	Parameter	RMSE (Test)	R Squared (Test)
Random Search CV	Random Forest	0.24	0.79
	Gradient Boosting	0.25	0.77
Grid Search CV	Random Forest	0.23	0.80
	Gradient Boosting	0.24	0.79

Table 7- Post Hyper tuning

Above table shows the results after tuning the parameters of our two best suited models i.e. Random Forest and Gradient Boosting. For tuning the parameters, we have used Random Search CV and Grid Search CV under which we have given the range of n_estimators, depth and CV folds.

5.2 MODEL SELECTION

So, from above tables we can see:

- Both the models- Gradient Boosting Default and Random Forest perform comparatively well while comparing their RMSE and R-Squared value.
- After this, I chose Random Forest CV and Grid Search CV to apply cross validation technique and see changes brought about by that.
- After applying tunings Random forest model shows best results compared to gradient boosting.
- So finally, we can say that Random forest model is the best method to make prediction for this project with highest explained variance of the target variables and lowest error chances with parameter tuning technique Grid Search CV.

Finally, I used **RANDOM FOREST MODEL** method to predict the target variable for the test data file shared in the problem statement.

5.3 PREDICTOR VS PREDICED VARIABLES

5.3.1 Passengers VS Fare

Single passengers are the most frequent travellers, and the highest fare payers too.

