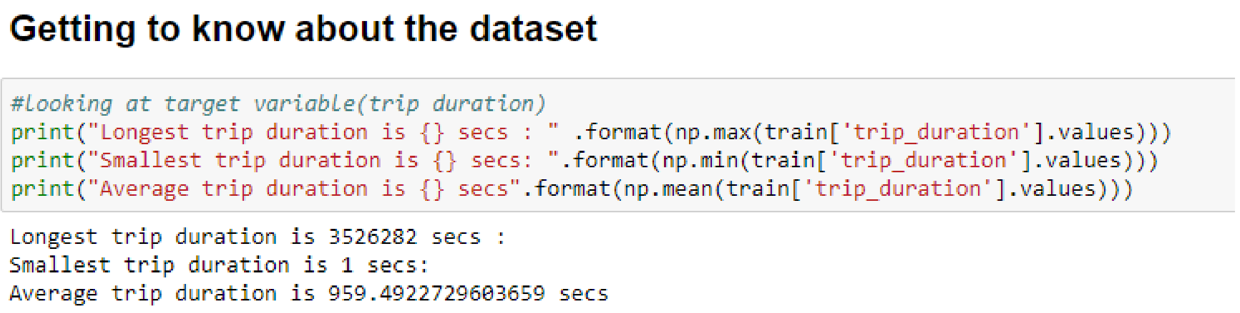
Taxi Trip Duration Analysis

Virinchi Ande

1. **Business Understanding Phase:**
   To predict the total ride duration of taxi trips in New York City and analyze the dataset for thoughtful insights.

2. **Data Understanding Phase:**
   The Dataset consists of 1458643 entries and 11 features with Trip_Duration as the Target Variable.

2.1 Data Visualizations:



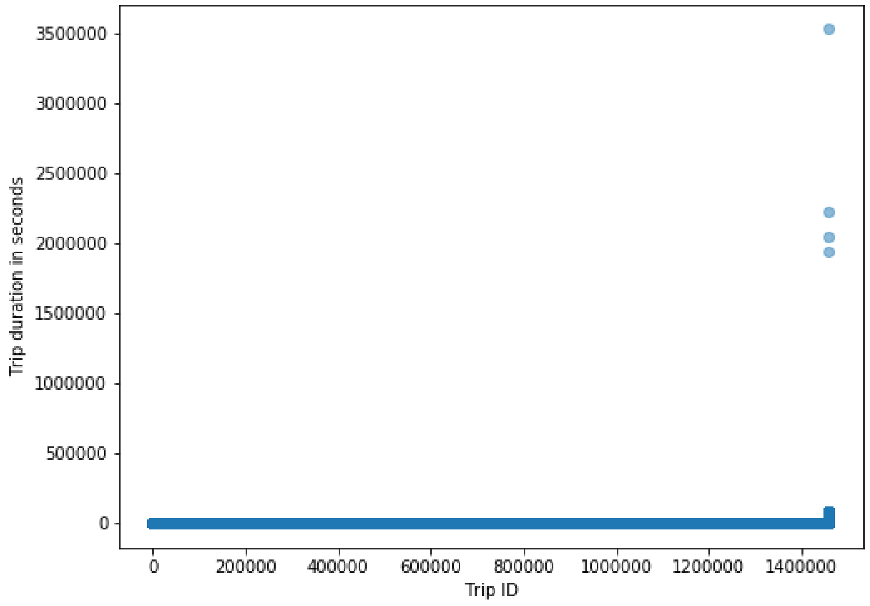
Getting to know about the dataset

```
#looking at target variable(trip duration)
print("Longest trip duration is {} secs : " .format(np.max(train['trip_duration'].values)))
print("Smallest trip duration is {} secs: ".format(np.min(train['trip_duration'].values)))
print("Average trip duration is {} secs".format(np.mean(train['trip_duration'].values)))
```

```
Longest trip duration is 3526282 secs :
Smallest trip duration is 1 secs:
Average trip duration is 959.4922729603659 secs
```

We can observe that the smallest trip is just 1 sec and longest trip is around 950 Hrs which are outliers. Lets see the Scatter plot of Trip_duration

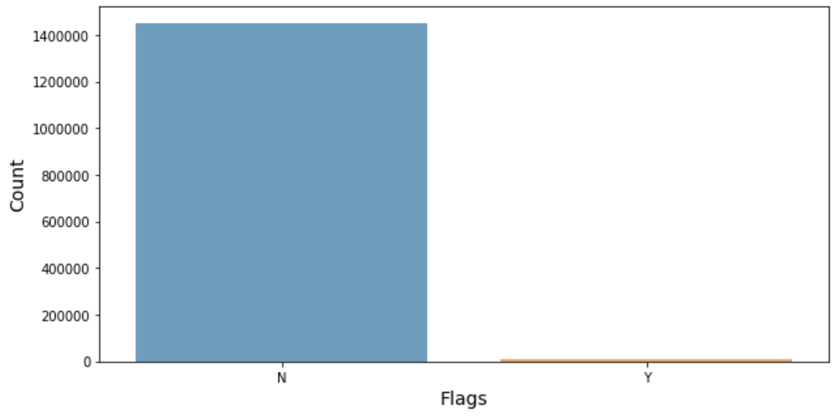
```
#Visualization is always better
f = plt.figure(figsize=(8,6))
plt.scatter(range(len(train['trip_duration'])), np.sort(train['trip_duration']), alpha=0.5)
plt.xlabel('Trip ID')
plt.ylabel('Trip duration in seconds')
plt.show()
```



Clearly there are 4 outliers. We need to remove those

Store_and_fwd_flag:

```
# Let's move to the store_and_fwd_flag column
flags = train['store_and_fwd_flag'].value_counts()

f = plt.figure(figsize=(10,5))
sns.barplot(flags.index, flags.values, alpha=0.7)
plt.xlabel('Flags', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.show()
```



