CAB FARE PREDICTION

JUNE 1

DATA SCIENCE PROJECT 01

Shriram HANGARGEKAR

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem Statement

Problem statement for this project is as below:

> ***"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."***

The historical data mentioned in the problem statement has following attributes:

Table 1-i: Description of historic data

Sr.	Attribute	Type	Description
1	pickup_datetime	timestamp	The time when the cab ride started
2	pickup_longitude	float	The longitude coordinate of where the cab ride started.
3	pickup_latitude	float	The latitude coordinate of where the cab ride started.
4	dropoff_longitude	float	The longitude coordinate of where the cab ride ended.
5	dropoff_latitude	float	The latitude coordinate of where the cab ride ended.
6	passenger_count	integer	The number of passengers in the cab ride.

1.2 Objective

The key objective of this project is to build a model that can predict fare of a trip for a cab rental service. Since the objective is to predict fare, which is continuous numeric output, a regression model is more appropriate to solve our problem. To achieve robust and fair prediction system, multiple models need to be created before finalizing one. Each model is tested and then is evaluated on the basis of different error metric. The report discusses aspects of project from initial data analysis through different data cleaning processes and feature engineering up to developing model for prediction, in addition to these performance parameters selected for evaluation of different model and finalizing on single model is also discussed.

1.3Work Flow

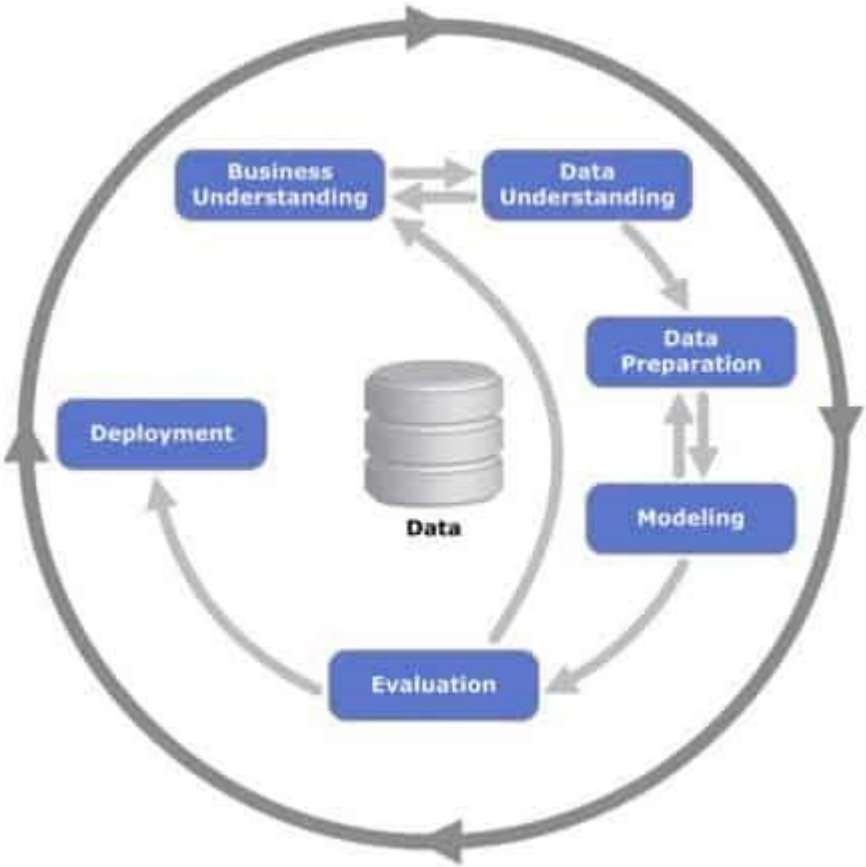


Figure 1-A: CRISP-DM method

Workflow during the project was arranged and prioritized by putting CRISP-DM framework at projects backbone. As the first step in this framework is to understand business, first chapter is aimed at understanding problem statement and its scope. In subsequent chapters, further steps viz. Data Understanding, Data Preparation, Modelling, and Evaluation are discussed. Data understanding covers data overview and basic exploratory analysis. Data preparation focuses on feature selection and data cleaning. After data has been prepared then project proceeds to develop the model, and this step comprises of selection of different algorithm and training those algorithm on train data. In the subsequent chapter, evaluation basis and criteria to select best performing model are discussed. After which final remarks give different observations from developing model.

2 Data Understanding

2.1 Data Overview

Data is divided in two parts viz. train data and test data. Train data is used for developing the model. Because train data is historical data, fair amount is available in data set. This available amount can be used to evaluate performance of different algorithms. There are six attributes available in both data, but in train data additional column has historical fair amount. Train data has 16067 observations and test data has 9914 observations.

There are 6 attributes available to base predictive model on:

- pickup_datetime
  This is a timestamp data, which indicates date and time of pickup by taxi. This data can be useful in explaining fluctuation in cab fare, as date and time can give insight into seasonal fee surge, or peak hour charges. Thus we can derive useful variables from this attribute.

- pickup_longitude, pickup_latitude, dropoff_longitude, and dropoff_latitude
  These attributes individually have little to contribute to cab fare. However by combining these, distance covered during cab ride can be determined. Based on observations from reality, it can be said that distance is one of the major deciding factor for cab fare.

- passenger_count
  This is an integer value. Number of passenger influences cab fare during ride. Travelling in group generally may lead to additional discount or travelling alone may attract additional charges. This too is an important attribute in predicting the cab fare.

- fare_amount
  This variable is present only in train data. This is the target variable. Model is developed based on train data to predict values for fare_amount of test data

Thus, there are six available features to train and develop predictive model. Feature selection is done in next chapter. After developing this basic understanding of data, project can be progressed to next step that is exploratory analysis. Behavior of all available attributes with respect to cab fare values can be studied with help of different graphic charts.

2.2Initial Data Analysis

Target of initial data analysis is to understand underlying structure of given data. Additionally this analysis also helps in detecting outlier points in data and anomalies in values from data. This understanding helps to build the assumption required for model building and exploratory analysis. Thus initial data analysis is an important step in model analysis.

In the initial data exploration, data is analyzed to get feel of data. In this process different statistical parameters such as mean, median, standard deviation, range of the data are observed and inconsistencies such as potential outliers, invalid values present in the data are checked. Furthermore, basic graphs are plotted to see the relation between attributes and target variable.

2.2.1 Data type of attributes

In preliminary analysis, data types of attributes were found to be inappropriate for further procedure, such as fare amount was object type, pickup date-time was also an object. These types were converted to suitable data type. Result of which can be seen in right image. Data type of an attribute needs to be correct so that any operation is possible with a little more ease. Thus data types are adjusted in train and test data.

#	Column	Non-Null Count	Dtype
0	fare_amount	16043 non-null	object
1	pickup_datetime	16067 non-null	object
2	pickup_longitude	16067 non-null	float64
3	pickup_latitude	16067 non-null	float64
4	dropoff_longitude	16067 non-null	float64
5	dropoff_latitude	16067 non-null	float64
6	passenger_count	16012 non-null	float64

Figure 2-A: Data types of train data

#	Column	Non-Null Count	Dtype
0	pickup_datetime	9914 non-null	object
1	pickup_longitude	9914 non-null	float64
2	pickup_latitude	9914 non-null	float64
3	dropoff_longitude	9914 non-null	float64
4	dropoff_latitude	9914 non-null	float64
5	passenger_count	9914 non-null	int64

Figure 2-B: Data types of test data

#	Column	Non-Null Count	Dtype
0	fare_amount	16042 non-null	float64
1	pickup_datetime	16066 non-null	datetime64
2	pickup_longitude	16067 non-null	float64
3	pickup_latitude	16067 non-null	float64
4	dropoff_longitude	16067 non-null	float64
5	dropoff_latitude	16067 non-null	float64
6	passenger_count	16012 non-null	float64

Figure 2-C: Adjusted data types of train data

#	Column	Non-Null Count	Dtype
0	pickup_datetime	9914 non-null	datetime64
1	pickup_longitude	9914 non-null	float64
2	pickup_latitude	9914 non-null	float64
3	dropoff_longitude	9914 non-null	float64
4	dropoff_latitude	9914 non-null	float64
5	passenger_count	9914 non-null	int64

Figure 2-D: Adjusted data types of test data

2.2.2 Statistical parameters

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	16042.000000	16067.000000	16067.000000	16067.000000	16067.000000	16012.000000
mean	15.015004	-72.462787	39.914725	-72.462328	39.897906	2.625070
std	430.460945	10.578384	6.826587	10.575062	6.187087	60.844122
min	-3.000000	-74.438233	-74.006893	-74.429332	-74.006377	0.000000
25%	6.000000	-73.992156	40.734927	-73.991182	40.734651	1.000000
50%	8.500000	-73.981698	40.752603	-73.980172	40.753567	1.000000
75%	12.500000	-73.966838	40.767381	-73.963643	40.768013	2.000000
max	54343.000000	40.766125	401.083332	40.802437	41.366138	5345.000000

Figure 2-E: Description of train data

Basic statistical parameters helps in understanding spread, maxima, minima, and central tendencies of the data. This chart gives simple picture of data. Couple of observations can be made from the above data:

- The number of total observations in train data is 16067. However, attributes fare_amount and passenger_count have fewer count than 16067. This means there are missing values. Missing values are dealt with in next chapter.
- The maximum pickup latitude is 401.08, but latitude of earth is measured within -90 to 90. Thus there are nonsensical entries in this column. Such entries are also need to be removed.
- Similarly, the maximum number of passenger_count is 5345, which is illogical, since a cab cannot carry 5345 passengers in single trip. For this project, the maximum carrying capacity of a cab (such as a SUV) is assumed to be 6. Additionally, the minimum number of passenger is 0, which may not be possible as cab ride must be booked by at least single person.
- The minimum amount of fare paid is -3, this again is unusual data point. The amount paid must be above zero. Thus any values equal to or less than zero must be eliminated.
- Additionally, from problem statement it can be safely inferred that pilot project ran across at most a state, since start-up is not launched at national level yet. After plotting geometric coordinates on map, it is observed that pilot project was launched in vicinity of New York City. And Tri-city area (New York, Rhode Island, and Philadelphia) is spread across longitude -75.5 to -71.8 and latitude 39.5 to 41.9, thus data outside these boundaries can be eliminated under pretext of erroneous entry.

After cleaning up data with these constraints developed with help of the understanding from problem statement and basic descriptive statistical analysis, data is more accurate and fair representation of business intent. And with that data is more suitable for further analysis. Since the boundaries established are based on common sense and business understanding, they can be dropped altogether.

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	15628.000000	15650.000000	15650.000000	15650.000000	15650.000000	15595.000000
mean	15.117504	-73.974837	40.750928	-73.973857	40.751417	1.650356
std	436.121517	0.041507	0.037986	0.039340	0.039648	1.265914
min	1.140000	-74.438233	39.603178	-74.429332	39.604972	1.000000
25%	6.000000	-73.992398	40.736578	-73.991372	40.736321	1.000000
50%	8.500000	-73.982055	40.753345	-73.980567	40.754256	1.000000
75%	12.500000	-73.968108	40.767809	-73.965390	40.768332	2.000000
max	54343.000000	-73.137393	41.366138	-73.137393	41.366138	6.000000

Figure 2-F: Description of corrected train data

After correction in data, number of entries is reduced to 15650 (originally 16067). Thus data is shrunk by approximately 3.5%. This loss of data is justifiable as the lost data was not adhering to practical conditions of business and it would have only distorted result of analysis. There still are missing values present in data. These can be treated during data preparation process.
On quick observation test dataset, no anomalies such as discussed above detected in test data set.

	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	9914	9914.000000	9914.000000	9914.000000	9914.000000	9914.000000
unique	1753	NaN	NaN	NaN	NaN	NaN
top	2011-12-13 22:00:00 UTC	NaN	NaN	NaN	NaN	NaN
freq	270	NaN	NaN	NaN	NaN	NaN
mean	NaN	-73.974722	40.751041	-73.973657	40.751743	1.671273
std	NaN	0.042774	0.033541	0.039072	0.035435	1.278747
min	NaN	-74.252193	40.573143	-74.263242	40.568973	1.000000
25%	NaN	-73.992501	40.736125	-73.991247	40.735254	1.000000
50%	NaN	-73.982326	40.753051	-73.980015	40.754065	1.000000
75%	NaN	-73.968013	40.767113	-73.964059	40.768757	2.000000
max	NaN	-72.986532	41.709555	-72.990963	41.696683	6.000000

Figure 2-G: Description of test data

2.2.3 Graphical Analysis

Graphical representation of data gives simplistic visual and intuitive picture of data. Two types of graphs are used viz. Univariate and Multivariate graphs. Univariate graphs such as box-plot, violin graph, and histograms give overall distribution of an individual variable. Whereas multivariate graphs such as scatter plot, grouped box-plot, and heat-map give overall picture of relationship between two attributes.

- **Univariate Graphs**

Univariate graphs represent distribution on individual attributes. As observed from below graphs, all attributes except passenger_count, which is multi-nodal, are largely concentrated around a single point. Since pilot project was conducted in small area, concentrated distribution of geographical coordinate is justifiable. Passenger count is concentrated at different values [1-6] which are allowable number of passenger in a cab. For the fare_amount attribute though majority values are near zero, extremities reaching up to 55,000 are bit of concern.

Figure 2-H: Violin plot for fare_amount

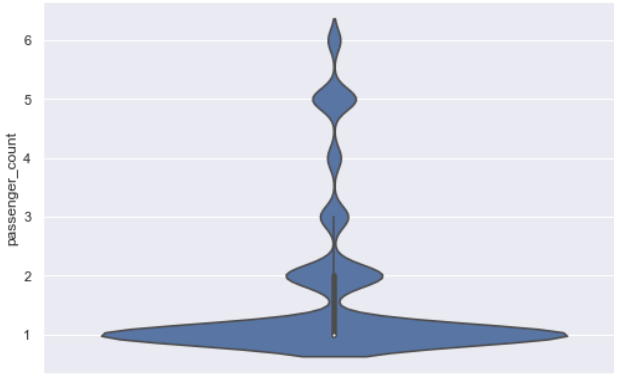


Figure 2-I: Violin plot for passenger_count

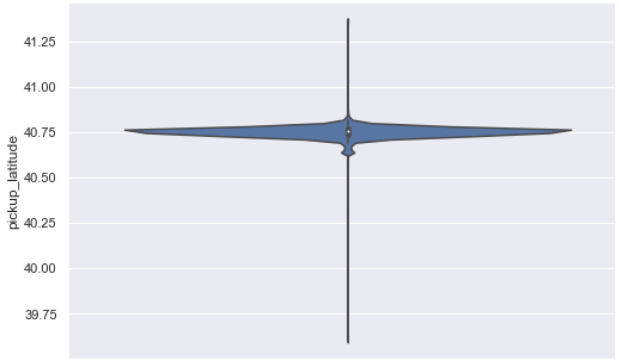


Figure 2-J: Violin plot for pickup_latitude

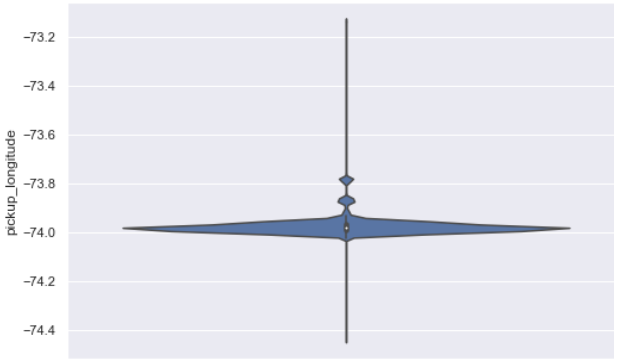


Figure 2-K: Violin plot for pickup_longitude

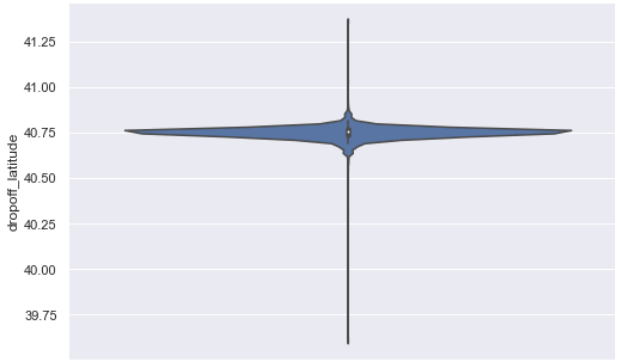


Figure 2-L: Violin plot for dropoff_latitude

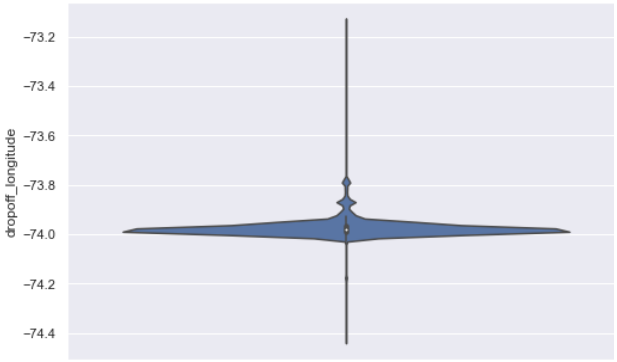


Figure 2-M: Violin plot for dropoff_longitude

❖ Bivariate Graphs

Pairwise graphs can give better picture of relation between a pair of attributes. As observed from scatter plots, line of best fit for all pairs is almost horizontal. This indicates that attributes do not have linear correlation among them. This is good for regression model as it requires input variables to be independent of each other.