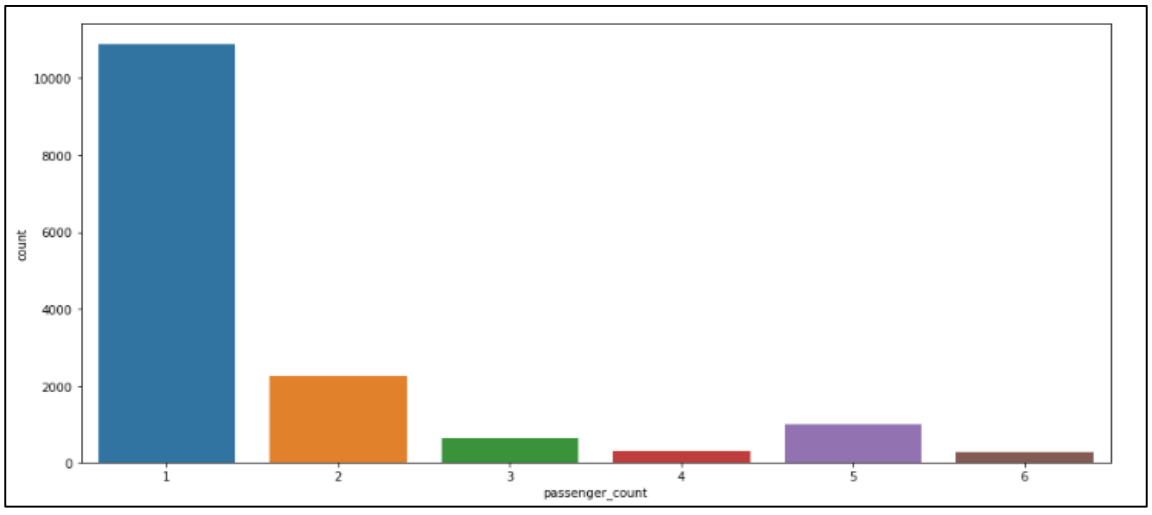


Figure 7-Passenger vs No. Of Passengers

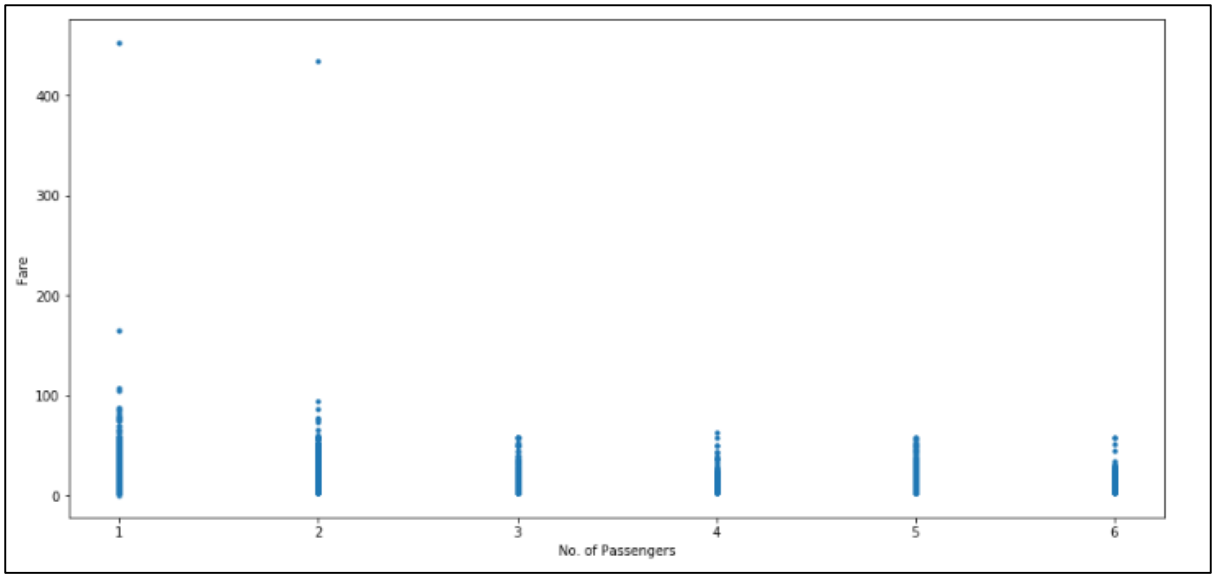


Figure 8-Passenger vs Fare

5.3.2 Date Of Month VS Fares

The fares throughout the month mostly seem uniform.

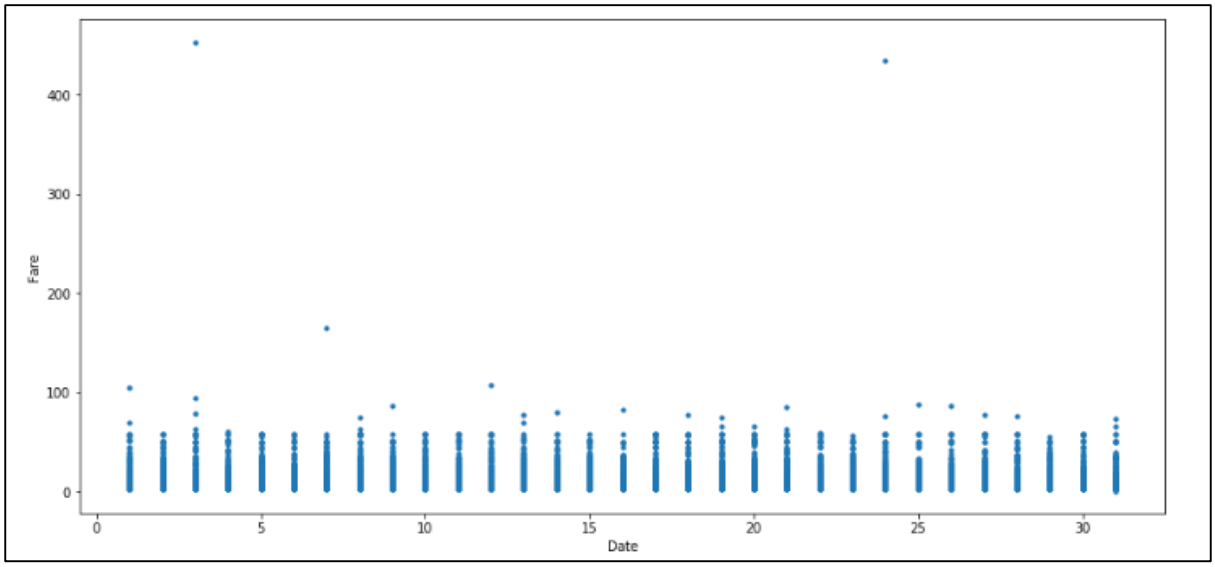


Figure 9- Date vs Fare

5.3.3 Hours VS Fares // Hours VS Frequency

• During hours 6 PM to 11PM the frequency of cab boarding is very due to peak hours

• Fare prices during 2PM to 8PM is bit high compared to all other time might be due to high demands.