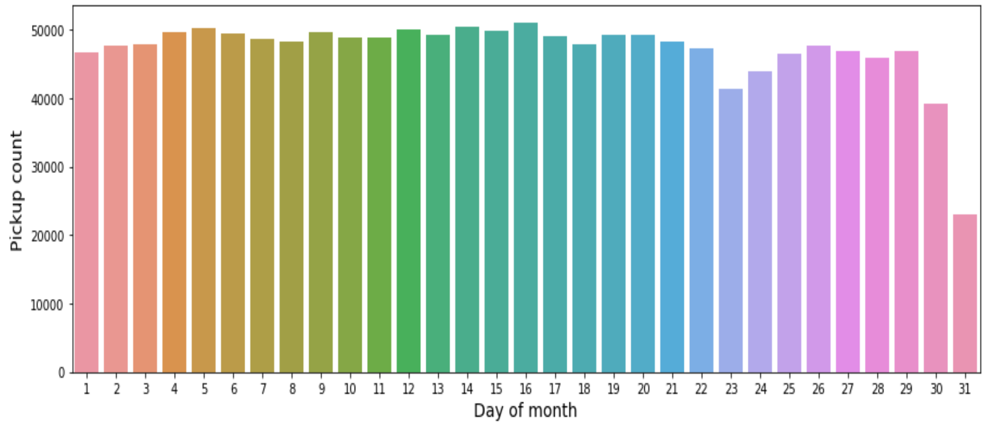
Seems like all details were immediately sent to vendors, very few were stored in device memory due to bad signal

Pickoff_datetime and Droppoff_datetime:

Lets convert the datetime to Month, Days, Weekdays and hours

- Checking at what date of a month the number of trips are more

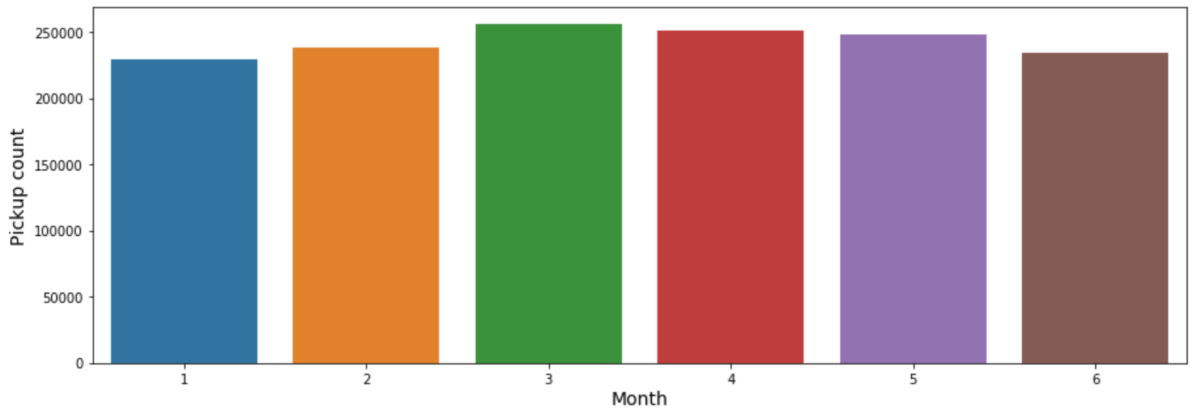
```
#Checking at what date of a month the number of trips are more
f = plt.figure(figsize=(15,5))
sns.countplot(x='pickup_day', data=train)
plt.xlabel('Day of month', fontsize=14)
plt.ylabel('Pickup count', fontsize=14)
plt.show()
```



All the days are having approximately same number of trips(We may think that 31st have less number of trips but there are only 3 months in 31st day from Jan to June

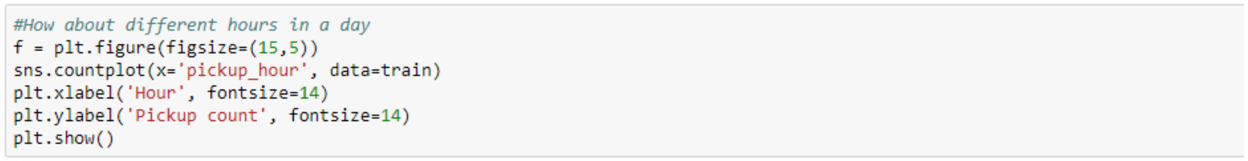
- Lets see if Month affects Number of trips

```
#Lets see if the month affects the number of trips
f = plt.figure(figsize=(15,5))
sns.countplot(x='pickup_month', data=train)
plt.xlabel('Month', fontsize=14)
plt.ylabel('Pickup count', fontsize=14)
plt.show()
```