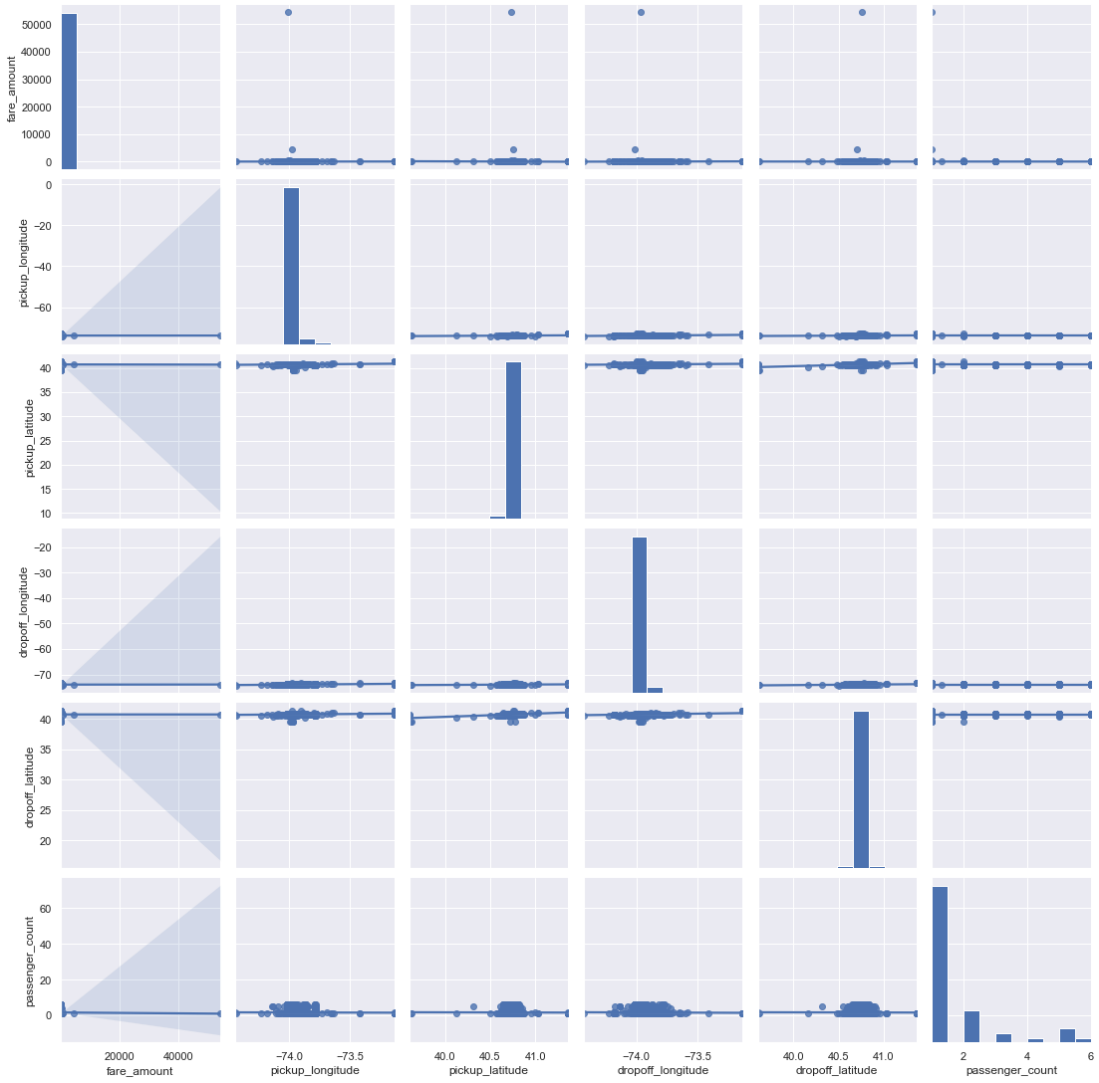


Figure 2-N: Pairwise plot for each attribute

3 Data Preparation

3.1 Feature Engineering

Before proceeding to cleaning the data it is important to understand attributes required for model building. Understanding these requirements helps in effective data preparation. In this process features are transformed on new features are created based on business domain knowledge.
For problem at hand, it is necessary to look at factors contributing to cab fare in real world. In cab services such as Ola Uber fare is total of fee for booking, fare based on travelled distance, charge on travelled time, and occasional surcharges depending upon season or demand. With this in mind a general formula can be approximated as below:

$$\boldsymbol{Fare = I + r_d \times D + r_t \times T + r_p \times P + S_{rush} + S_{long}}$$

Where,

- I : *Initial or basic minimum fare*
- D : *Distance travelled*
- r_d : *Charge per km travelled*
- T : *Time for ride*
- r_t : *Charge per min travelled*
- P : *Passenger travelled*
- r_p : *Charge per passenger*
- S_{rush} : *Surcharge for rush hours, rainy seasons etc.*
- S_{long} : *Surcharge for long rides, outstation trips etc.*

Thus, ignoring fixed charge (intercept) and rate for variable, as those are estimated through ML models, a basic structure of features for model can be represented as below:

$$\boldsymbol{Fare \leftarrow \langle Distance | Time | Passenger | RushHour | LongRide \rangle}$$

If average speed of every cab is assumed to be same, time required for ride is dependent on distance travelled and rush hours. Thus it is safe to drop time variable from the model. Another reason to approximate time attribute is that data set contain only pickup time and not the dropoff time. Thus refined model will look like below:

$$\boldsymbol{Fare \leftarrow \langle Distance | Passenger | RushHour | LongRide \rangle}$$

Now, with required model defined, available attributes need to be transformed into features that most appropriately represent model.

- Distance: Because pickup and dropping points are available. It is easy to calculate distance. For purpose of this project geodesic distance, distance measured on approximated ellipsoid of the size of the earth is measured. Other methods such as spherical are less accurate, and calculating actual street distance from maps through API's is either too costly or too limiting for the project. Hence with geodesic distance a safe approximation can be derived.
- Rush Hours: These are tricky to define. But as pickup time is available, a good inference can be made about rush hours. From pickup date time, useful features that define importance of ours can be extracted. Rush hours are cyclic in nature. Thus features are created on different bases of time-cycles they follow. For analysis following features are derived:
    - Year : As the prices for cab changes each year, to account for the inflation of each year, a new categorical variable 'year' is created. This will contain values from 2009 to 2015.
    - Season : Use of cab changes every season, in rainy season people are more likely to use the cab. To account for seasonal change, categorical variable 'season' is created. This will have values from Winter, Summer, Fall, Spring.
    - WeekDay : Though the rush hours change each day of month. To avoid exhaustive list of each date, rather than each day of month, day will represent whether the day was weekday or weekend.
    - Shift : based on the time of the pickup, the shift, which cab was operating in, can be determined. As this is another indicator of rush hours or late night charges. Time of hours are categorized in Morning, Evening, Night, and Dawn.
- Trip Length: With help of derived distance feature, trips can be categorized as long and short trips. Looking at general trend in the market, trips covering more than 15km are considered as long trips.

With these feature creation, the model is transformed into following structure:

$$\boldsymbol{Fare} \leftarrow \langle \boldsymbol{Distance}|\boldsymbol{Passengers}|\boldsymbol{Year}|\boldsymbol{Season}|\boldsymbol{WeekDay}|\boldsymbol{Shift}|\boldsymbol{TripLength} \rangle$$

In this process several categorical variables have been created. These variables are not suitable for regression models. With one-hot-coding this problem is resolved. Additionally one dummy variable (highlighted in light orange) is dropped to avoid multi-colinearity problem, shaping model in following structure:

<table>
<tr><th>Target</th><th>Input Feature</th><th colspan="2">Derived Features</th><th colspan="7">Dependent Variables</th></tr>
<tr><td rowspan="9">fare_amount</td><td rowspan="4">pickup_datetime</td><td>Year</td><td>Year</td><td>2009</td><td>2010</td><td>2011</td><td>2012</td><td>2013</td><td>2014</td><td>2015</td></tr>
<tr><td>Month</td><td>Season</td><td colspan="2">Winter</td><td colspan="2">Spring</td><td colspan="2">Summer</td><td>Fall</td></tr>
<tr><td>Weekday</td><td>Weekend</td><td colspan="4">Weekday</td><td colspan="3">Weekend</td></tr>
<tr><td>Hour</td><td>Shift</td><td colspan="2">Dawn</td><td colspan="2">Morning</td><td colspan="2">Evening</td><td>Night</td></tr>
<tr><td>pickup_latitude</td><td rowspan="4">Distance</td><td>Distance</td><td colspan="7">Distance</td></tr>
<tr><td>pickup_longitude</td><td rowspan="3">Long Ride</td><td colspan="3" rowspan="2">Short Ride</td><td colspan="4" rowspan="2">Long Ride</td></tr>
<tr><td>dropoff_latitude</td></tr>
<tr><td>dropoff_longitude</td></tr>
<tr><td>passenger_count</td><td>Passengers</td><td>Passengers</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr>
</table>

Figure 3-A: Feature development schematic

3.2Data Cleaning

First, step in data preparation for modelling is to clean data. In this process data is analyzed for missing values and outliers. Missing values harm data analysis as erroneous data might be considered for analysis by program. Outlier values tend to deviate model outcome by either inflating or deflating coefficient during model training.

3.2.1 Missing Value Analysis

Upon analyzing percentage data with missing values following data is obtained:

	Missing Count	Missing Pcent
fare_amount	22	0.140575
pickup_datetime	1	0.006390
pickup_longitude	0	0.000000
pickup_latitude	0	0.000000
dropoff_longitude	0	0.000000
dropoff_latitude	0	0.000000
passenger_count	55	0.351438
geodesic	0	0.000000
trip	0	0.000000
year	1	0.006390
month	1	0.006390
dayofwk	1	0.006390
hour	1	0.006390
season	1	0.006390
weekday	1	0.006390
shift	1	0.006390

Figure 3-B: Missing value analysis

Problem of missing values can be solved in two ways. First to drop the observations altogether and second to impute the value. Imputing value has different methods to do so. The way to deal with missing value is chosen on variable by variable basis.

- fare_amount: Since this is the target variable, imputing values for this variable may lead to erroneous training and validation of the model. Also, missing value percentage is comparatively small, hence dropping these observation is better way to deal with these missing values.
- pickup_datetime: There is only one missing value for this variable. Features derived from this variable too have carried same missing value. Thus dropping this observation is an easier choice to deal with the missing value.
- passenger_count: This variable has 55 missing values. Though missing values are very few (~0.35%). To avoid further loss of data, values for this variable are imputed.

3.2.2 Outlier Analysis

As observed with help of violin plots during graphical analysis there are large number of outliers in data. Outliers occur in data either because of entry error or because of special case. In given data it is possible that some of routine trips such as a hotel to airport may have fixed fare, which is not calculated by usual formula. At this stage model is incapable of handling such cases. These type of observations complicate the model unnecessary. But dropping these observations may lead to loss of data, which is not desirable. Thus outliers are replaced with NaN and are imputed along with missing values detected earlier.

After Replacing outliers with missing values, below are total missing values in Data:

```
fare_amount        1328
geodesic           1335
year                  0
month                 0
hour                  0
dayofwk               0
passenger_count      55
dtype: int64
```

Figure 3-C: Total missing values in data

3.2.3 Imputation Method

Imputation is nothing but using statistics to determine missing values. For imputation a subset of train data is created with columns: fare_amount, geodesic, year, month, day_of_week, hour, and passenger_count. For the purpose of imputation these variables are assumed to be numeric. There are two major methods for imputation: first using central tendency and second using algorithms such as KNN imputation.

Before starting with imputation, a sample is taken from dataset and random values are set to NaN. This will help in determining performance of imputation method.

- **Central Tendency**

Since passenger count is only feature with missing value and is categorical. Mode can be used to impute value, but the mode will increase bias in the data. Another tendency can be used is mean. The mean is then rounded off for passenger count.

Although, using mean to impute is simple approach, it has potential to include discrepancies in data, such as to assume every fare amount that is missing to be 19 has its own disadvantages.

Table 3-i: Imputation values from central tendency (mean)

Mean Imputation	fare_amount	geodesic	Passenger_count
Mean	8.949	2.479	~2 (1.653)

❖ **KNN Imputation**

KNN Imputation calculates missing values based on nearest neighbors. Algorithm calculates distance between point to impute and points in cluster. And fills value with average of the cluster. This approach is more accurate approximation than using mean of whole data. But before choosing KNN imputation method, trials were performed on sample data. Random values in a sample were set to NaN and imputed with KNNImputer.

Table 3-ii: Imputation values from KNN Imputation

Sr No	passenger_count		Fare_amount		geodesic	
	Original	Imputed	Original	Imputed	Original	Imputed
1	2	1	4.5	7.80	1.0	2.13
2	1	2	17.0	8.62	4.8	2.13
3	1	2	12.0	6.62	3.6	1.13
4	1	3	9.5	7.62	2.9	1.51
5	1	2	4.5	9.22	1.1	3.01

As deviations from original values are observed in imputed values, to validate whether deviations are within acceptable limit standard deviation of original and imputed values is checked. If standard deviation differs significantly then it is not okay to use KNN Imputation.

Table 3-iii: Std dev for Imputed and Original values

Std Dev	passenger_count	Fare_amount	geodesic
Original	1.390	4.137	1.519
Imputed	1.387	4.082	1.506

However standard deviations are pretty close to original values. Thus all missing values are imputed with KNN Imputation.

3.3 Feature Selection

In this stage, all features are analyzed to check their importance and inter-correlation with other features. Correlation, VIF are some of the indicators of this.

3.3.1 Correlation Analysis

If the two or more features are highly correlated then they carry same information in data. That is redundant for model building and potentially inflating for result. Thus one of the highly correlated features should be kept in model. Heatmap is plotted to analyze correlation.