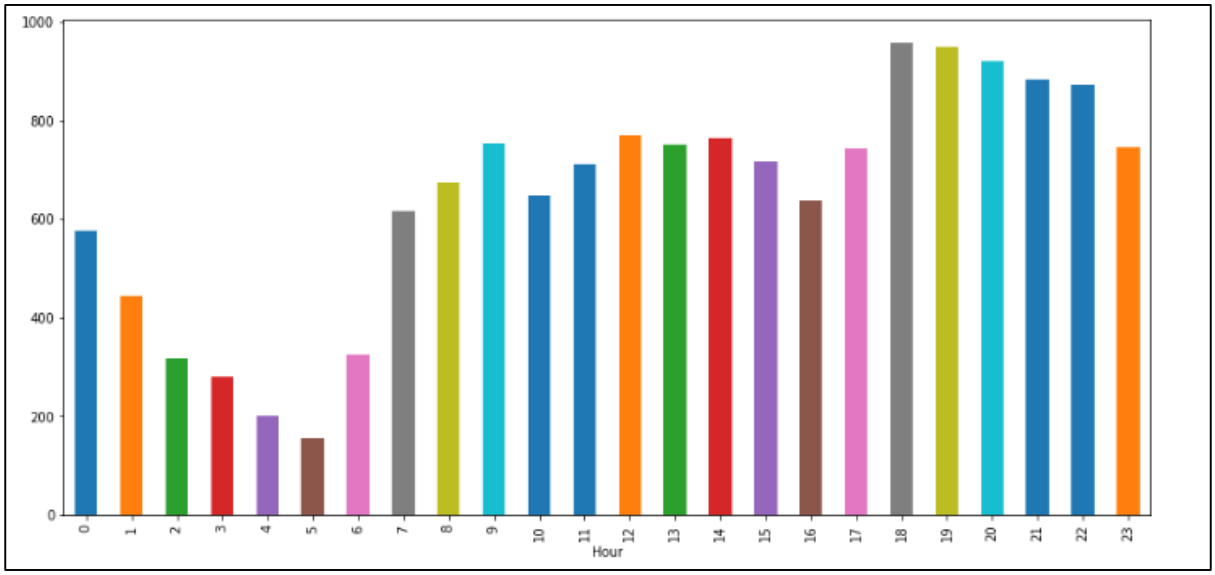


Figure 10- Hours Vs Frequency

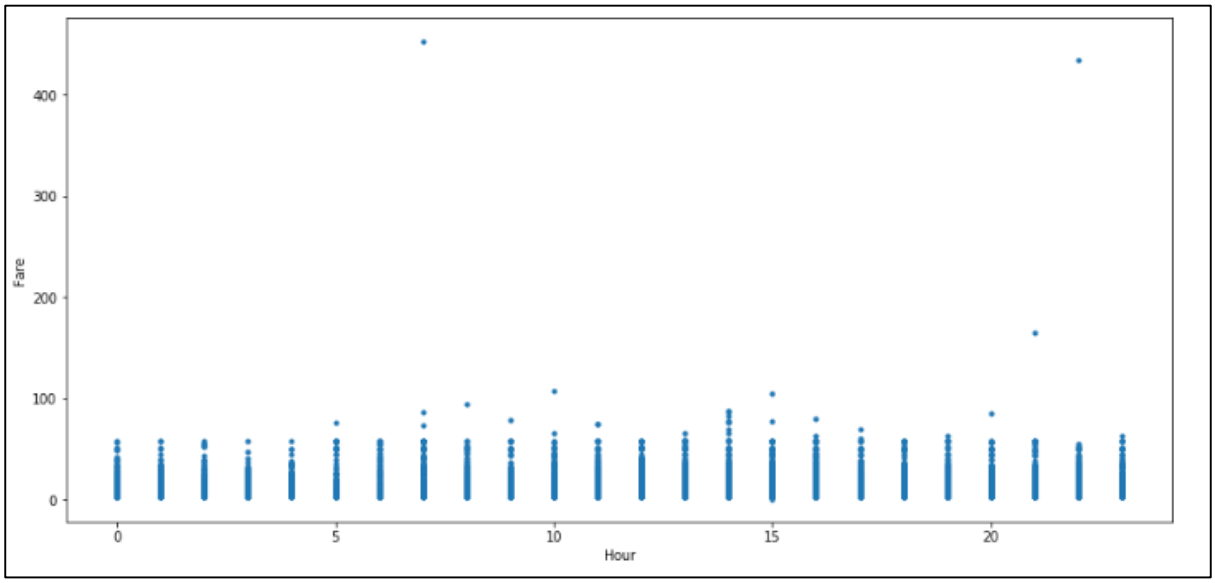


Figure 111- Hours Vs Fare

5.3.4 Week Day VS Fare // Week Day VS Frequency

>Cab fare is high on Friday, Saturday and Monday, may be during weekend and first day of the working day they charge high fares because of high demands of cabs.