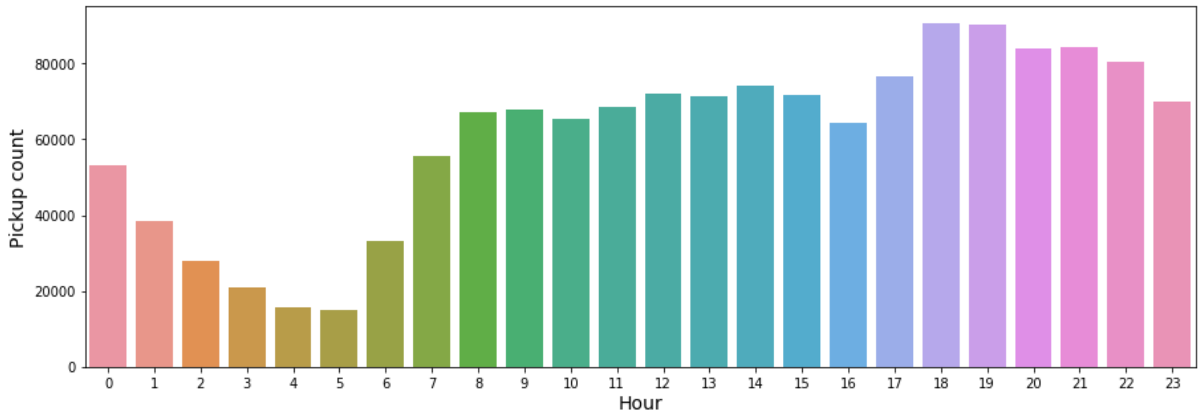


Seems like the month doesn't affect the number of trips much

- How about different hours in a day



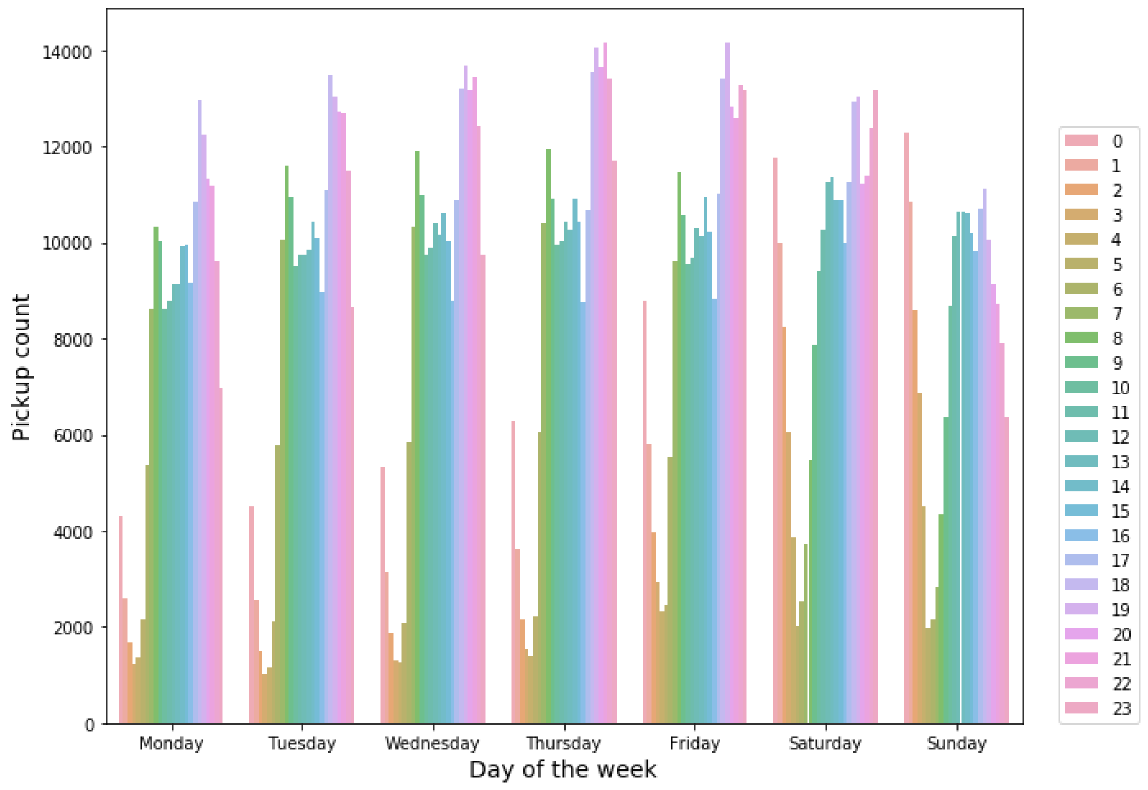
```
#How about different hours in a day
f = plt.figure(figsize=(15,5))
sns.countplot(x='pickup_hour', data=train)
plt.xlabel('Hour', fontsize=14)
plt.ylabel('Pickup count', fontsize=14)
plt.show()
```



It is evident that the number of trips after midnight are less and more trips occurred after 6:00 PM

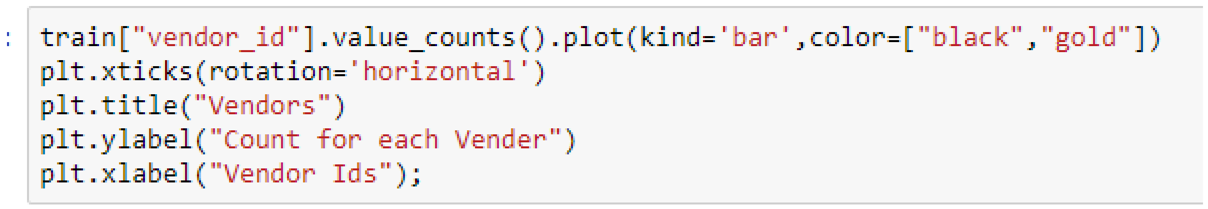
- Segmenting the days of weeks with hours

```
#How about segmenting days with hours
f = plt.figure(figsize=(10,8))
days = [i for i in range(7)]
sns.countplot(x='pickup_weekday', data=train, hue='pickup_hour', alpha=0.8)
plt.xlabel('Day of the week', fontsize=14)
plt.ylabel('Pickup count', fontsize=14)
plt.xticks(days, ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'))
plt.legend(loc=(1.04,0))
plt.show()
```

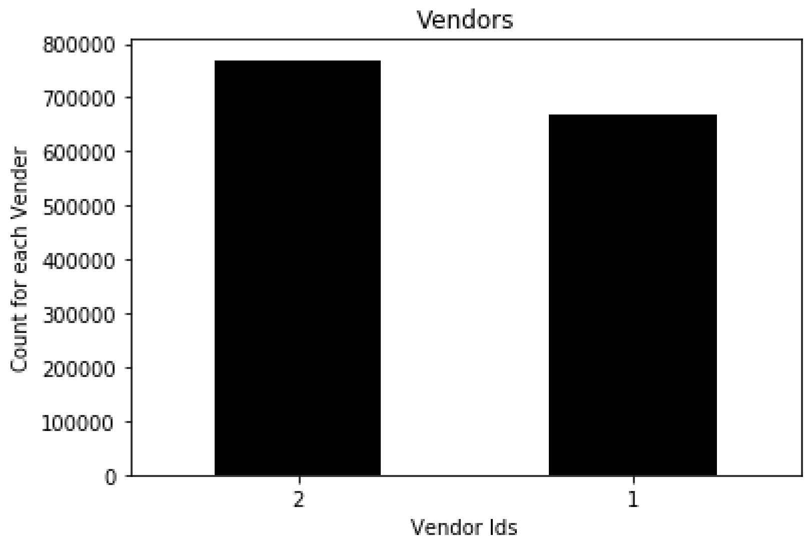


We can observe that early morning rides are more during Saturdays and Sundays (As it is a weekend may be people returning home after partying)

Vendor_id:

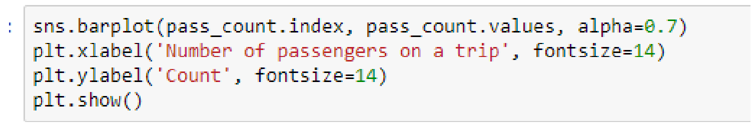


```
train["vendor_id"].value_counts().plot(kind='bar',color=["black","gold"])
plt.xticks(rotation='horizontal')
plt.title("Vendors")
plt.ylabel("Count for each Vender")
plt.xlabel("Vendor Ids");
```

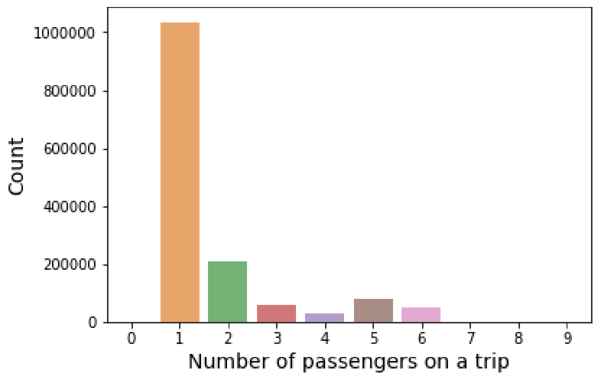


Seems like vendor 2 have more share of taxi rides in the dataset

Number of passengers:



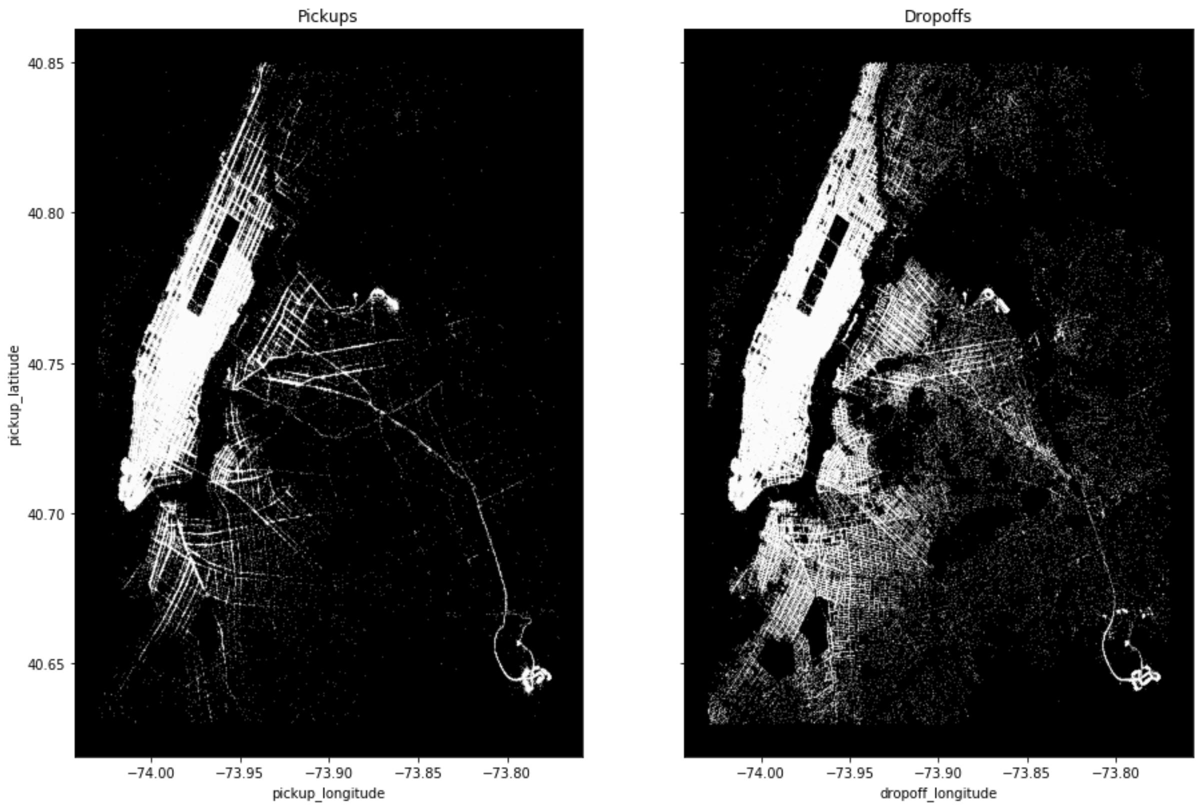
```
sns.barplot(pass_count.index, pass_count.values, alpha=0.7)
plt.xlabel('Number of passengers on a trip', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.show()
```



Insights:
1) Most of the trips have only 1 passenger and average being 1.66

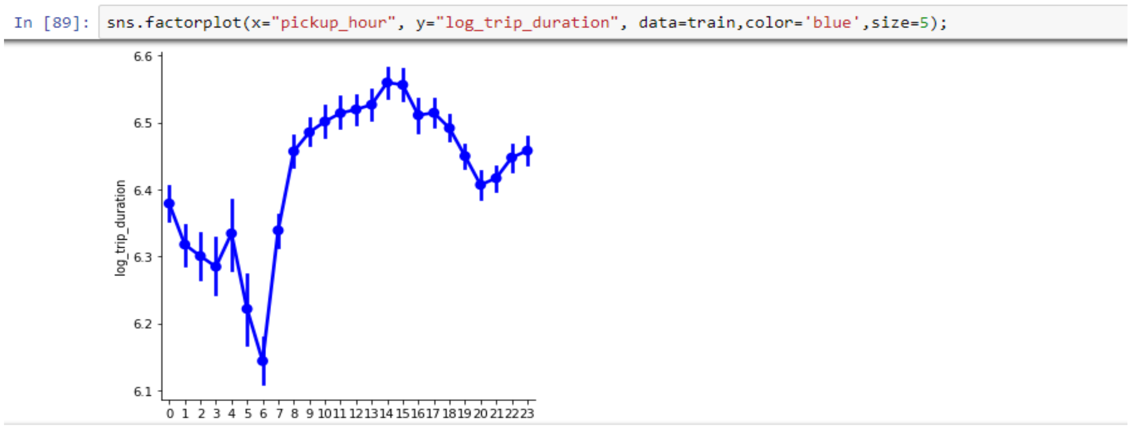
2) There are 60 trips with 0 passengers (which is interesting because no taxi ride can happen without passenger but a taxi may be called to a particular location and customer may be charged for it.)
3) There are 3 trips with 7 passenger count
4) There are 1 trip each for 9 and 8 number of passengers which is an outlier and can remove these rows

Pickup_latitude, Pickup_Longitude, Dropoff_latitude and Dropoff_longitude:



The pickups are more in Manhattan area and the drop offs are evenly distributed over the Manhattan and residential area (assumption)

Factor plot between Trip_duration and pickup_hour:

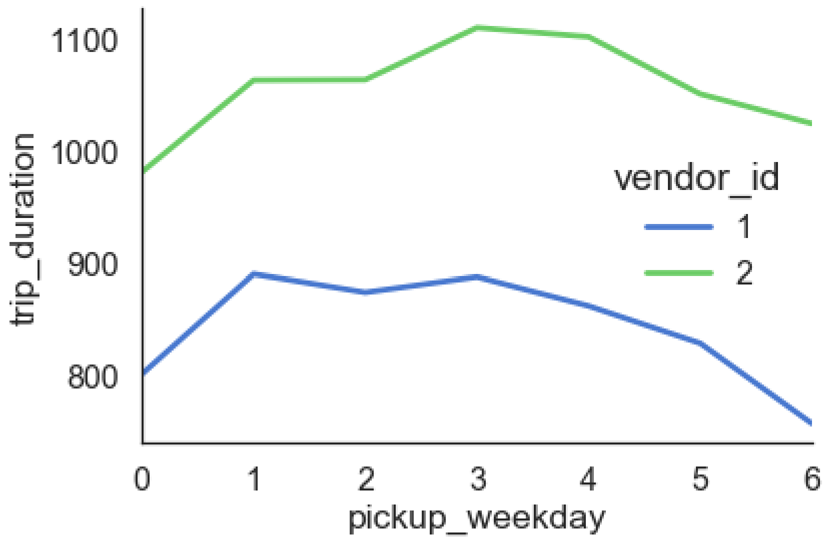


```
In [89]: sns.factorplot(x="pickup_hour", y="log_trip_duration", data=train,color='blue',size=5);
```

Insights:

- Lowest trip durations were in early mornings past midnight. May be people don't want to walk in dark
- Traffic starts increasing once its 8AM people starting to offices and reaches it's peak around 3 O' Clock in noon.
- And then starts decreasing slowly and again peaks after 8 till late nigh 11 O' Clock. May be because people going home after work hours.

Time Duration plot between vendors:



It is evident that the taxies which are with vendor 2 have more trip duration

3. Data Preprocessing:

This phase is the most important which you should spend more time as the cleaner and preprocessed your data is the more accurately the algorithm predicts
Trip_duration:

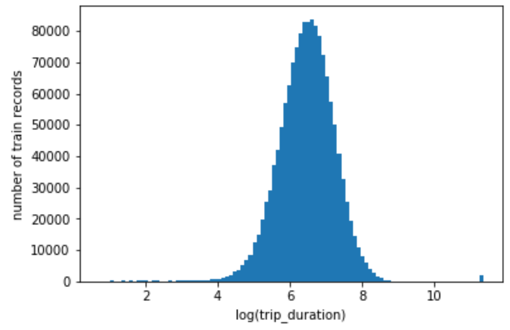
The target variable is right skewed. So, we need to remove or reduce the skewness for better results. There are four methods which we can use to reduce the skewness

1. Log Transform
2. Square root transform
3. Inverse square root transform
4. Box-Cox transform

As there are few 0 value records we added 1 to the trip_duration feature to avoid Inifinity or NaN’s values

```
#Deleting outliers
train = train.drop(train[(train['trip_duration']>1600000) & (train['trip_duration']<4000000)].index)
```

```
train['log_trip_duration'] = np.log(train['trip_duration'].values + 1)
plt.hist(train['log_trip_duration'].values, bins=100)
plt.xlabel('log(trip_duration)')
plt.ylabel('number of train records')
plt.show()
```



```
print("Skewness: %f" % train['log_trip_duration'].skew())
```

```
Skewness: -0.268602
```

Feature Extraction:

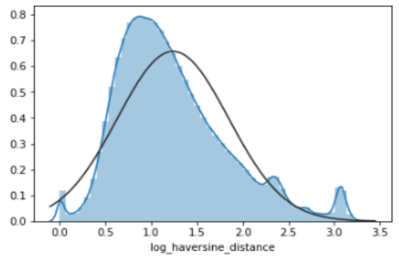
Calculating Distance and Speeds:

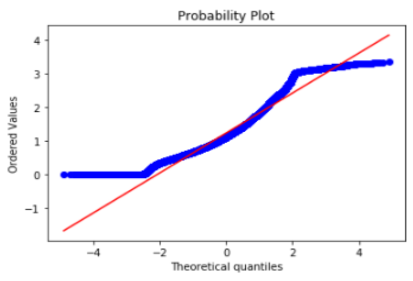
1. The haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitudes.
2. Degrees of latitude are parallel so the distance between each degree remains almost constant.Each degree of latitude is approximately 69 miles (111 kilometers) .

Removing the skewness of new variable haversine_distance:

```
train['log_haversine_distance'] = np.log1p(train['haversine_distance'])
```

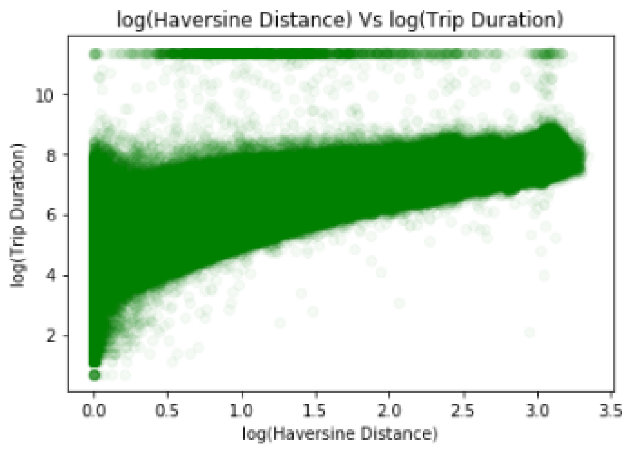
```
sns.distplot(train['log_haversine_distance'], fit = norm);
fig = plt.figure()
res = stats.probplot(train['log_haversine_distance'], plot=plt)
plt.show()
```





Scatter plot between log_trip_duration and log_haversine_distance:

```
plt.scatter(train.log_haversine_distance,train.log_trip_duration,color="green",alpha=0.04)
plt.ylabel("log(Trip Duration)")
plt.xlabel("log(Haversine Distance)")
plt.title("log(Haversine Distance) Vs log(Trip Duration)");
```