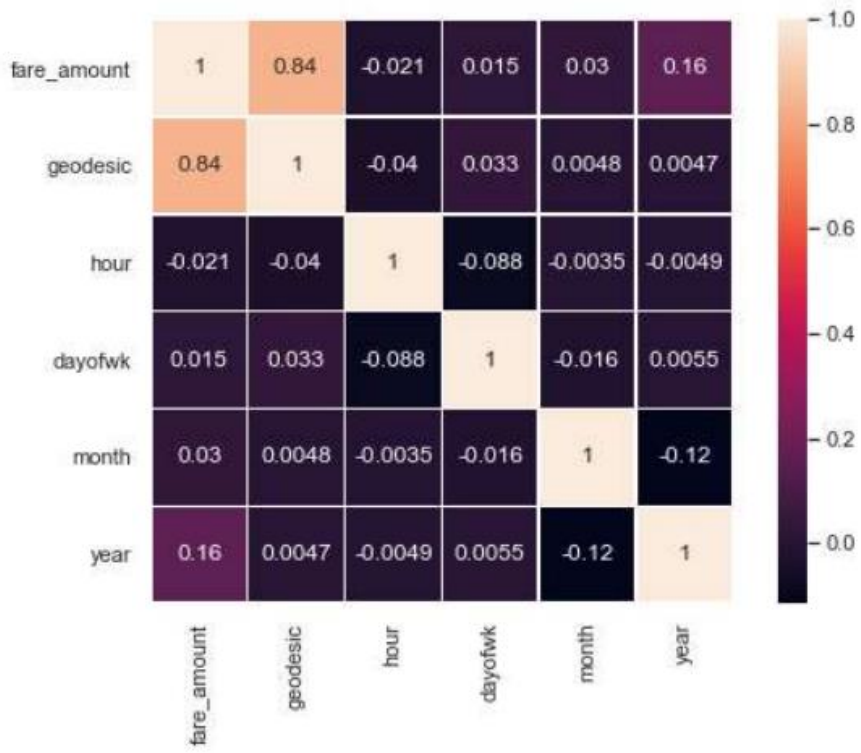


Figure 3-D: Heatmap of correlation of variables

As observed from heatmap, only high correlation observed is between geodesic and fare_amount. Since, fare_amount is target variable and geodesic is dependent variable, this correlation is acceptable. Apart from this correlation there is no other correlation in data. For detail understanding scatterplot is created for geodesic and fare_amount.

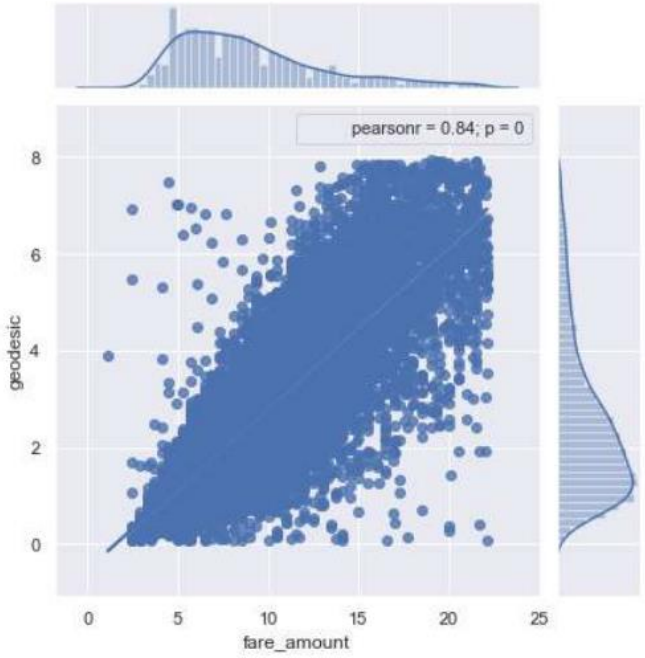


Figure 3-E: Scatterplot between fare_amount and geodesic

3.3.2 Multi co-linearity Analysis

Multi co-linearity is checked as there is possibility that combination of multiple variables may have strong co-linearity which is as harmful as the correlation between two variables. This multi co-linearity is checked with help of variance inflation factor (VIF).

- If the VIF is equal to 1 then it means there is no co-linearity in variable. This is ideal result.
- If the VIF is between 1 and 5 then it indicates moderate co-linearity. This is acceptable condition.
- If the VIF is more than 5 then it indicates high co-linearity. This is not desirable outcome.
- However if more than one variable have VIF greater than 5 then only one with highest VIF is dropped.

	VIF	INPUTS
0	18.454017	Intercept
1	1.021123	geodesic
2	1.040561	passenger_count_2
3	1.019089	passenger_count_3
4	1.011653	passenger_count_4
5	1.025704	passenger_count_5
6	1.017457	passenger_count_6
7	1.687326	year_2010
8	1.687821	year_2011
9	1.709854	year_2012
10	1.709451	year_2013
11	1.666266	year_2014
12	1.409693	year_2015
13	1.640270	season_spring
14	1.549580	season_summer
15	1.586195	season_winter
16	1.015913	weekday_weekend
17	2.179955	shift_evening
18	2.123692	shift_morning
19	2.055446	shift_night
20	1.002973	trip_long

Figure 3-F: VIF for dependent variables

Since there is no co-linearity and discrepancy in data, data can be used for model building.

4 Modeling

4.1 Pre-modeling Steps

Before building model, some steps are taken that help in building model. These steps include splitting data into train and test, understanding influence of hyper-parameter on model and defining evaluation metrics for data. These preparatory steps ease process of modeling and achieve better results.

4.1.1 Train and Validation data split

Before building model, train data is split into train set for training purpose and test set for validation purpose. This helps in evaluating performance of model. Since it is impossible to know true outcome of test_data, pseudo test set is created to validate performance of model.

- Whole data is divided in dependent variables (X) and target variable (Y)
- 30% of data is in test set and it has 4688 observations
- 70% of data in in train data set and it has 10939 observations
- X_test, Y_test are validation data set
- X_train, Y_train are training data set

4.1.2 Hyper-parameter optimization

Performance of model depends on the value of different parameters attributed to it. These parameters controls behavior of model. To get optimum performance out of model tuning of these parameters is necessary. Two methods used for hyper parameter optimization:

- GridSearchCV:
  In GridSearchCV, the algorithm creates a grid of all combinations of values of hyper-parameters within provided range. And every set of hyper-parameters in grid are used to train model and to score the test data. Since every possible combination is tried, this method at times can be inefficient, but always provides the best results.
- RandomizedSearchCV:
  In RandomizedSearchCV, the algorithm as name suggests tests random combinations of hyper-parameters within provided range. And several iterations are performed to get good results. This means that not every combination is tested. Thus algorithm compromises on the accuracy of the results but provides better results in less time compared to grid search algorithm.

4.1.3 Evaluation Metrics

There are multiple metrics such as mean squared error (MSE), root mean squared error (RMSE), mean absolute percentage error (MAPE), coefficient of determination (R^2), root mean squared log error (RMSLE), Adjusted R^2. Each metric is defined differently hence is interpreted differently. For the purpose of this project, three metrics are used:

- MAPE : Mean Absolute Percentage Error is measure of accuracy of regression model. MAPE is calculated by taking average of absolute percentage deviation of predicted values with respect to actual values. This metric is used as it gives relative deviation of predicted values. It is calculated with following formula:

$$\boldsymbol{MAPE} = \frac{\boldsymbol{100}}{\boldsymbol{n}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \left| \frac{\boldsymbol{Y_{true} - Y_{pred}}}{\boldsymbol{Y_{true}}} \right|$$

- RMSE : Root Mean Squared Error is measure of average error of regression model. RMSE is calculated by taking average of square of error of prediction. Squaring errors make them positive and avoid cancelling positive and negative errors. This is absolute measure of deviation and is calculated with following formula:

$$\boldsymbol{RMSE} = \sqrt{\frac{\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} (\boldsymbol{Y_{true} - Y_{pred}})^{\boldsymbol{2}}}{\boldsymbol{n}}}$$

- R^2 : Coefficient of Determination (R^2) is measure of the proportion of variation in predictions explained by all of independent variables in model. This is calculated by subtracting ratio of sum of square of residual to total sum of squares from one. This measures how close predictions and is calculated with following formula:

$$\boldsymbol{R^2} = \boldsymbol{1} - \frac{\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} (\boldsymbol{Y_{true} - Y_{pred}})^{\boldsymbol{2}}}{\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} (\boldsymbol{Y_{true} - \bar{Y}})^{\boldsymbol{2}}}$$

Y_{true} : Actual value of target variable
Y_{pred} : Predicted value of target variable
$\bar{Y}$: $Mean\ of\ target\ variable$
n : $Number\ of\ observations$

4.2Model Building

Since objective is to predict target variable 'fare_amount', regression models need to be used to achieve objective. As there are several regression algorithms available, before finalizing one, different algorithms are tried:

- Linear Regression
- Ridge Regression
- Lasso Regression
- Decision Tree Regression
- Random Forest Regression
- XGBoost Regression

4.2.1 Linear Regression

Linear regression builds a model between target variable and features by assuming linear relationship between them. Observed data is fitted to linear equation by assigning weight to each feature. This is most basic and common model used for prediction problem.

- **Hyper-parameter Optimization**

With GridSearchCV following parameters are obtained:

```
Optimum Parameters: {'copy_X': True, 'fit_intercept': True, 'normalize': True}
Optimum score     : 0.7314191269457941
```

Figure 4-A: Linear regression optimum hyper-parameter

- **Model Building**

Following are the coefficients of linear regression model:

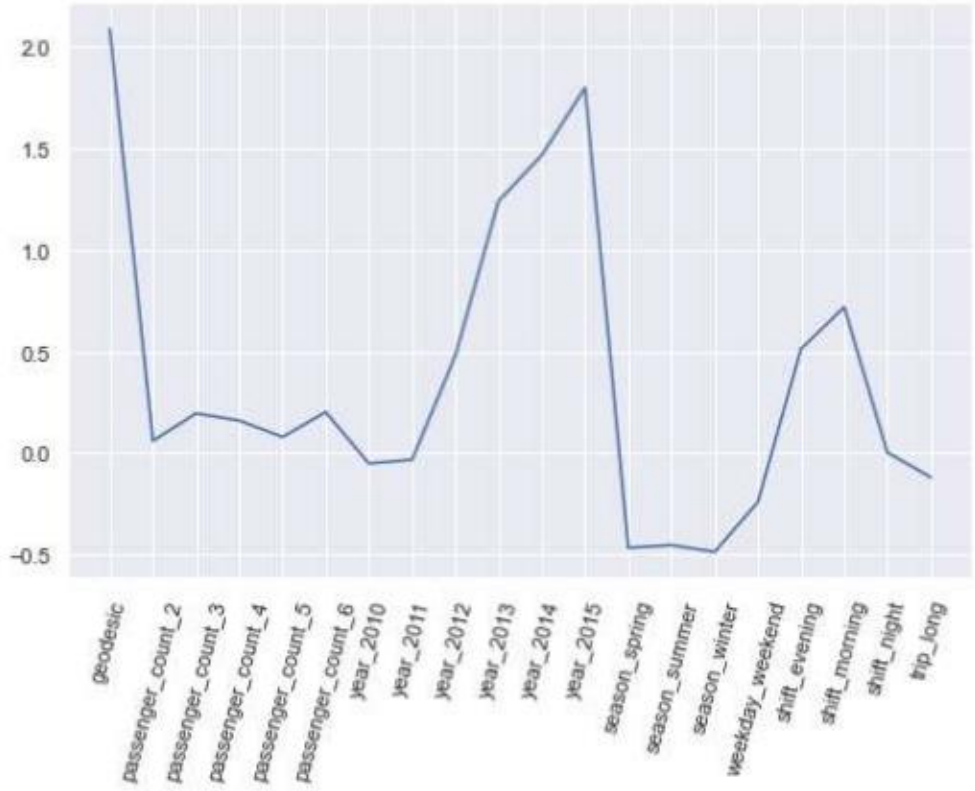


Figure 4-B: Line Plot for Coefficients of Linear regression

- **Model Performance**

Following are scores of train and test data sets:

Table 4-i : Performance of Linear Regression

	MAPE	R2	RMSE
train	17.4482	0.7278	2.1124
test	17.8761	0.7425	2.1122

4.2.2 Ridge Regression

Ridge regression adds a penalty term in cost function to determine coefficients. This model uses L2 regularization technique. It shrinks the coefficient and thus prevents multi-colinearity and reduces model complexity.

❖ Hyper-parameter Optimization

With GridSearchCV following parameters are obtained:

```
Optimum Parameters: {'alpha': 1.0, 'max_iter': 100, 'normalize': False}
Optimum score     : 0.7314196704388056
```

Figure 4-C: Ridge regression optimum hyper-parameter

❖ Model Building

Following are the coefficients of ridge regression model:

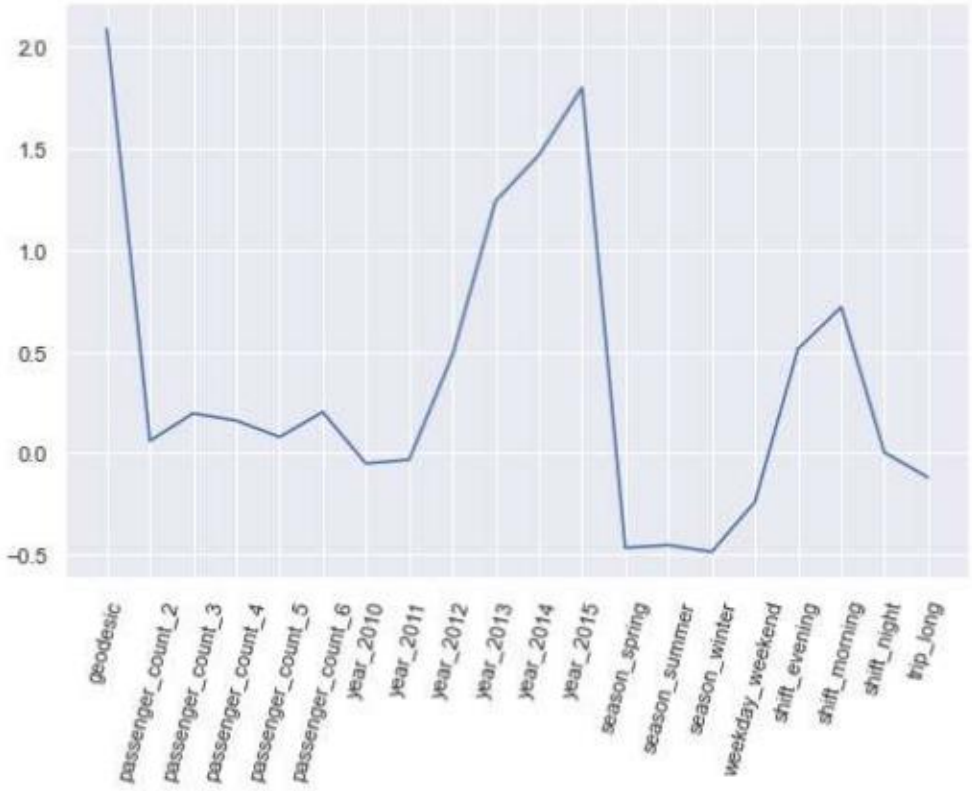


Figure 4-D: Line Plot for Coefficients of Ridge regression

❖ Model Performance

Following are scores of train and test data sets:

Table 4-ii: Performance of Ridge Regression

	MAPE	R2	RMSE
train	17.4479	0.7278	2.1125
test	17.8754	0.7225	2.1122

4.2.3 LASSO Regression

LASSO means Least Absolute Shrinkage and Selection Operator. This model uses L1 regularization technique. It shrinks the coefficient and also automatically selects feature model. And thus reduces multi-colinearity in data.