>The day of the week does not seem to have much influence on the number of cabs ride

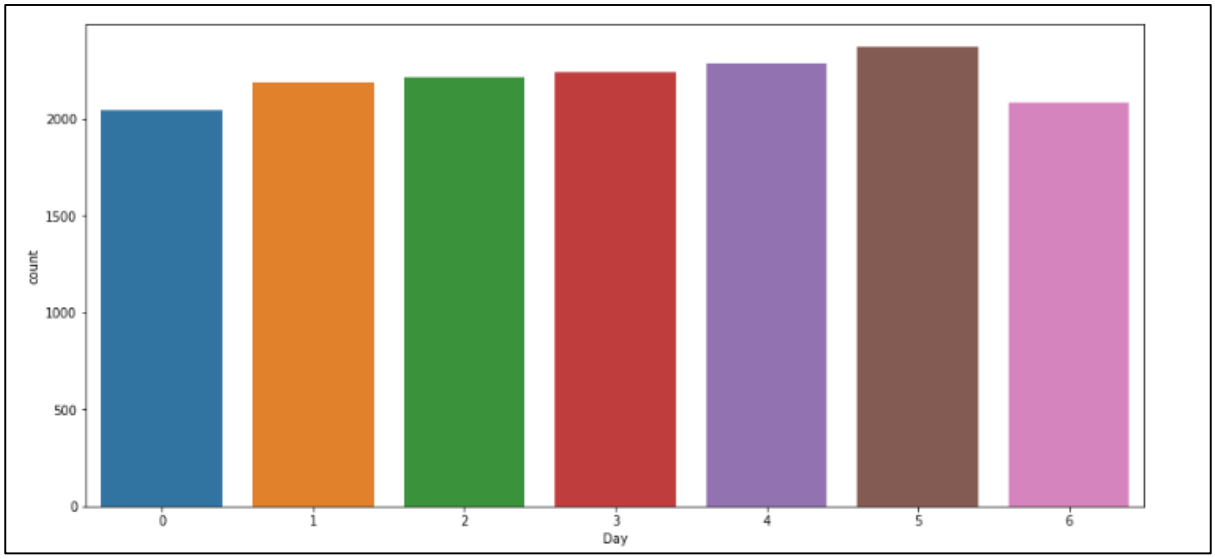


Figure 12-Week Day VS Frequency

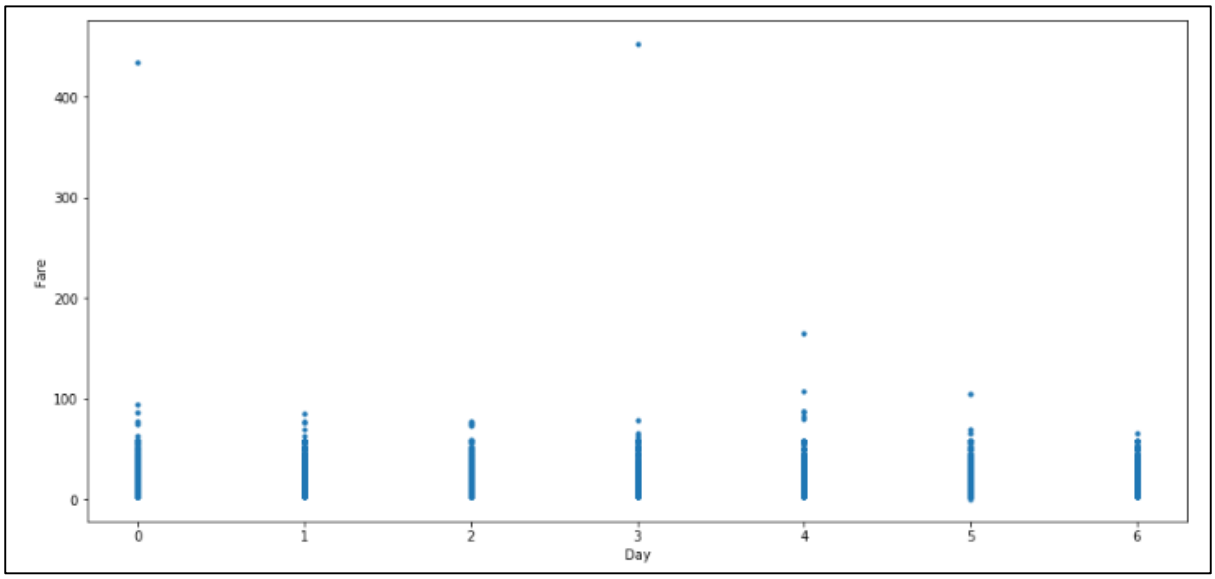
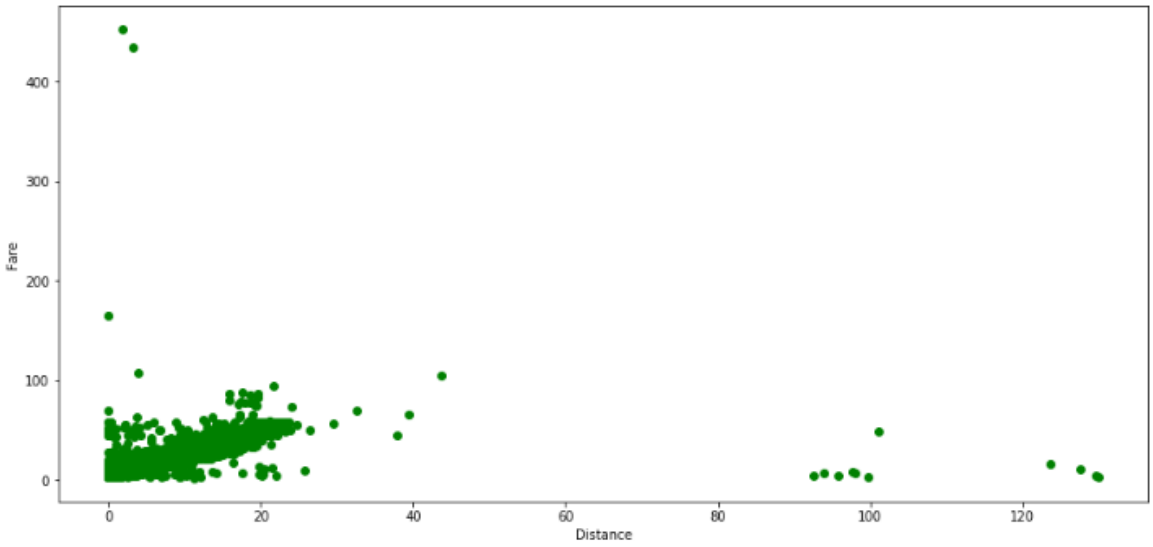