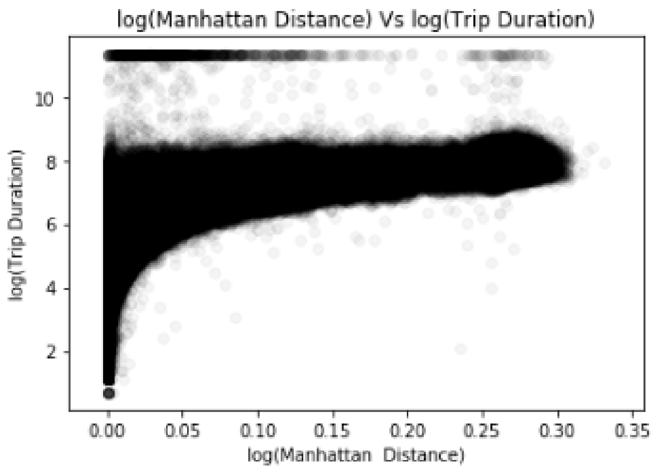


A linear relationship is observed but there is a data capping

Similarly calculating the Euclidean distance and Manhattan distance

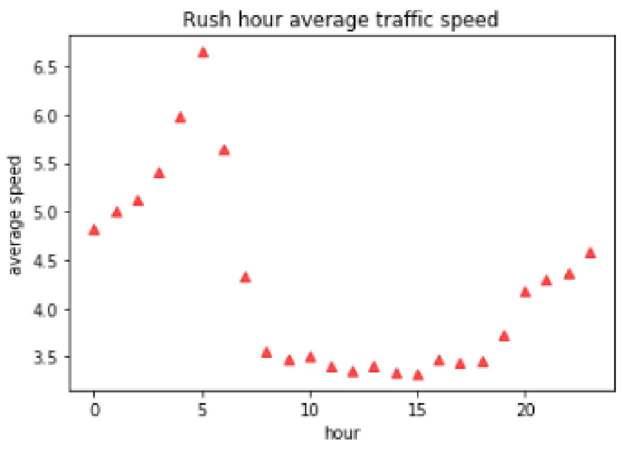
Scatter plot between log_trip_duration and log_mnhattan_distance:

```
plt.scatter(train.log_manhattan_distance,train.log_trip_duration,color="black",alpha=0.04)
plt.ylabel("log(Trip Duration)")
plt.xlabel("log(Manhattan  Distance)")
plt.title("log(Manhattan Distance) Vs log(Trip Duration)");
```



Lets see how the average speed is varying with time in a day:

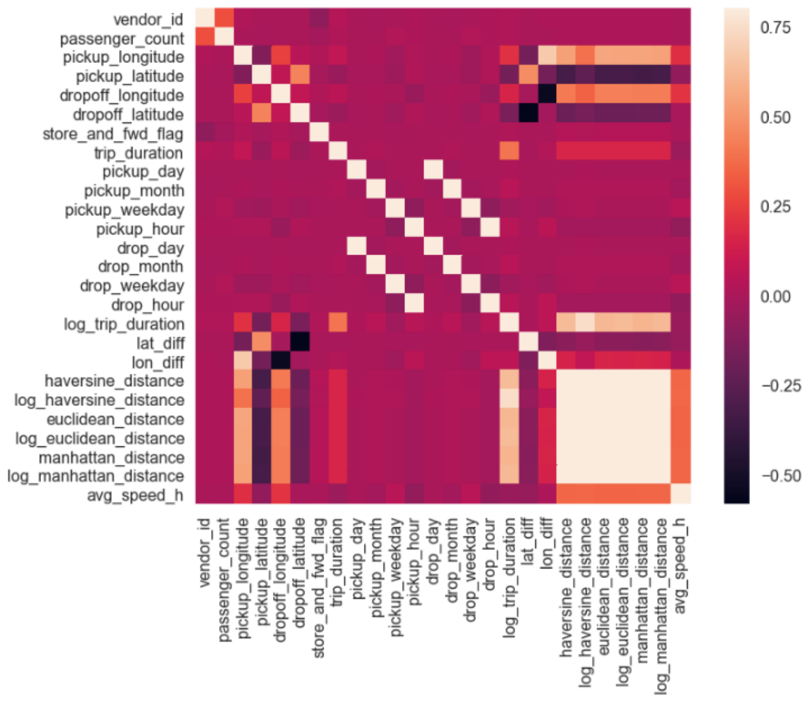
```
plt.plot(train.groupby('pickup_hour').mean()['avg_speed_h'], '^', lw=2, alpha=0.7,color='red')
plt.xlabel('hour')
plt.ylabel('average speed')
plt.title('Rush hour average traffic speed')
plt.show()
```



We can observe that average speed is high after midnight and less during 8:00 AM to 8:00 PM

Correlation matrix:

```
#correlation matrix
import matplotlib.pyplot as plt
import seaborn as sns
corrmat = train.corr()
f, ax = plt.subplots(figsize=(12, 9))
sns.heatmap(corrmat, vmax=.8, square=True);
plt.show()
```



For further analysis we can drop the following variables as we have the log transform of those variables in the dataset and we are storing the target variable in y_train1 variable

```
y_train1 = train_temp['log_trip_duration']
```

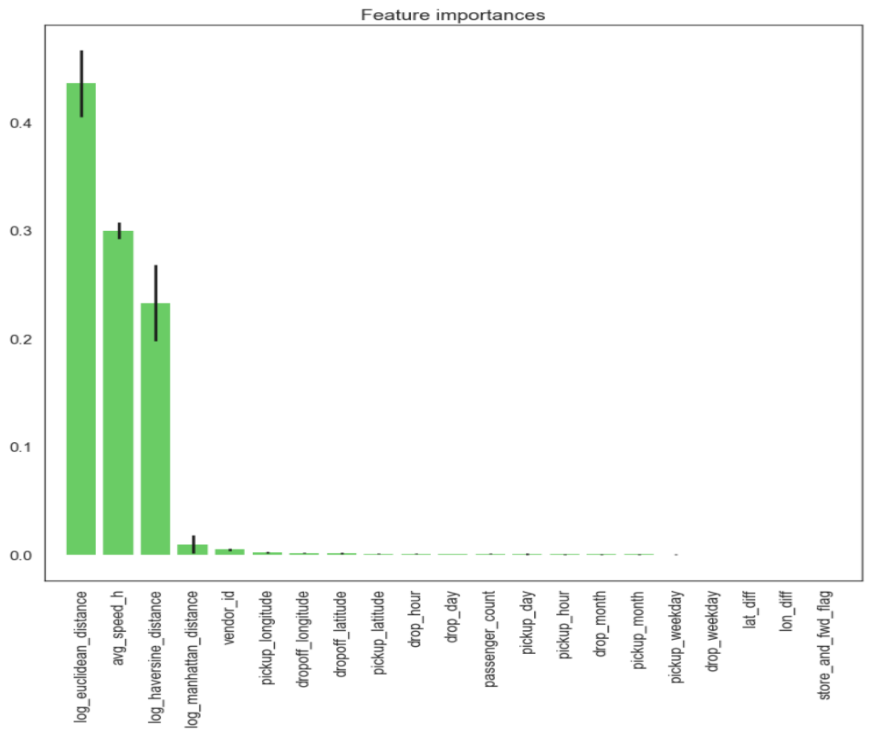
```
x_train1 = train_temp.drop(['log_trip_duration','trip_duration','haversine_distance','manhattan_distance','euclidean_distance']
                            , axis =1)
```

Feature Importance:
The feature_importance_ function in Random Forest algorithm gives us the scores of independent variables which contribute towards predicting the target variable.