❖ Hyper-parameter Optimization

With GridSearchCV following parameters are obtained:

```
Optimum Parameters: {'alpha': 0.0004452958509942655, 'max_iter': 100, 'normalize': False}
Optimum score     : 0.7314249966863244
```

Figure 4-E: LASSO regression optimum hyper-parameter

❖ Model Building

Following are the coefficients of LASSO regression model:

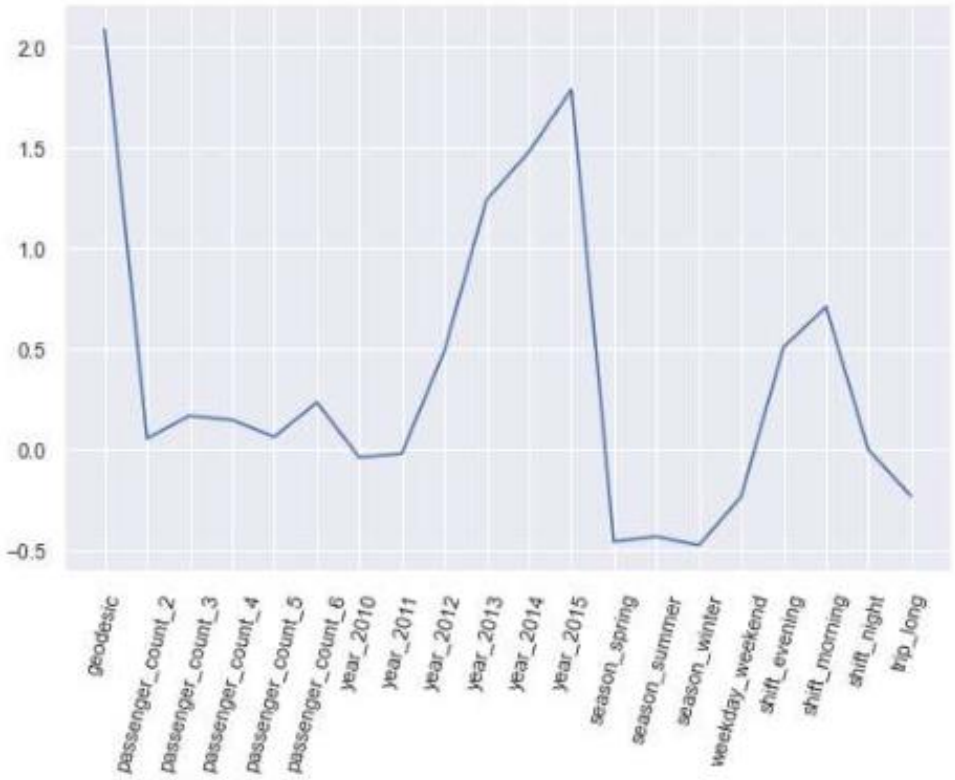


Figure 4-F: Line Plot for Coefficients of LASSO regression

❖ Model Performance

Following are scores of train and test data sets:

Table 4-iii: Performance of LASSO Regression

	MAPE	R2	RMSE
train	17.4453	0.7278	2.1125
test	17.8746	0.7425	2.1123

4.2.4 Decision Tree Regression

Decision tree builds regression models by breaking down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed. The final result is a tree with decision nodes and leaf nodes.

❖ Hyper-parameter Optimization

With RandomizedSearchCV following parameters are obtained:

```
Optimum Parameters: {'min_samples_split': 4, 'max_depth': 6, 'criterion': 'mse'}
Optimum score     : 0.7125962743074015
```

Figure 4-G: Decision tree regression optimum hyper-parameter

❖ Model Building

Following are the feature importance of decision tree regression model:

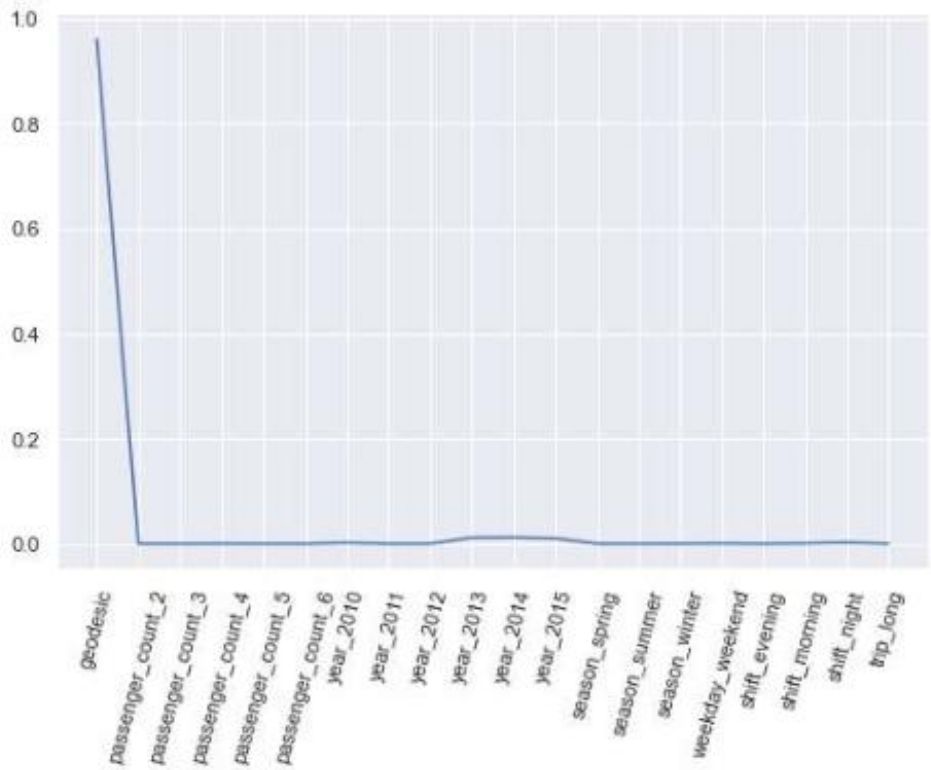


Figure 4-H: Line Plot for feature importance of Decision tree regression

❖ Model Performance

Following are scores of train and test data sets:

Table 4-iv: Performance of Decision Tree Regression

	MAPE	R2	RMSE
train	17.5361	0.7295	2.1059
test	18.4744	0.7275	2.1731

4.2.5 Random Forest Regression

Random Forest Regression is a supervised learning algorithm that uses ensemble method for regression. Ensemble method is a technique that combines predictions from multiple machine learning algorithms to make a more accurate prediction than a single model.

❖ Hyper-parameter Optimization

With RandomizedSearchCV following parameters are obtained:

```
Optimum Parameters: {'n_estimators': 200, 'min_samples_split': 3, 'min_samples_leaf': 4, 'max_features': 'auto', 'max_de
pth': 14, 'bootstrap': True}
Optimum score     : 0.7257660829011023
```

Figure 4-I: Random forest regression optimum hyper-parameter

❖ Model Building

Following are the coefficients of Random forest regression model:

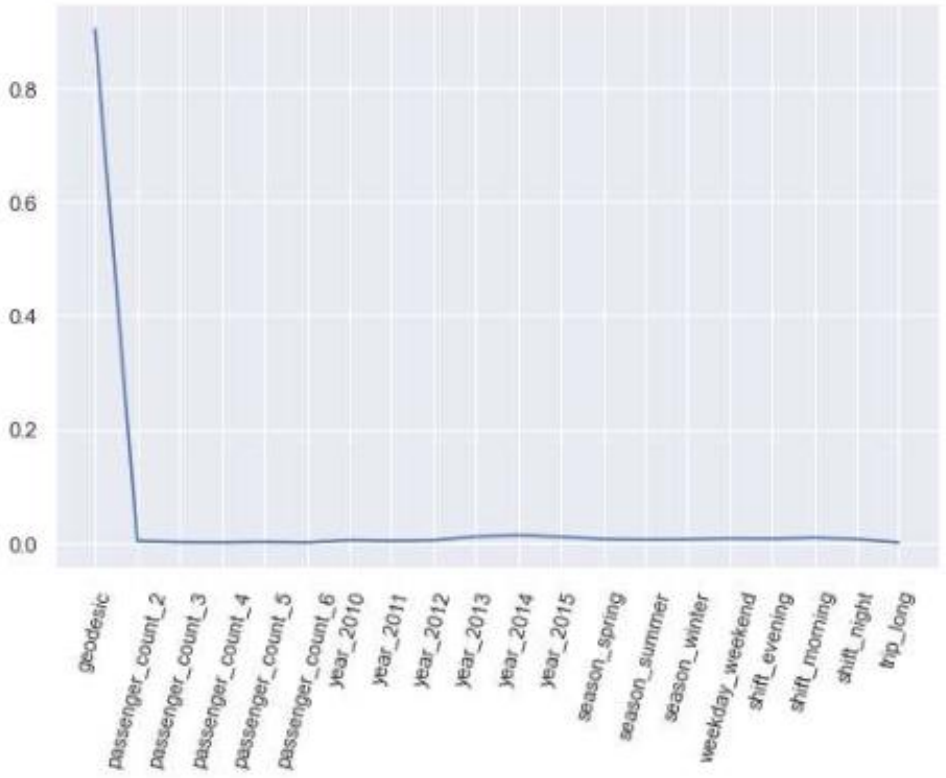


Figure 4-J: Line Plot for feature importance of Random forest regression

❖ Model Performance

Following are scores of train and test data sets:

Table 4-v: Performance of Random Forest Regression

	MAPE	R2	RMSE
train	13.7593	0.8307	1.6663
test	17.9317	0.7424	2.1127

4.2.6 XGBoost Regression

XGBoost is ensemble technique and expects to have the base learners which are uniformly bad at the remainder so that when all the predictions are combined, bad predictions cancels out and better one sums up to form final good predictions.

❖ Hyper-parameter Optimization

With RandomizedSearchCV following parameters are obtained:

```
Optimum Parameters: {'reg_alpha': 0.11721022975334805, 'n_estimators': 100, 'learning_rate': 0.1, 'colsample_bytree': 0.
9000000000000001}
Optimum score     : 0.7384874075837606
```

Figure 4-K: XGBoost regression optimum hyper-parameter

❖ Model Building

Following are the coefficients of Random forest regression model:

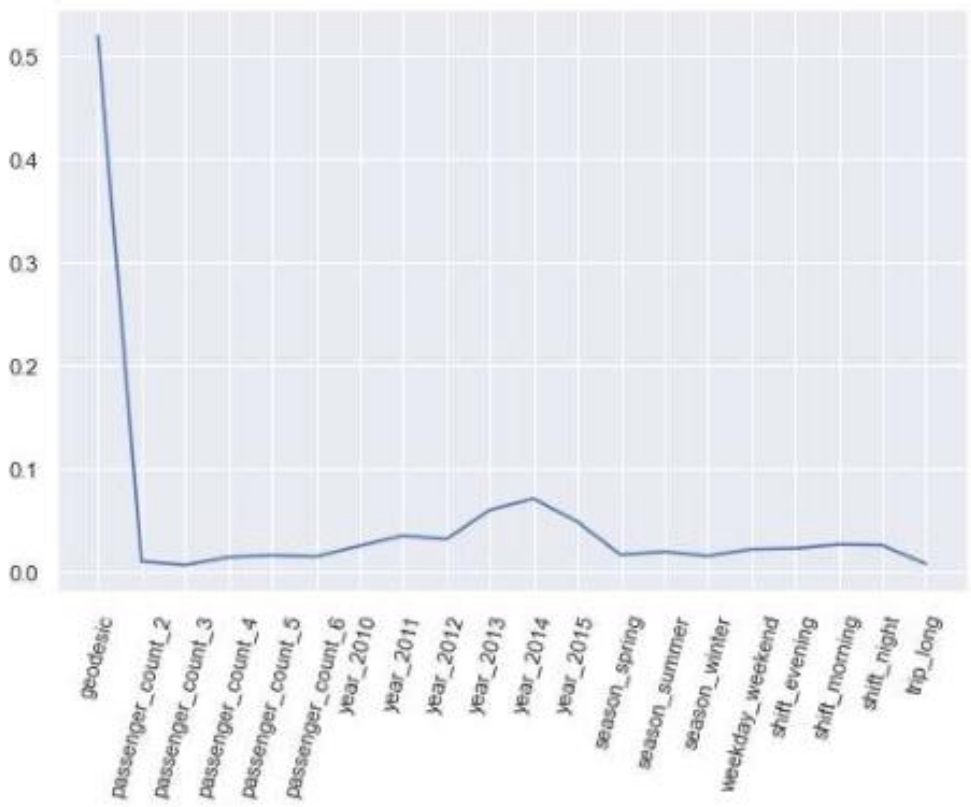


Figure 4-L: Line Plot for feature importance of XGBoost regression

❖ Model Performance

Following are scores of train and test data sets:

Table 4-vi: Performance of XGBoost Regression

	MAPE	R2	RMSE
train	15.8738	0.7759	1.9168
test	17.4035	0.7527	2.0702

4.3 Model Selection

Based on evaluation metrics, all models performed equally well. But XGBoost regression and Random forest regression have better performance than the rest. When compared with XGBoost, Random forest performs better on train data but has slightly bad performance on test data. Whereas XGBoost performs comparatively well on train and test data.

Better results on train data for random forest may be attributed to over-fitting of the data. Thus creating big imbalance in train and test performance. On the other hand XGBoost has balanced performance on train and test data.

Thus owing to balanced performance and low risk over-fitting, XGBoost is better performer for regression model. To predict on final values, XGBoost regression is chosen. Output is saved in csv file with respect to index of input data.

5 Conclusion

Though the accuracy model is acceptable, it can be improved with better and complex programming. The model including following consideration will have better accuracy:

- A model that can account for long trips with fixed price will improve prediction accuracy greatly. Such trips include airport to airport rides or from bus/rail station to famous touring spot. These trips generally are not calculated but have fixed price. Identifying such situation will boost accuracy of model.
- A model that can estimate special charges such as at the toll-booth will also have positive impact on accuracy of model. Based on pickup and dropoff location it can be estimated whether there is toll booth in the route.
- Additional data of dropoff datetime can also help in improving accuracy of model. As it will give clear picture of traffic and total duration of time. This will help I determining whether traffic was heavy to incur additional charges.
- Another challenge to deal with is round trip problem. Since pickup and dropoff locations are same distance travelled during ride will calculated as zero. A solution to this is recording geographic coordination of intermediate. This will help interfere distance traveled based on time and intermediate distance.

And obviously analyzing data as it is generated will also help for betterment of model.

6 Appendix A

PYTHON CODE

Importing libraries and setting up directory

Libraries used:

1. os: to handle directory structure
2. panda: to handle dataframes
3. munpy and scipy: for numeric and scintific calculation methods
4. matplotlib and seaborn: to plot and graphic visualization of data
5. sklearn: for various machine learning models such as KNN imputation, Parameter Search CV, Regression models
6. patsy: to define structure of model
7. xgboost: to improve accuracy through ensemble technique
8. joblib: to save model

In [1]:
```
import os
import pandas as pd
import numpy as np
import scipy as sp
from geopy.distance import geodesic, distance
import matplotlib as mpl
import seaborn as sns
from sklearn.impute import KNNImputer
from patsy import dmatrices
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.model_selection import train_test_split
from sklearn import metrics as mtr
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.linear_model import LinearRegression,Ridge,Lasso
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
import xgboost as xgb
import joblib

os.chdir('K:\\Data_Science\\Project_01\\Trials')
```

Loading and getting an overview of train and test data

Train and test csv files saved in different dataframes[dfTrain, dfTest].