Figure 13- Week Day VS Fare

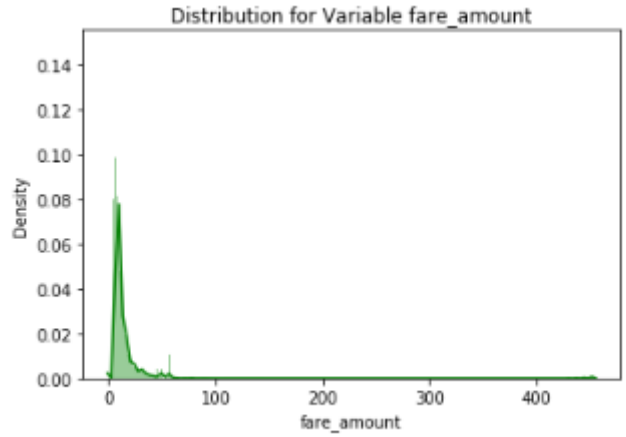
REFERENCES

1. For Data Cleaning and Model Development - https://edwisor.com/career-data-scientist
2. For other code related queries - https://www.analyticsvidhya.com/blog/2016/03/practical-guide-principal-component-analysis- python/
3. For Visualization – https://www.udemy.com/python-for-data-science-and-machine-learning-bootcamp/
4. https://towardsdatascience.com/
5. https://stackoverflow.com/
6. Google searches and Wikipedia pages for Theoretical basics
7. Python and R documentation
8. Online project reference

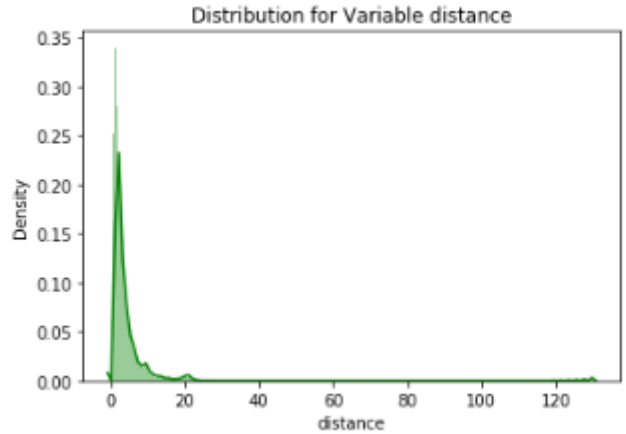
APPENDEX



14 - DISTANCE VS FARE



distance



15 - FARE VS DENSITY AND FARE VS DISTANCE