The Nodes which are near to the root node in a tree are more contributing towards predicting the target variable

```
clf_rf_5 = RandomForestRegressor()
clr_rf_5 = clf_rf_5.fit(x_train1,y_train1)
importances = clr_rf_5.feature_importances_
std = np.std([tree.feature_importances_ for tree in clf_rf_5.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(x_train.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

# Plot the feature importances of the forest

plt.figure(1, figsize=(14, 13))
plt.title("Feature importances")
plt.bar(range(x_train1.shape[1]), importances[indices],
       color="g", yerr=std[indices], align="center")
plt.xticks(range(x_train1.shape[1]), x_train1.columns[indices],rotation=90)
plt.xlim([-1, x_train1.shape[1]])
plt.show()
```

```
Feature ranking:
1. feature 18 (0.436374)
2. feature 20 (0.300013)
3. feature 17 (0.233254)
4. feature 19 (0.010023)
5. feature 0 (0.005036)
6. feature 2 (0.002284)
7. feature 4 (0.001871)
8. feature 5 (0.001720)
```



From the above plot it is evident that log_euclidean_distance, avg_speed_h and log_haversine_distance contributes the most in predicting the target variable.

Dropping Features:

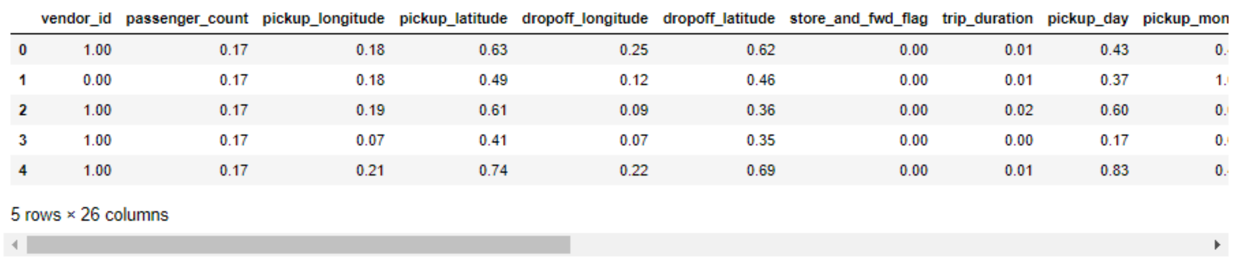
```
x_train1 = train_temp.drop(['log_trip_duration','trip_duration','haversine_distance','manhattan_distance','euclidean_distance',
                            'drop_weekday','drop_day','drop_hour','drop_month'], axis =1)
```

- ➢ dropoff_datetime column would affect the prediction because during the training process the algorithm can learn that the difference between pickup_datetime and dropoff_datetime will determine the target variable
- ➢ Haversine_distance, Manhattan_distance, Euclidean_distance columns need to be dropped as there are log transform of the features mentioned above
- ➢ Obviously, we need to drop the target variable which is Trip_duration and log transform of target variable log_trip_duration

Normalizing the data:

The original dataset which we used till now the values are very much wide spread as the log distances are small numbers and other columns have high numbers so we need to normalize the data so that all columns fall between 0 to 1

```
from sklearn import preprocessing
min_max_scaler = preprocessing.MinMaxScaler()
np_scaled = min_max_scaler.fit_transform(train1)
train_norm = pd.DataFrame(np_scaled, columns = train1.columns)
train_norm.head(5)
```



	vendor_id	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd_flag	trip_duration	pickup_day	pickup_mon
0	1.00	0.17	0.18	0.63	0.25	0.62	0.00	0.01	0.43	0.
1	0.00	0.17	0.18	0.49	0.12	0.46	0.00	0.01	0.37	1.
2	1.00	0.17	0.19	0.61	0.09	0.36	0.00	0.02	0.60	0.
3	1.00	0.17	0.07	0.41	0.07	0.35	0.00	0.00	0.17	0.
4	1.00	0.17	0.21	0.74	0.22	0.69	0.00	0.01	0.83	0.

5 rows × 26 columns

Splitting the data into Train and Validation set:

Because of computational issue I took just 100k values from the original dataset
We need a dataset to train and a validation set to evaluate our models

```
#Dividing into train and test
import numpy as np
import xgboost as xgb
from sklearn.cross_validation import train_test_split
x_train, x_val, y_train, y_val = train_test_split(train1[feature_names_1].values, y, test_size=0.2, random_state=42)
dtrain = xgb.DMatrix(x_train, label=y_train)
dvalid = xgb.DMatrix(x_val, label=y_val)
watchlist = [(dtrain, 'train'), (dvalid, 'valid')]

print(len(x_train))
print(len(x_val))
print(y_train.shape)
print(y_val.shape)
```

```
80000
20000
(80000,)
(20000,)
```

Used Scikitlearn train_test_split package to split the data set in to train and validation set x_train = independent variables

y_train = dependent variable(log_trip_duration)

x_val = independent variables in validation dataset

y_val = independent variables in validation dataset

Modelling:
Since the data pre-processing is done we can proceed with the modelling.
I plan to use three regression modelling techniques which are

1. XGBoost
2. Random Forest Regressor
3. Ridge Regression
4. Lasso Regression
5. Averaging Regressor

XGBoost:

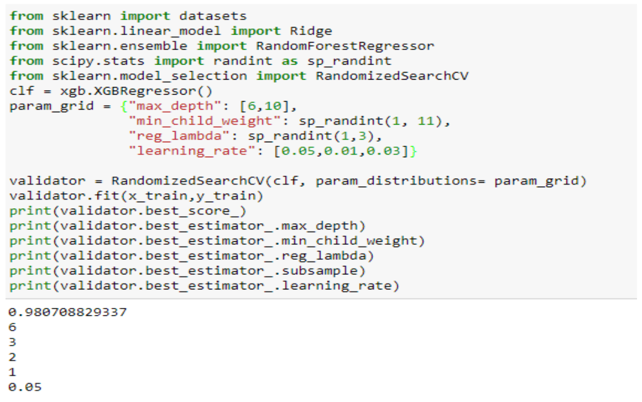
XGBoost is an advanced implementation of Gradient Boosting algorithms.