In [2]:
```
dfTrain =  pd.read_csv('train_cab.csv')
dfTest = pd.read_csv('test.csv')
dfData = [dfTrain, dfTest]

for data in dfData:
    data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16067 entries, 0 to 16066
Data columns (total 7 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   fare_amount        16043 non-null  object 
 1   pickup_datetime    16067 non-null  object 
 2   pickup_longitude   16067 non-null  float64
 3   pickup_latitude    16067 non-null  float64
 4   dropoff_longitude  16067 non-null  float64
 5   dropoff_latitude   16067 non-null  float64
 6   passenger_count    16012 non-null  float64
dtypes: float64(5), object(2)
memory usage: 878.8+ KB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9914 entries, 0 to 9913
Data columns (total 6 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   pickup_datetime    9914 non-null   object 
 1   pickup_longitude   9914 non-null   float64
 2   pickup_latitude    9914 non-null   float64
 3   dropoff_longitude  9914 non-null   float64
 4   dropoff_latitude   9914 non-null   float64
 5   passenger_count    9914 non-null   int64  
dtypes: float64(4), int64(1), object(1)
memory usage: 464.8+ KB
```

In [3]:
```
dfTrain.describe(include = 'all')
```

Out[3]:

	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	16043	16067	16067.000000	16067.000000	16067.000000	16067.000000	16012.000000
unique	468	16021	NaN	NaN	NaN	NaN	NaN
top	6.5	2012-12-06 18:05:00 UTC	NaN	NaN	NaN	NaN	NaN
freq	759	2	NaN	NaN	NaN	NaN	NaN
mean	NaN	NaN	-72.462787	39.914725	-72.462328	39.897906	2.625070
std	NaN	NaN	10.578384	6.826587	10.575062	6.187087	60.844122
min	NaN	NaN	-74.438233	-74.006893	-74.429332	-74.006377	0.000000
25%	NaN	NaN	-73.992156	40.734927	-73.991182	40.734651	1.000000
50%	NaN	NaN	-73.981698	40.752603	-73.980172	40.753567	1.000000
75%	NaN	NaN	-73.966838	40.767381	-73.963643	40.768013	2.000000
max	NaN	NaN	40.766125	401.083332	40.802437	41.366138	5345.000000

In [4]:
```
dfTest.describe(include = 'all')
```

Out[4]:

	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	9914	9914.000000	9914.000000	9914.000000	9914.000000	9914.000000
unique	1753	NaN	NaN	NaN	NaN	NaN
top	2011-12-13 22:00:00 UTC	NaN	NaN	NaN	NaN	NaN
freq	270	NaN	NaN	NaN	NaN	NaN
mean	NaN	-73.974722	40.751041	-73.973657	40.751743	1.671273
std	NaN	0.042774	0.033541	0.039072	0.035435	1.278747
min	NaN	-74.252193	40.573143	-74.263242	40.568973	1.000000
25%	NaN	-73.992501	40.736125	-73.991247	40.735254	1.000000
50%	NaN	-73.982326	40.753051	-73.980015	40.754065	1.000000
75%	NaN	-73.968013	40.767113	-73.964059	40.768757	2.000000
max	NaN	-72.986532	41.709555	-72.990963	41.696683	6.000000

Correcting data types of data frames

- pickup_datetime saved as "datetime" type
- fare amount saved as "numeric" type

In [5]:
```
dfTrain['fare_amount'] = pd.to_numeric(dfTrain['fare_amount'], errors = 'coerce')

for data in dfData:
    data['pickup_datetime'] = pd.to_datetime(data['pickup_datetime'], errors = 'coerce')
    data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16067 entries, 0 to 16066
Data columns (total 7 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   fare_amount        16042 non-null  float64
 1   pickup_datetime    16066 non-null  datetime64[ns, UTC]
 2   pickup_longitude   16067 non-null  float64
 3   pickup_latitude    16067 non-null  float64
 4   dropoff_longitude  16067 non-null  float64
 5   dropoff_latitude   16067 non-null  float64
 6   passenger_count    16012 non-null  float64
dtypes: datetime64[ns, UTC](1), float64(6)
memory usage: 878.8 KB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9914 entries, 0 to 9913
Data columns (total 6 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   pickup_datetime    9914 non-null   datetime64[ns, UTC]
 1   pickup_longitude   9914 non-null   float64
 2   pickup_latitude    9914 non-null   float64
 3   dropoff_longitude  9914 non-null   float64
 4   dropoff_latitude   9914 non-null   float64
 5   passenger_count    9914 non-null   int64
dtypes: datetime64[ns, UTC](1), float64(4), int64(1)
memory usage: 464.8 KB
```

Removing nonsensical values from train data set

removed observations containing nonsensical values:

1. unusual fare amount such as non-positive
2. unreal passenger count such as 0 or more than 6
3. and extreme geolocations dropped.

In [6]:
```
#removing data with NOK fare amount
dfTrain = dfTrain[~(dfTrain['fare_amount']<1)]

#removing data with NOK passenger count
dfTrain = dfTrain[~((dfTrain['passenger_count']<1) | (dfTrain['passenger_count']>6))]

#removing data with NOK pickup longitude
dfTrain = dfTrain[~((dfTrain['pickup_longitude']<-75.5) | (dfTrain['pickup_longitude']>-71.8))]

#removing data with NOK pickup lattitude
dfTrain = dfTrain[~((dfTrain['pickup_latitude']<39.5) | (dfTrain['pickup_latitude']>41.9))]

#removing data with NOK dropoff longitude
dfTrain = dfTrain[~((dfTrain['dropoff_longitude']<-75.5) | (dfTrain['dropoff_longitude']>-71.8))]

#removing data with NOK dropoff lattitude
dfTrain = dfTrain[~((dfTrain['dropoff_latitude']<39.5) | (dfTrain['dropoff_latitude']>41.9))]

dfTrain.describe()
```

Out[6]:

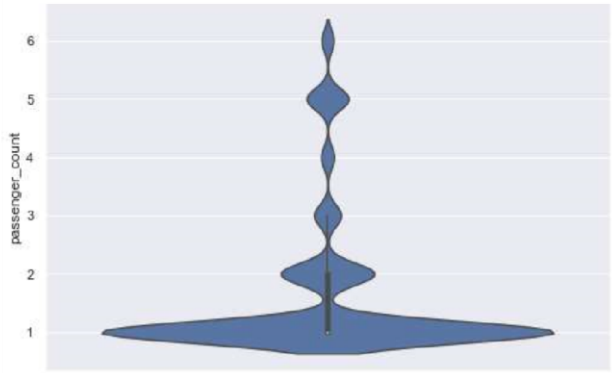
	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	15628.000000	15650.000000	15650.000000	15650.000000	15650.000000	15595.000000
mean	15.117504	-73.974837	40.750928	-73.973857	40.751417	1.650356
std	436.121517	0.041507	0.037986	0.039340	0.039648	1.265914
min	1.140000	-74.438233	39.603178	-74.429332	39.604972	1.000000
25%	6.000000	-73.992398	40.736578	-73.991372	40.736321	1.000000
50%	8.500000	-73.982055	40.753345	-73.980567	40.754256	1.000000
75%	12.500000	-73.968108	40.767809	-73.965390	40.768332	2.000000
max	54343.000000	-73.137393	41.366138	-73.137393	41.366138	6.000000

Visualizing distribution of the data in train dataset

- violin plot used for distribution of individual variables
- pairwise plot used for correlation of each pair of variables

In [7]:
```
sns.set(rc={'figure.figsize':(8.4,5.5)})
sns.violinplot(y = dfTrain['passenger_count'])
```

Out[7]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1ede2c91eb0>
```

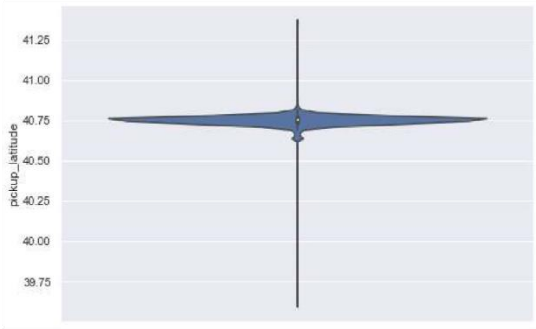


In [8]:
```
sns.set(rc={'figure.figsize':(8.4,5.5)})
sns.violinplot(y =dfTrain['fare_amount'])
```

Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x1ede2caa790>

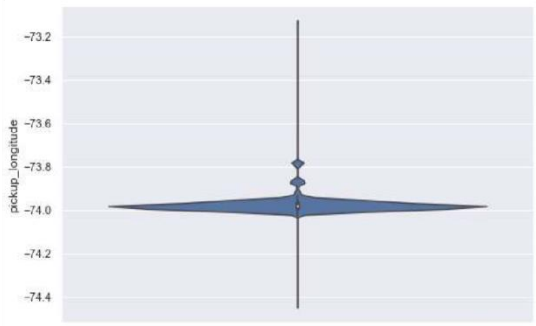
In [9]:
```
sns.set(rc={'figure.figsize':(8.4,5.5)})
sns.violinplot(y = dfTrain['pickup_latitude'])
```

Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x1ede2baecd0>



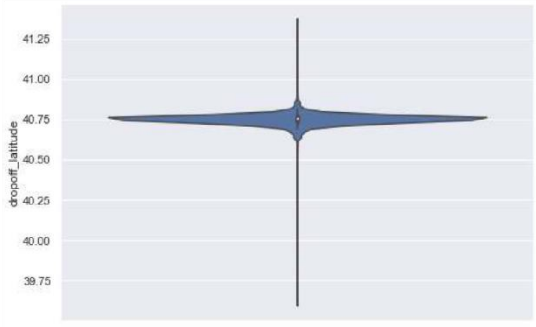
In [10]:
```
sns.set(rc={'figure.figsize':(8.4,5.5)})
sns.violinplot(y = dfTrain['pickup_longitude'])
```

Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x1ede2bfb520>



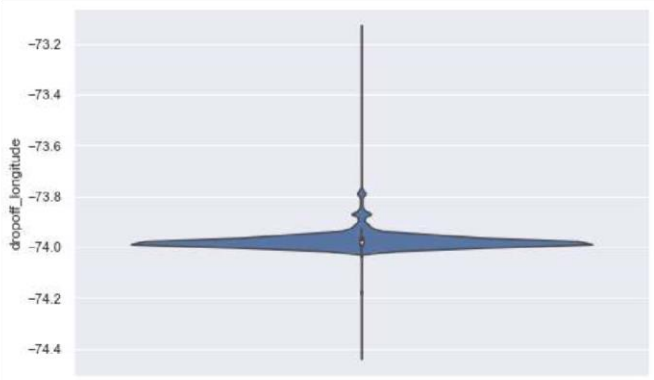
In [11]:
```
sns.set(rc={'figure.figsize':(8.4,5.5)})
sns.violinplot(y = dfTrain['dropoff_latitude'])
```

Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x1ede2cfa3d0>



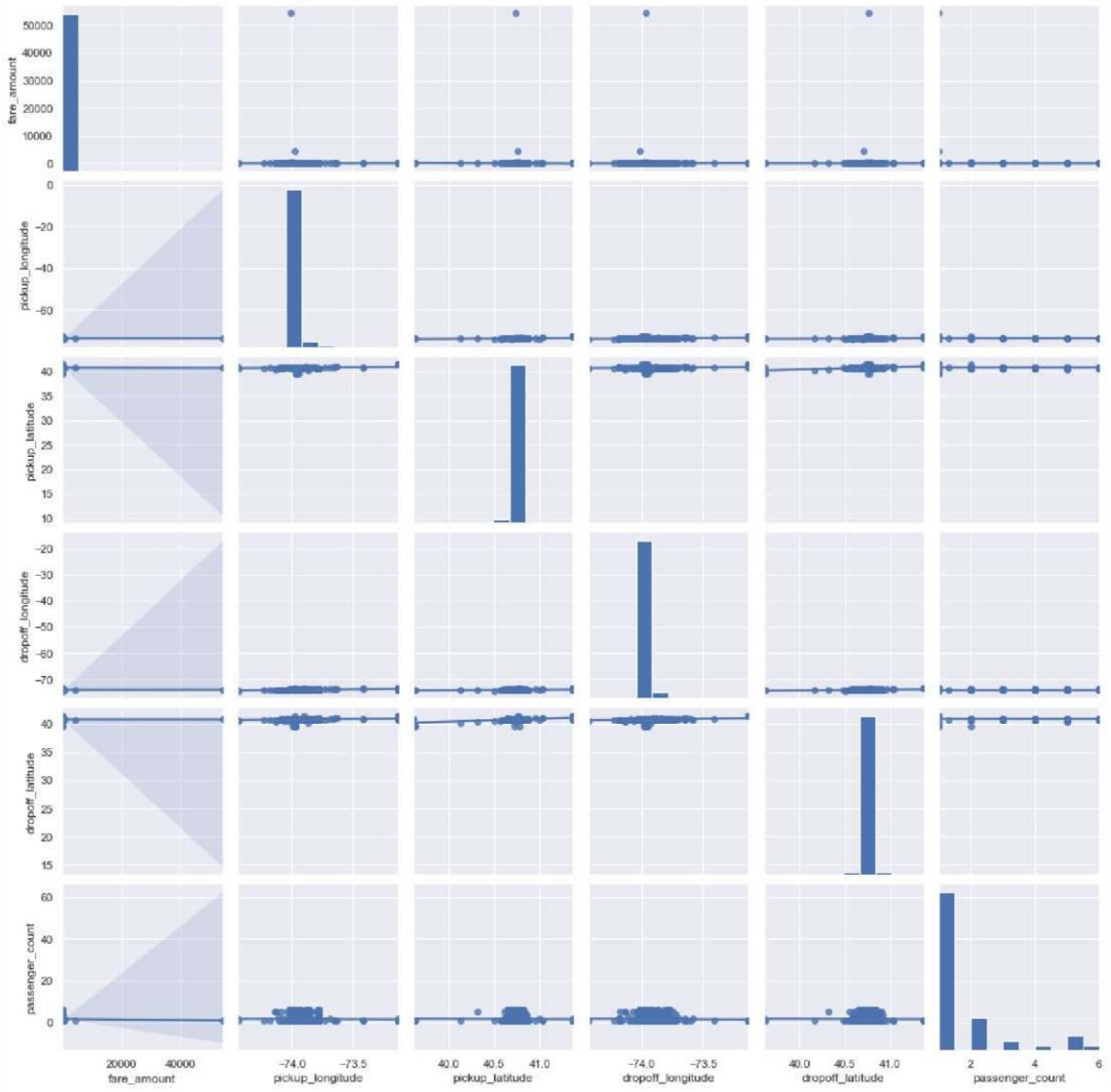
In [12]:
```
sns.set(rc={'figure.figsize':(8.4,5.5)})
sns.violinplot(y =dfTrain['dropoff_longitude'])
```

Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x1ede2d40a60>



In [13]:
```
num_var=['fare_amount','pickup_longitude','pickup_latitude','dropoff_longitude','dropoff_latitude', 'passenger_count']
sns.pairplot(data=dfTrain[num_var],kind='reg',dropna=True)
```

Out[13]: <seaborn.axisgrid.PairGrid at 0x1ede43f2a60>



Deriving new and usefull features for analysis

new features created to better explain model

features based on geographic coordinates

In [14]:
```
def dist(row):
    pickup = (row['pickup_latitude'], row['pickup_longitude'])
    dropoff = (row['dropoff_latitude'], row['dropoff_longitude'])
    return geodesic(pickup, dropoff).km

def len_trip(row):
    if (row['geodesic'] >= 15):
        return 'long'
    elif (row['geodesic'] < 15):
        return 'short'

dfTrain['geodesic'] = dfTrain.apply(dist, axis =1)
dfTest['geodesic'] = dfTest.apply(dist, axis = 1)

dfTrain['trip'] = dfTrain.apply(len_trip, axis =1)
dfTest['trip'] = dfTest.apply(len_trip, axis =1)
```

features based on pickup date_time

In [15]:
```
def season(row):
    month = row['pickup_datetime'].month
    if (month >2) & (month <= 5):
        return 'spring'
    elif (month >5) & (month <=8 ):
        return 'summer'
    elif (month >8) & (month <= 11):
         return'fall'
    elif (month > 11)|(month <= 2) :
        return 'winter'

def shift(row):
    hour = row['pickup_datetime'].hour
    if (hour > 2) & (hour <=8):
        return 'dawn'
    elif (hour > 8) & (hour <= 14):
        return 'morning'
    elif (hour > 14) & (hour <= 20):
        return 'evening'
    elif (hour > 20) | (hour <= 2):
        return 'night'

def weekDay(row):
    day = row['pickup_datetime'].dayofweek
    if (day >= 5):
        return 'weekend'
    elif (day < 5):
        return 'weekday'



#feature derivation from datetime
dfTrain['year'] = dfTrain.apply(lambda row: row['pickup_datetime'].year, axis =1)
dfTest['year'] = dfTest.apply(lambda row: row['pickup_datetime'].year, axis =1)

dfTrain['month'] = dfTrain.apply(lambda row: row['pickup_datetime'].month, axis =1)
dfTest['month'] = dfTest.apply(lambda row: row['pickup_datetime'].month, axis =1)

dfTrain['dayofwk'] = dfTrain.apply(lambda row: row['pickup_datetime'].dayofweek, axis =1)
dfTest['dayofwk'] = dfTest.apply(lambda row: row['pickup_datetime'].dayofweek, axis =1)

dfTrain['hour'] = dfTrain.apply(lambda row: row['pickup_datetime'].hour, axis =1)
dfTest['hour'] = dfTest.apply(lambda row: row['pickup_datetime'].hour, axis =1)

#new variables for seasonal surcharges
dfTrain['season'] = dfTrain.apply(season, axis =1)
dfTest['season'] = dfTest.apply(season, axis =1)

dfTrain['weekday'] = dfTrain.apply(weekDay, axis =1)
dfTest['weekday'] = dfTest.apply(weekDay, axis =1)

dfTrain['shift'] = dfTrain.apply(shift, axis =1)
dfTest['shift'] = dfTest.apply(shift, axis =1)
```

Cleaning data train data

missing values and outliers for each attribute checked

In [16]:
```
miss_val = pd.DataFrame(dfTrain.isnull().sum())
miss_val = miss_val.rename(columns = {'index': 'Variables', 0: 'Missing Count'})
miss_val['Missing Pcent'] = (miss_val['Missing Count']/len(dfTrain))*100
miss_val
```

Out[16]:

	Missing Count	Missing Pcent
fare_amount	22	0.140575
pickup_datetime	1	0.006390
pickup_longitude	0	0.000000
pickup_latitude	0	0.000000
dropoff_longitude	0	0.000000
dropoff_latitude	0	0.000000
passenger_count	55	0.351438
geodesic	0	0.000000
trip	0	0.000000
year	1	0.006390
month	1	0.006390
dayofwk	1	0.006390
hour	1	0.006390
season	1	0.006390
weekday	1	0.006390
shift	1	0.006390

Treating missing values

In first step, some observation with missing values dropped with insignificant loss in data

In [17]:
```
dfTrain.loc[(dfTrain['geodesic']<0.05), 'geodesic'] = np.nan
dfTrain = dfTrain.dropna(subset = ['fare_amount', 'pickup_datetime', 'geodesic']).reset_index(drop = True)
```

Replaceing outliers with NaN to impute values later

In second step, outlier values also replaced with NaN and masked as missing values

In [18]:
```
def replaceOutlier(col):
    Q1 = np.percentile(dfTrain[col], 25)
    Q3 = np.percentile(dfTrain[col], 75)
    Max_Bound = Q3 + 1.5*(Q3-Q1)
    Min_Bound = Q1 - 1.5*(Q3-Q1)

    dfTrain.loc[((dfTrain[col]<Min_Bound)|(dfTrain[col]>Max_Bound)), col] = np.nan
    return


to_modify = ['fare_amount', 'geodesic']

for var in to_modify:
    replaceOutlier(var)

dfTrain.describe()
```

Out[18]:

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count	geodesic	year	mon
count	14076.000000	15404.000000	15404.000000	15404.000000	15404.000000	15349.000000	14069.000000	15404.000000	15404.0000(
mean	8.949371	-73.975289	40.750763	-73.974313	40.751263	1.653026	2.478721	2011.739938	6.27187
std	4.131822	0.037878	0.036232	0.035432	0.037995	1.268635	1.660241	1.870407	3.4495-
min	1.140000	-74.438233	39.603178	-74.227047	39.604972	1.000000	0.059825	2009.000000	1.0000(
25%	5.700000	-73.992410	40.736621	-73.991376	40.736394	1.000000	1.224985	2010.000000	3.0000(
50%	8.000000	-73.982102	40.753366	-73.980612	40.754266	1.000000	2.008429	2012.000000	6.0000(
75%	11.000000	-73.968232	40.767805	-73.965603	40.768338	2.000000	3.304178	2013.000000	9.0000(
max	22.100000	-73.137393	41.366138	-73.137393	41.366138	6.000000	7.930015	2015.000000	12.0000(

Imputing missing values

After replacing outliers with missing vallues, all missing values are imputed. Two methods checked for imputing missing values

- With statistical mean: for imputing missing values first mean of data checked
- With KNN Imputation: for KNN imputation a subset of data created

In [19]:
```
dfImputer = dfTrain[['fare_amount', 'geodesic', 'year', 'month', 'hour', 'dayofwk', 'passenger_count']].copy()
dfImputer.isnull().sum()
```

Out[19]:
```
fare_amount        1328
geodesic           1335
year                  0
month                 0
hour                  0
dayofwk               0
passenger_count      55
dtype: int64
```

of the subset a sample taken for validation.

In [20]:
```
dfImputTrial = dfImputer.sample(frac = 0.02).copy().dropna().reset_index(drop = True)
dfImputValid = dfImputTrial.copy()
ranInd = np.random.randint(0, high = len(dfImputTrial), size = 5)
dfImputTrial.isnull().sum()
```

Out[20]:
```
fare_amount        0
geodesic           0
year               0
month              0
hour               0
dayofwk            0
passenger_count    0
dtype: int64
```

random values set to NaN to validate KNN Imputation

In [21]:
```
for i in ranInd:
    dfImputTrial['passenger_count'].loc[i] = np.nan
    dfImputTrial['geodesic'].loc[i] = np.nan
    dfImputTrial['fare_amount'].loc[i] = np.nan
```

With statistical mean: for imputing missing values first mean of data checked

In [22]:
```
print(dfImputer['passenger_count'].mean(), dfImputer['fare_amount'].mean(), dfImputer['geodesic'].mean())
```

```
1.6530262557821356 8.94937127024727 2.4787213725522497
```

imputed values crosschecked with true values

In [23]:
```
naImputer = KNNImputer(n_neighbors = 5)
naImputer.fit(dfImputTrial)
naImputed = naImputer.transform(dfImputTrial)
print('IP', 'VP', 'IF', 'VF', 'IG', 'VG')
for i in ranInd:
    print((naImputed[i, 6].round()),'|', dfImputValid['passenger_count'].loc[i],'||',
          (naImputed[i, 0].round(decimals = 2)),'|', dfImputValid['fare_amount'].loc[i].round(decimals = 1), '||',
          (naImputed[i, 1].round(decimals = 2)),'|', dfImputValid['geodesic'].loc[i].round(decimals = 1))
```

```
IP VP IF VF IG VG
1.0 | 2.0 || 7.8 | 4.5 || 2.13 | 1.0
2.0 | 1.0 || 8.62 | 17.0 || 2.13 | 4.8
2.0 | 1.0 || 6.62 | 12.0 || 1.13 | 3.6
3.0 | 1.0 || 7.62 | 9.5 || 1.51 | 2.9
2.0 | 1.0 || 9.22 | 4.5 || 3.01 | 1.1
```

std deviation checked before and after imputation

In [24]:
```
print(np.std(naImputed[:,6].round()), np.std(naImputed[:,0]), np.std(naImputed[:,1]))
print(dfImputValid['passenger_count'].std(),dfImputValid['fare_amount'].std(),dfImputValid['geodesic'].std())
```

```
1.3874280758538362 4.081993798048414 1.5065464202356897
1.3900615918031953 4.136580069157395 1.5191336009200622
```

imputing values with KNN Imputation for original data

In [25]:
```
theImputer = KNNImputer(n_neighbors = 5)
theImputer.fit(dfImputer)
theImputed = theImputer.transform(dfImputer)

dfTrain['passenger_count'] = theImputed[:,6].round()
dfTrain['fare_amount'] = theImputed[:,0]
dfTrain['geodesic'] = theImputed[:,1]
```

In [26]:
```
dfTrain.isnull().sum()
```

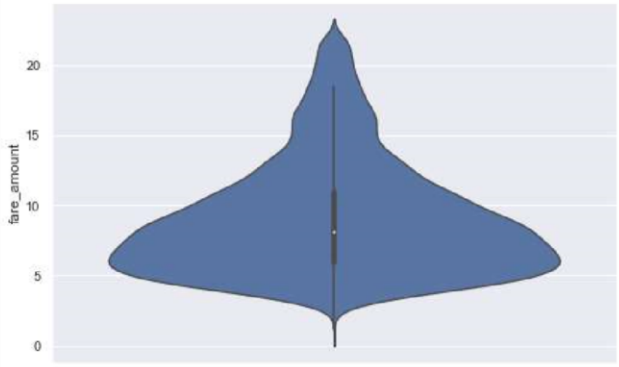
Out[26]:
```
fare_amount          0
pickup_datetime      0
pickup_longitude     0
pickup_latitude      0
dropoff_longitude    0
dropoff_latitude     0
passenger_count      0
geodesic             0
trip                 0
year                 0
month                0
dayofwk              0
hour                 0
season               0
weekday              0
shift                0
dtype: int64
```

Understanding relation between variables

distribution and correlation after missing value and oulier correction checked

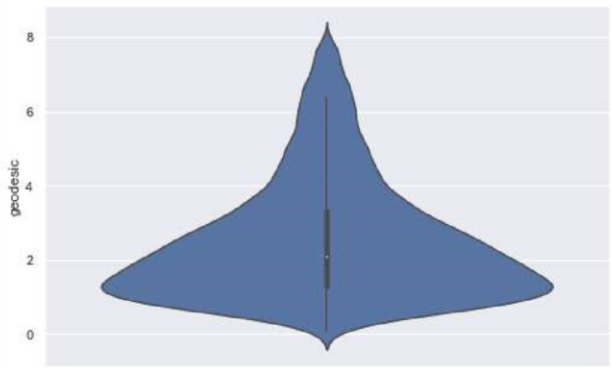
In [27]:
```
sns.set(rc={'figure.figsize':(8.4,5.5)})
sns.violinplot(y =dfTrain['fare_amount'])
```

Out[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1ede636dac0>
```



In [28]:
```
sns.set(rc={'figure.figsize':(8.4,5.5)})
sns.violinplot(y =dfTrain['geodesic'])
```

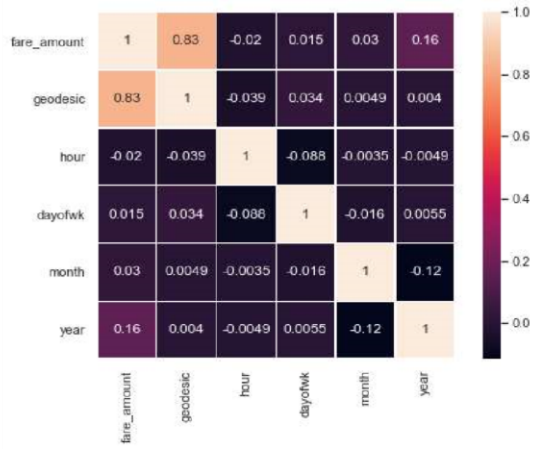
Out[28]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1ede63d1cd0>
```



If any variables are correleated is checked with help of heatmap of correlation coeficient

In [29]:
```
sns.heatmap(dfTrain[['fare_amount', 'geodesic', 'hour', 'dayofwk','month', 'year']].corr(), square=True, linewidths=0.5,
linecolor='w',annot=True)
```

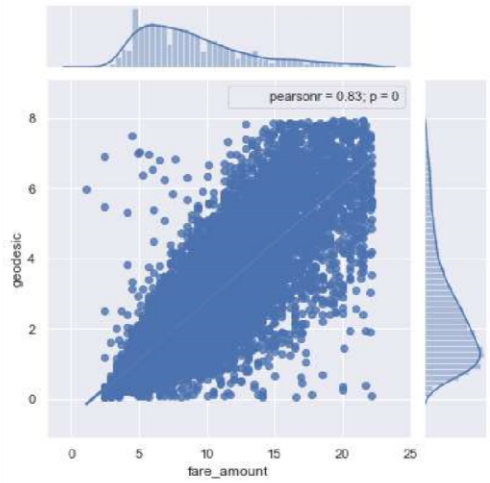
Out[29]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1ede6409790>
```



In [30]:
```
sns.jointplot(x='fare_amount',y='geodesic',data=dfTrain, kind = 'reg').annotate(sp.stats.pearsonr)
```

```
C:\Users\ASUS\anaconda3\lib\site-packages\seaborn\axisgrid.py:1840: UserWarning: JointGrid annotation is deprecated and
will be removed in a future release.
  warnings.warn(UserWarning(msg))
```

Out[30]:
```
<seaborn.axisgrid.JointGrid at 0x1ede2c08b20>
```



Transforming variables to suit model

categorical varibles transformed with one-hot coding. old variables dropped from the data set. one dummy variable of each category dropped to avoid multi-colinearity problem.

In [31]:
```
categorical = ['passenger_count', 'year', 'season', 'weekday', 'shift', 'trip']
integral = ['passenger_count', 'year']

dfTrain[integral] = dfTrain[integral].apply(np.int64)
dfTest[integral] = dfTest[integral].apply(np.int64)


#dummy variables created
for var in categorical:
    dfDummy1 = pd.get_dummies(dfTrain[var], prefix = var)
    dfTrain = dfTrain.join(dfDummy1)

    dfDummy2 = pd.get_dummies(dfTest[var], prefix = var)
    dfTest = dfTest.join(dfDummy2)
```

In [32]:
```
#old variables dropped
dfTrain = dfTrain.drop(['pickup_datetime', 'pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitud
e',
                        'passenger_count', 'year', 'season', 'weekday', 'month', 'shift', 'trip', 'dayofwk', 'hour'], axis =1)
dfTest = dfTest.drop(['pickup_datetime', 'pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude',
                        'passenger_count', 'year', 'season', 'weekday', 'month', 'shift', 'trip', 'dayofwk', 'hour'], axis =1)


#one dummy variable dropped
dfTrain = dfTrain.drop(['passenger_count_1', 'year_2009', 'season_fall', 'weekday_weekday', 'shift_dawn', 'trip_short'],
axis =1)
dfTest = dfTest.drop(['passenger_count_1', 'year_2009', 'season_fall', 'weekday_weekday', 'shift_dawn', 'trip_short'], a
xis =1)
```

Checking multicolinearity in data

with help of VIF, multicolinearity in data is checked.

- if VIF = 1 -> No colinearity in any variables.
- if 1 <= VIF <= 5 -> Moderate corinearity
- if VIF > 5 -> High colinearity if multiple variables have more than 5 VIF, one with highest VIF will be dropped

In [33]:
```
features = " + ".join(dfTrain.columns)
features = features.replace("fare_amount + ", "")

target, attributes = dmatrices("fare_amount ~ " + features, dfTrain, return_type='dataframe')
```

In [34]:
```
vif = pd.DataFrame()
vif['VIF'] = [variance_inflation_factor(attributes.values, i) for i in range(attributes.shape[1])]
vif['INPUTS'] = attributes.columns
vif
```

Out[34]:

	VIF	INPUTS
0	18.446588	Intercept
1	1.021056	geodesic
2	1.040459	passenger_count_2
3	1.019080	passenger_count_3
4	1.011632	passenger_count_4
5	1.025666	passenger_count_5
6	1.017437	passenger_count_6
7	1.687318	year_2010
8	1.687776	year_2011
9	1.709798	year_2012
10	1.709428	year_2013
11	1.666284	year_2014
12	1.409693	year_2015
13	1.640248	season_spring
14	1.549578	season_summer
15	1.586209	season_winter
16	1.015841	weekday_weekend
17	2.179903	shift_evening
18	2.123685	shift_morning
19	2.055585	shift_night
20	1.003065	trip_long

Building models and checking performance

- train data split to validate performance.
- sevral functions defined to visualize and calculate model statistics

In [35]:
```
X = dfTrain.drop(['fare_amount'], axis = 1).values
Y = dfTrain['fare_amount'].values

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.3, random_state=35)
```

In [36]:
```
def scores(y_true, y_pred):

    mape = np.mean(np.abs((y_true - y_pred)/y_true))*100

    r2 = mtr.r2_score(y_true, y_pred)

    rms = np.sqrt(mtr.mean_squared_error(y_true, y_pred))

    return [mape, r2, rms]

def model_scores(model):

    y_trPred = model.predict(X_train)
    trScore = scores(Y_train, y_trPred)

    y_tsPred = model.predict(X_test)
    tsScore = scores(Y_test, y_tsPred)

    data = {'train' : trScore, 'test' : tsScore}
    dfScore = pd.DataFrame(data, index = ['MAPE', 'R2', 'RMSE'])

    print(dfScore)
    return dfScore

def model_plot(model):

    coefModel = model.coef_

    mpl.pyplot.plot(range(len(dfTest.columns)), coefModel)
    mpl.pyplot.xticks(range(len(dfTest.columns)), dfTest.columns.values, rotation=75)

    return

def tree_plot(model):

    coefModel = model.feature_importances_

    mpl.pyplot.plot(range(len(dfTest.columns)), coefModel)
    mpl.pyplot.xticks(range(len(dfTest.columns)), dfTest.columns.values, rotation=75)

    return
```

Linear Regression Model

In [37]:
```
lin_reg_trial = LinearRegression()

param_grid = {'fit_intercept': [True, False], 'normalize': [True, False], 'copy_X': [True, False]}

LinRegCV = GridSearchCV(lin_reg_trial, param_grid, cv=5, scoring='r2')
LinRegCV.fit(X, Y)

print("Optimum Parameters: {}".format(LinRegCV.best_params_))
print("Optimum score     : {}".format(LinRegCV.best_score_))
```

```
Optimum Parameters: {'copy_X': True, 'fit_intercept': True, 'normalize': True}
Optimum score     : 0.7314191269457941
```

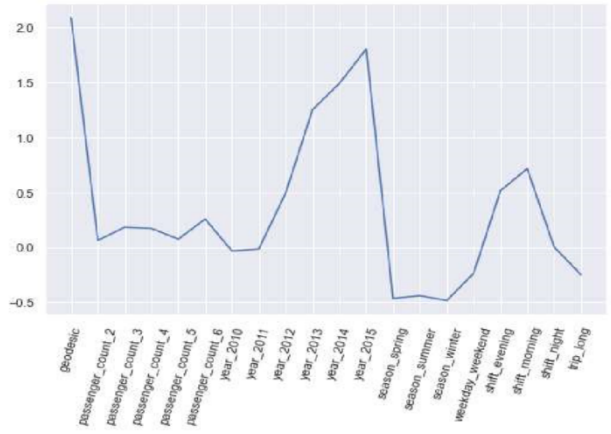
In [38]:
```
lin_reg_model = LinearRegression(copy_X = True, fit_intercept = True, normalize = True)
lin_reg_model.fit(X_train,Y_train)

Yp_linreg = lin_reg_model.predict(X_test)

scoreLinReg = model_scores(lin_reg_model)

model_plot(lin_reg_model)
```

```
          train       test
MAPE  17.448193  17.876085
R2     0.727850   0.742516
RMSE   2.112474   2.112242
```



Ridge Regression Model

In [39]:
```
ridge_reg_trial = Ridge()

param_grid = {'alpha':np.logspace(-4, 0, 75), 'normalize':[True,False], 'max_iter':range(100,1000,100)}

RidgeRegCV = GridSearchCV(ridge_reg_trial, param_grid, cv=5, scoring='r2')
RidgeRegCV.fit(X, Y)

print("Optimum Parameters: {}".format(RidgeRegCV.best_params_))
print("Optimum score     : {}".format(RidgeRegCV.best_score_))
```

```
Optimum Parameters: {'alpha': 1.0, 'max_iter': 100, 'normalize': False}
Optimum score     : 0.7314196704388056
```

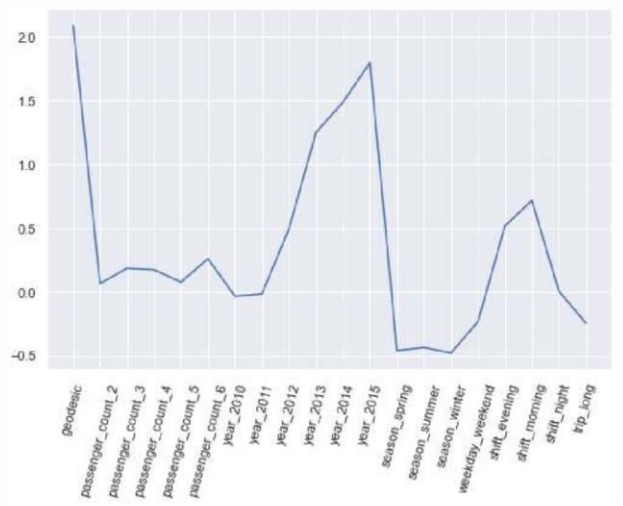
In [40]:
```
ridge_reg_model = Ridge(alpha = 1.0, max_iter = 100, normalize = False)
ridge_reg_model.fit(X_train,Y_train)

Yp_ridgeReg = ridge_reg_model.predict(X_test)

scoreRidgeReg = model_scores(ridge_reg_model)

model_plot(ridge_reg_model)
```

```
          train       test
MAPE  17.447869  17.875432
R2     0.727850   0.742517
RMSE   2.112475   2.112241
```



Lasso Regression Model

In [41]:
```
lasso_reg_trial = Lasso()

param_grid = {'alpha':np.logspace(-4, 0, 75), 'normalize':[True,False], 'max_iter':range(100,1000,100)}

LassoRegCV = GridSearchCV(lasso_reg_trial, param_grid, cv=5, scoring='r2')
LassoRegCV.fit(X, Y)

print("Optimum Parameters: {}".format(LassoRegCV.best_params_))
print("Optimum score     : {}".format(LassoRegCV.best_score_))
```

```
Optimum Parameters: {'alpha': 0.0004452958509942655, 'max_iter': 100, 'normalize': False}
Optimum score     : 0.7314249966863244
```

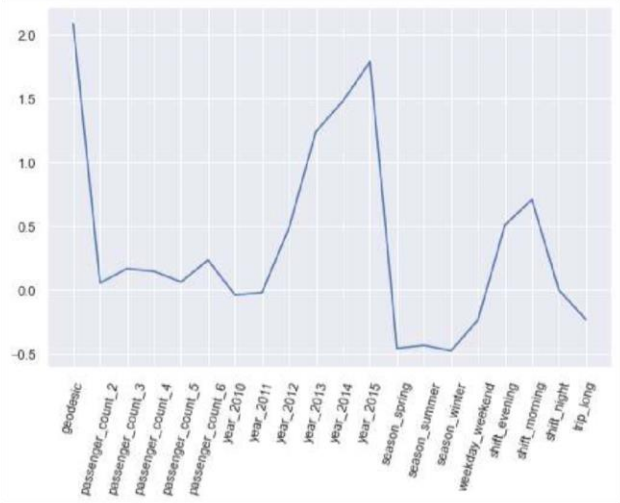
In [42]:
```
lasso_reg_model = Lasso(alpha = 0.0004452958509942655, max_iter = 100, normalize = False)
lasso_reg_model.fit(X_train,Y_train)

Yp_lassoReg = lasso_reg_model.predict(X_test)

scoreLassoReg = model_scores(lasso_reg_model)

model_plot(lasso_reg_model)
```

```
          train       test
MAPE  17.445332  17.874590
R2     0.727846   0.742497
RMSE   2.112492   2.112322
```



Decision Tree Regression Model

In [43]:
```
detree_reg_trial = DecisionTreeRegressor()

param_grid = {'criterion': ['mse', 'mae'], 'max_depth': range(2,10,2), 'min_samples_split': range(2,10,2)}

DeTreeRegCV = RandomizedSearchCV(detree_reg_trial, param_grid, cv=5, scoring='r2')
DeTreeRegCV.fit(X, Y)

print("Optimum Parameters: {}".format(DeTreeRegCV.best_params_))
print("Optimum score     : {}".format(DeTreeRegCV.best_score_))
```

```
Optimum Parameters: {'min_samples_split': 4, 'max_depth': 6, 'criterion': 'mse'}
Optimum score     : 0.7125962743074015
```

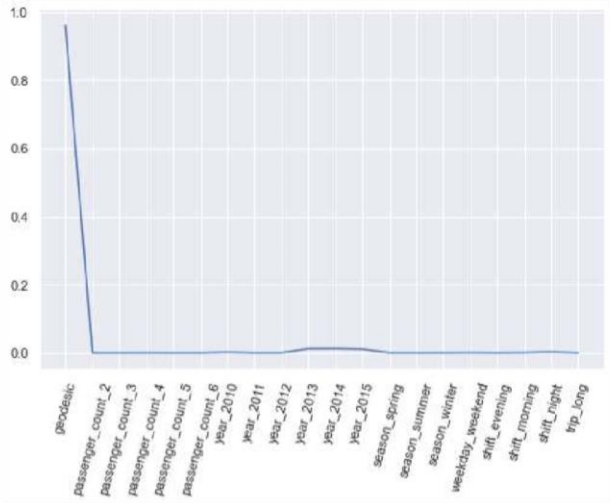
In [45]:
```
detree_reg_model = DecisionTreeRegressor(min_samples_split = 4, max_depth = 6, criterion = 'mse')
detree_reg_model.fit(X_train,Y_train)

Yp_detreeReg = detree_reg_model.predict(X_test)

scoreDetreeReg = model_scores(detree_reg_model)

tree_plot(detree_reg_model)
```

```
          train       test
MAPE  17.536055  18.474367
R2     0.729539   0.727468
RMSE   2.105909   2.173092
```



Random Forest Regression Model

In [46]:
```
randfr_reg_trial = RandomForestRegressor()

param_grid = {'n_estimators': range(100,500,100), 'max_depth': range(5,20,3), 'min_samples_leaf':range(2,5,1), 'max_features':['auto','sqrt','log2'], 'bootstrap': [True, False], 'min_samples_split': range(2,5,1)}

RandFrRegCV = RandomizedSearchCV(randfr_reg_trial, param_grid, cv=5)
RandFrRegCV.fit(X,Y)

print("Optimum Parameters: {}".format(RandFrRegCV.best_params_))
print("Optimum score     : {}".format(RandFrRegCV.best_score_))
```

```
Optimum Parameters: {'n_estimators': 200, 'min_samples_split': 3, 'min_samples_leaf': 4, 'max_features': 'auto', 'max_depth': 14, 'bootstrap': True}
Optimum score     : 0.7257660829011023
```

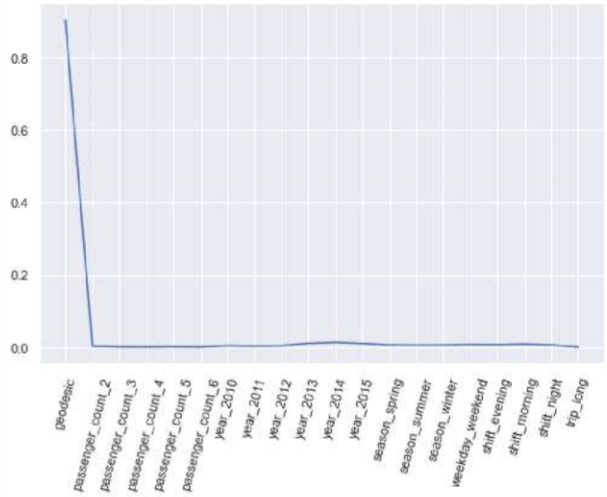
In [47]:
```
rforest_reg_model = RandomForestRegressor(n_estimators = 200, min_samples_split = 3, min_samples_leaf = 4, max_features = 'auto', max_depth = 14, bootstrap = True)
rforest_reg_model.fit(X_train,Y_train)

Yp_rforestReg = rforest_reg_model.predict(X_test)

scoreRForestReg = model_scores(rforest_reg_model)

tree_plot(rforest_reg_model)
```

	train	test
MAPE	13.759257	17.931695
R2	0.830670	0.742405
RMSE	1.666303	2.112698



Improving Accuracy through xgboost

In [48]:
```
dData = xgb.DMatrix(data=X,label=Y)
dTrain = xgb.DMatrix(X_train, label=Y_train)
dTest = xgb.DMatrix(X_test)

params = {"objective":"reg:squarederror",'colsample_bytree': 0.5,'learning_rate': 0.1, 'max_depth': 5, 'alpha': 10}
cv_results = xgb.cv(dtrain=dData, params=params, nfold=5, num_boost_round=75, early_stopping_rounds=10, metrics="rmse", as_pandas=True, seed=123)

print((cv_results["test-rmse-mean"]).tail(1))
```

```
74    2.082558
Name: test-rmse-mean, dtype: float64
```

In [49]:
```
xgb_reg_trial = xgb.XGBRegressor()

param_grid = {'n_estimators': range(100,500,100),  'reg_alpha':np.logspace(-3, 0, 30), 'colsample_bytree': np.arange(0.1,1,0.2), 'learning_rate': np.arange(0.1, 1, 0.1)}
XGBRegCV = RandomizedSearchCV(xgb_reg_trial, param_grid, cv=5)
XGBRegCV.fit(X,Y)

print("Optimum Parameters: {}".format(XGBRegCV.best_params_))
print("Optimum score     : {}".format(XGBRegCV.best_score_))
```

```
Optimum Parameters: {'reg_alpha': 0.11721022975334805, 'n_estimators': 100, 'learning_rate': 0.1, 'colsample_bytree': 0.9000000000000001}
Optimum score     : 0.7384874075837606
```

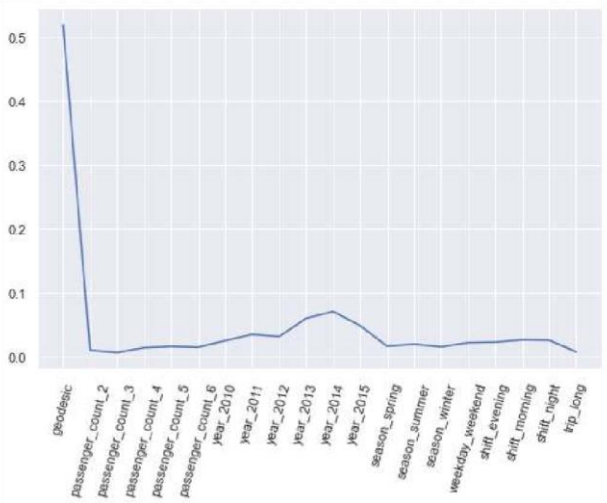
In [50]:
```
xgb_reg_model = xgb.XGBRegressor(reg_alpha = 0.11721022975334805, n_estimators = 100, learning_rate = 0.1, max_depth = 5
, colsample_bytree = 0.9000000000000001)
xgb_reg_model.fit(X_train,Y_train)

Yp_xgbReg = xgb_reg_model.predict(X_test)

scoreRForestReg = model_scores(xgb_reg_model)

tree_plot(xgb_reg_model)
```

```
          train       test
MAPE  15.873809  17.403499
R2     0.775923   0.752668
RMSE   1.916841   2.070184
```



Training entire train data set and predicting values for test dataset

In [51]:
```
xgb_reg_final = xgb.XGBRegressor(reg_alpha = 0.11721022975334805, n_estimators = 100, learning_rate = 0.1, max_depth = 5
, colsample_bytree = 0.9000000000000001)
xgb_reg_final.fit(X, Y)

Y_Check = xgb_reg_final.predict(X)

testScores = scores(Y, Y_Check)
pd.DataFrame(testScores, index = ['MAPE', 'R2', 'RMSE'])
```

```
C:\Users\ASUS\anaconda3\lib\site-packages\xgboost\data.py:112: UserWarning: Use subset (sliced data) of np.ndarray is no
t recommended because it will generate extra copies and increase memory consumption
  warnings.warn(
```

Out[51]:

	0
MAPE	16.103973
R2	0.774570
RMSE	1.938980

Predicing fare amount for test dataset and saving output and final model.

In [52]:
```
Y_Out = xgb_reg_final.predict(dfTest.values)

pd.DataFrame({'fare_amount' : Y_Out}).to_csv("cabfare_xgb_output_py.csv")

joblib.dump(xgb_reg_final, 'cabfare_xgbmodel_py.pkl')
```

```
C:\Users\ASUS\anaconda3\lib\site-packages\xgboost\data.py:112: UserWarning: Use subset (sliced data) of np.ndarray is no
t recommended because it will generate extra copies and increase memory consumption
  warnings.warn(
```

Out[52]:
```
['cabfare_xgbmodel_py.pkl']
```

In [ ]:

7 Appendix B

R CODE

```
#############################################################################
###-------------clearing environment and setting up directory--------------###

rm(list = ls())
setwd("K:/Data_Science/Project_01/Trials")
getwd()

###--------------------------- importing libraries -------------------------###

libs = c("ggplot2", "geosphere", "corrgram", "DMwR", "usdm", "caret", "raster",
"rpart", "randomForest", "xgboost","stats", "sp")

#load Packages
lapply(libs, require, character.only = TRUE)
rm(libs)


###----------------------------- loading data ------------------------------###

dfTrain <- read.csv("train_cab.csv", header=TRUE, na.strings = c(" ", "",
"NA"))
dfTest <- read.csv("test.csv", header=TRUE)

###----------------------------overview of data----------------------------###

str(dfTrain)
#'data.frame':    16067 obs. of  7 variables:
#$ fare_amount      : chr  "4.5" "16.9" "5.7" "7.7" ...
#$ pickup_datetime  : chr  "2009-06-15 17:26:21 UTC"...
#$ pickup_longitude : num  -73.8 -74 -74 -74 -74 ...
#$ pickup_latitude  : num  40.7 40.7 40.8 40.7 40.8 ...
#$ dropoff_longitude: num  -73.8 -74 -74 -74 -74 ...
#$ dropoff_latitude : num  40.7 40.8 40.8 40.8 40.8 ...
#$ passenger_count  : num  1 1 2 1 1 1 1 1 1 2 ...

str(dfTest)
#'data.frame':    9914 obs. of  6 variables:
#$ pickup_datetime  : chr  "2015-01-27 13:08:24 UTC"...
#$ pickup_longitude : num  -74 -74 -74 -74 -74 ...
#$ pickup_latitude  : num  40.8 40.7 40.8 40.8 40.8 ...
#$ dropoff_longitude: num  -74 -74 -74 -74 -74 ...
#$ dropoff_latitude : num  40.7 40.7 40.7 40.8 40.7 ...
#$ passenger_count  : int  1 1 1 1 1 1 1 1 1 1 ...

summary(dfTrain)
summary(dfTest)
```

```
###-------------------------adjusting data types--------------------------###

dfTrain$pickup_datetime <- as.POSIXct(dfTrain$pickup_datetime, format = '%Y-%m-
%d %H:%M:%S', tz ='UTC')

dfTrain$fare_amount <- as.numeric(dfTrain$fare_amount)
dfTrain$passenger_count <- as.integer(dfTrain$passenger_count)

dfTest$pickup_datetime <- as.POSIXct(dfTest$pickup_datetime, format = '%Y-%m-%d
%H:%M:%S', tz ='UTC')

###------------------- removing non-sensical values-----------------------###

dfTrain <- subset(dfTrain, !(dfTrain$fare_amount < 1))
#observation with negative fare amount removed

dfTrain <- subset(dfTrain, !((dfTrain$passenger_count < 1) |
(dfTrain$passenger_count > 6)))
#observation with illogical passenger count removed

dfTrain <- subset(dfTrain, !((dfTrain$pickup_longitude < -
75.5)|(dfTrain$pickup_longitude > -71.8)))
#observation with out of bound pickup longitude removed

dfTrain <- subset(dfTrain, !((dfTrain$pickup_latitude < 39.5) |
(dfTrain$pickup_latitude > 41.9)))
#observation with out of bound pickup latitude removed

dfTrain <- subset(dfTrain, !((dfTrain$dropoff_longitude < -75.5) |
(dfTrain$dropoff_longitude > -71.8)))
#observation with out of bound dropoff longitude removed

dfTrain <- subset(dfTrain, !((dfTrain$dropoff_latitude < 39.5) |
(dfTrain$dropoff_latitude > 41.9)))
#observation with out of bound dropoff latitude removed

dfTrain <- na.omit(dfTrain)
#missing values omited since number is too low compared to data size

###------------------distribution of individual variables-----------------###
ggplot(dfTrain, aes(x = dropoff_latitude)) + geom_area( stat = 'count')
ggplot(dfTrain, aes(x = dropoff_longitude)) + geom_area( stat = 'count')
ggplot(dfTrain, aes(x = pickup_latitude)) + geom_area( stat = 'count')
ggplot(dfTrain, aes(x = pickup_longitude)) + geom_area( stat = 'count')
ggplot(dfTrain, aes(x = fare_amount)) + geom_area( stat = 'count')
ggplot(dfTrain, aes(x = passenger_count)) + geom_area( stat = 'count')
```

```
###-------------------------feature tranformation-------------------------###

#function to create new categorical variable based on length of trip
trip_len <- function(distance){

  if (distance >=15){ return('long') }
  else if (distance <15){ return('short')}

}

#new varaible geodesic distance
dfTrain$distGeod <- distGeo(cbind(dfTrain$pickup_longitude,
dfTrain$pickup_latitude), cbind(dfTrain$dropoff_longitude,
dfTrain$dropoff_latitude))/1000
dfTest$distGeod <- distGeo(cbind(dfTest$pickup_longitude,
dfTest$pickup_latitude), cbind(dfTest$dropoff_longitude,
dfTest$dropoff_latitude))/1000

#new varaible trip length
dfTrain$tripLength <- as.factor(as.character(lapply(X = dfTrain$distGeod, FUN =
trip_len)))
dfTest$tripLength <- as.factor(as.character(lapply(dfTest$distGeod, trip_len)))

#function to create new categorical variable based on month
season <- function(month){

  if (!(is.na(month))){
    if ((month > 2)&(month <= 5 )){ return('spring') }
    else if ((month > 5)&(month <= 8 )){ return('summer') }
    else if ((month > 8)&(month <= 11 )){ return('fall') }
    else if ((month > 11)|(month <= 2 )){ return('winter') }
  }

}

#function to create new categorical variable based on hour
shift <- function(hour){

  if (!(is.na(hour))){
    if ((hour > 2) & (hour <= 8 )){ return('dawn') }
    else if ((hour > 8) & (hour <= 14 )){ return('morning') }
    else if ((hour > 14) & (hour <= 20 )){ return('evening') }
    else if ((hour > 20) | (hour <= 2 )){ return('night') }
  }

}
```

```
#function to create new categorical variable based on weekday
weekend <- function(wkday){

  if (!(is.na(wkday))){
    if (wkday >= 6){ return('wkend') }
    else if (wkday < 6 ){ return('wkday')}
  }

}

#new varaible pickup year
dfTrain$p_year <- as.factor(format(dfTrain$pickup_datetime,"%Y"))
dfTest$p_year <- as.factor(format(dfTest$pickup_datetime,"%Y"))

#new varaible pickup month
dfTrain$p_mnth <- as.integer(format(dfTrain$pickup_datetime,"%m"))
dfTest$p_mnth <- as.integer(format(dfTest$pickup_datetime,"%m"))

#new varaible pickup day
dfTrain$p_wkdy <- as.integer(format(as.Date(dfTrain$pickup_datetime),"%u"))
dfTest$p_wkdy <- as.integer(format(as.Date(dfTest$pickup_datetime),"%u"))

#new varaible pickup hour
dfTrain$p_hour <- as.integer(format(dfTrain$pickup_datetime,"%H"))
dfTest$p_hour <- as.integer(format(dfTest$pickup_datetime,"%H"))

#new varaible pickup season
dfTrain$p_sson <- as.factor(as.character(lapply(dfTrain$p_mnth, season)))
dfTest$p_sson <- as.factor(as.character(lapply(dfTest$p_mnth, season)))

#new varaible pickup shift
dfTrain$p_shft <- as.factor(as.character(lapply(dfTrain$p_hour, shift)))
dfTest$p_shft <- as.factor(as.character(lapply(dfTest$p_hour, shift)))

#new varaible pickup weekend
dfTrain$p_wknd <- as.factor(as.character(lapply(dfTrain$p_wkdy, weekend)))
dfTest$p_wknd <- as.factor(as.character(lapply(dfTest$p_wkdy, weekend)))


###-----------------------------outlier analysis-----------------------------###

#function to canculate and replace outliers from data with NA
```

```
replace_outlier <- function(column){

  lw_lmt <- (quantile(column, 0.25) - 1.5 * IQR(column, na.rm = FALSE))
  up_lmt <- (quantile(column, 0.75) + 1.5 * IQR(column, na.rm = FALSE))
  column[(column > up_lmt)|(column < lw_lmt)] <- NA
  return(column)

}

#outlier replcaed with NA for continuous variables
dfTrain$fare_amount <- replace_outlier(dfTrain$fare_amount)
dfTrain$distGeod <- replace_outlier(dfTrain$distGeod)

#trial to impute NA values
##sample from train data to perform trial
dfImputTrial <- na.omit(dfTrain)
dfImputTrial <- dfImputTrial[sample(nrow(dfImputTrial), 500, replace = F), ]
dfImputTrial <- subset(dfImputTrial, select = c(fare_amount, distGeod, p_year,
p_mnth, p_hour, p_wkdy, passenger_count))
rownames(dfImputTrial) <- 1:NROW(dfImputTrial)

##5 random values set to na
fare_sample <- c(7, 130, 375, 283, 461)
geod_sample <- c(38, 271, 459, 390, 135)

true_fare <- dfImputTrial$fare_amount[fare_sample]
true_geod <- dfImputTrial$distGeod[geod_sample]

dfImputTrial$fare_amount[fare_sample] <- NA
dfImputTrial$distGeod[geod_sample] <- NA


##imputing with mean
mean_fare <- mean(dfTrain$fare_amount, na.rm = TRUE)
mean_geod <- mean(dfTrain$distGeod, na.rm = TRUE)

absErr_mFare <- sum(abs(true_fare - mean_fare))
#Error in mean imputation: [1] 15.64046
absErr_mGeod <- sum(abs(true_geod - mean_geod))
#Error in mean imputation:[1] 4.055288

##imputing with KNN
dfImputTrial <- knnImputation(dfImputTrial, k = 3)

knn_fare <- dfImputTrial$fare_amount[fare_sample]
knn_geod <- dfImputTrial$distGeod[geod_sample]
```

```
absErr_kFare <- sum(abs(true_fare - knn_fare))
#Error in KNN imputation:[1] 7.74958
absErr_kGeod <- sum(abs(true_geod - knn_geod))
#Error in KNN imputation:[1] 3.017385

#Based on absolute error KNN method is selected for imputation
dfImputation <- subset(dfTrain, select = c(fare_amount, distGeod, p_year,
p_mnth, p_hour, p_wkdy, passenger_count))
dfImputation <- knnImputation(dfImputation, k = 3)
sum(is.na(dfImputation))
#missing values in data: [1] 0

dfTrain$fare_amount <- dfImputation$fare_amount
dfTrain$distGeod <- dfImputation$distGeod

sum(is.na(dfTrain))
#missing values in data: [1] 0

###----------------------------feature selection--------------------------###

#correlation analysis
corrgram(dfImputation, order = F,
         upper.panel=panel.pie, text.panel=panel.txt, main = "Correlation
Plot")

#scatter plot between geodesic distance and fare amount
ggplot(dfTrain, aes(distGeod, fare_amount)) + geom_point()

#adjusting data type for passenger count
dfTrain$passenger_count <- as.factor(dfTrain$passenger_count)
dfTest$passenger_count <- as.factor(dfTest$passenger_count)

#removing used variables
dfTrain <- subset(dfTrain, select = -c(pickup_datetime, pickup_longitude,
pickup_latitude, dropoff_longitude, dropoff_latitude, p_mnth, p_wkdy, p_hour))
dfTest <- subset(dfTest, select = -c(pickup_datetime, pickup_longitude,
pickup_latitude, dropoff_longitude, dropoff_latitude, p_mnth, p_wkdy, p_hour))

#one-hot coding categorical variables
dummy <- dummyVars(" ~ . ", data = dfTrain, fullRank = TRUE)
dfTrain <- data.frame(predict(dummy, newdata = dfTrain))

dummy <- dummyVars(" ~ . ", data = dfTest, fullRank = TRUE)
dfTest <- data.frame(predict(dummy, newdata = dfTest))
```

```
#vif analysis for multicolinearity

vif(dfTrain[,-1])

#           Variables      VIF
#1  passenger_count.2   1.035537
#2  passenger_count.3   1.019321
#3  passenger_count.4   1.012757
#4  passenger_count.5   1.023401
#5  passenger_count.6   1.017955
#6           distGeod   1.019970
#7   tripLength.short   1.004666
#8        p_year.2010   1.692045
#9        p_year.2011   1.686567
#10       p_year.2012   1.704758
#11       p_year.2013   1.705418
#12       p_year.2014   1.666356
#13       p_year.2015   1.373740
#14     p_sson.spring   1.627420
#15     p_sson.summer   1.549805
#16     p_sson.winter   1.588536
#17    p_shft.evening   2.184980
#18    p_shft.morning   2.139119
#19      p_shft.night   2.082691
#20      p_wknd.wkend   1.019661
# no vif more than 5

###------------------------------model building-----------------------------###

#train-test split
set.seed(1000)
tr.idx = createDataPartition(dfTrain$fare_amount,p=0.75,list = FALSE)
to_train <- dfTrain[tr.idx, ]
to_test <- dfTrain[-tr.idx, ]

#linear regression
linearModel <- lm(fare_amount ~.,data=to_train)
summary(linearModel)
linearPredTr = predict(linearModel,to_test[,2:21])
regr.eval(to_test[,1],linearPredTr)
#mae       mse      rmse      mape
#1.5290301 4.6964678 2.1671335 0.1811715
```

```
#decision tree regression
DTModel <- rpart(fare_amount ~ ., data = to_train, method = "anova")
summary(DTModel)
dtreePredTr <- predict(DTModel,to_test[,2:21])
regr.eval(to_test[,1], dtreePredTr)
#mae       mse      rmse      mape
#1.7642704 5.8476199 2.4181853 0.2122266

#random forest regression
RFModel <- randomForest(fare_amount ~., data=to_train)
summary(RFModel)
rforstPredTr <- predict(RFModel, to_test[,2:21])
regr.eval(to_test[,1], rforstPredTr)
#mae       mse      rmse      mape
#1.5355820 4.6534580 2.1571875 0.1841384

#improving accuracy through XGBoost regression
to_train_matrix <- as.matrix(sapply(to_train[-1],as.numeric))
to_test_matrix <- as.matrix(sapply(to_test[-1],as.numeric))

XGBModel <- xgboost(data = to_train_matrix, label = to_train$fare_amount,
nrounds = 15,verbose = FALSE)
summary(XGBModel)
xgbPredTr <- predict(XGBModel, to_test_matrix)
regr.eval(to_test[,1], xgbPredTr)
#mae       mse      rmse      mape
#1.4774921 4.5175955 2.1254636 0.1734023

###------------------------------final model--------------------------------###
dfTrain_matrix <- as.matrix(sapply(dfTrain[-1],as.numeric))
dfTest_matrix <- as.matrix(sapply(dfTest,as.numeric))

Final_XGB_Model <- xgboost(data = dfTrain_matrix, label = dfTrain$fare_amount,
nrounds = 15,verbose = FALSE)
XGB_Prediction <- predict(Final_XGB_Model, dfTest_matrix)

XGB_Pred = data.frame("fare_amount" = XGB_Prediction)

#saving predictions in csv file
write.csv(XGB_Pred,"cabfare_xgb_output_r.csv",row.names = TRUE)

#saving model in dump format
saveRDS(Final_XGB_Model, "./cabfare_xgbmodel_r")

##############################################################################
```