People are obsessed over XGBoost because of following advantages

- **Regularization:**
  - ➢ Standard GBM has no regularization like XGBoost, therefore it helps reducing overfitting
  - ➢ In fact, XGBoost is also known as 'regularized boosting' technique
- **Parallel Processing:**
  - XGBoost implements parallel processing and is very fast as compared to GBM
- **Handling Missing Values:**
  - XGBoost can handle missing values

Hyperparameter tuning:

Firstly the algorithm must be fine tuned to get the optimal values of hyper-parameters. For this I have used RandomizedSearchCV package from Scikitlearn which takes the range of hyperparameters and gives out the best combination of parameters for which algorithm performed better



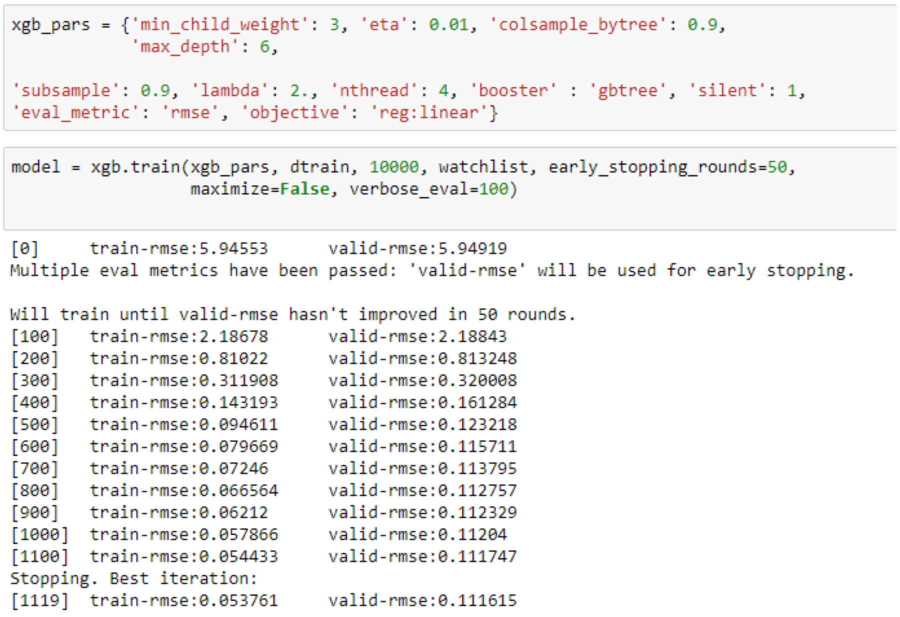
```
from sklearn import datasets
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor
from scipy.stats import randint as sp_randint
from sklearn.model_selection import RandomizedSearchCV
clf = xgb.XGBRegressor()
param_grid = {"max_depth": [6,10],
              "min_child_weight": sp_randint(1, 11),
              "reg_lambda": sp_randint(1,3),
              "learning_rate": [0.05,0.01,0.03]}

validator = RandomizedSearchCV(clf, param_distributions= param_grid)
validator.fit(x_train,y_train)
print(validator.best_score_)
print(validator.best_estimator_.max_depth)
print(validator.best_estimator_.min_child_weight)
print(validator.best_estimator_.reg_lambda)
print(validator.best_estimator_.subsample)
print(validator.best_estimator_.learning_rate)
```

```
0.980708829337
6
3
2
1
0.05
```

Here reg_lambda parameter imposes either L1 regularization or L2 regularization to prevent from overfitting.

Training XGBoost on training set and evaluating on validation set:

Here I used 50 early stopping rounds. Which is if there is no improvement of validation RMSE after 50 iterations the algorithm will stop

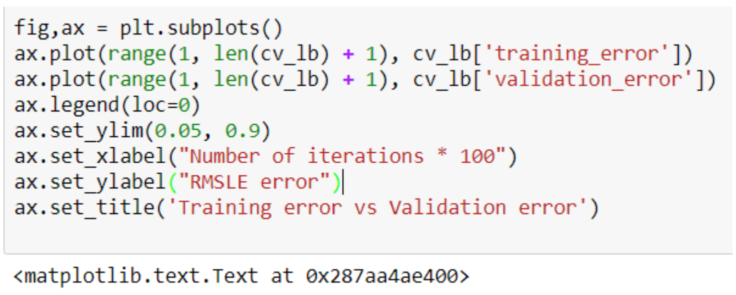


```
xgb_pars = {'min_child_weight': 3, 'eta': 0.01, 'colsample_bytree': 0.9,
            'max_depth': 6,

'subsample': 0.9, 'lambda': 2., 'nthread': 4, 'booster' : 'gbtree', 'silent': 1,
'eval_metric': 'rmse', 'objective': 'reg:linear'}
```

```
model = xgb.train(xgb_pars, dtrain, 10000, watchlist, early_stopping_rounds=50,
                  maximize=False, verbose_eval=100)
```

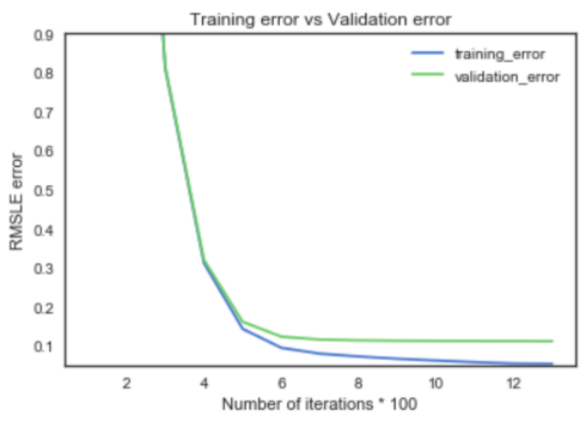
```
[0]     train-rmse:5.94553      valid-rmse:5.94919
Multiple eval metrics have been passed: 'valid-rmse' will be used for early stopping.

Will train until valid-rmse hasn't improved in 50 rounds.
[100]   train-rmse:2.18678      valid-rmse:2.18843
[200]   train-rmse:0.81022      valid-rmse:0.813248
[300]   train-rmse:0.311908     valid-rmse:0.320008
[400]   train-rmse:0.143193     valid-rmse:0.161284
[500]   train-rmse:0.094611     valid-rmse:0.123218
[600]   train-rmse:0.079669     valid-rmse:0.115711
[700]   train-rmse:0.07246      valid-rmse:0.113795
[800]   train-rmse:0.066564     valid-rmse:0.112757
[900]   train-rmse:0.06212      valid-rmse:0.112329
[1000]  train-rmse:0.057866     valid-rmse:0.11204
[1100]  train-rmse:0.054433     valid-rmse:0.111747
Stopping. Best iteration:
[1119]  train-rmse:0.053761     valid-rmse:0.111615
```

We can observe that there is no overfitting as the training and validation RMSE are constantly decreasing and due to the L2 penalty we applied on the parameters

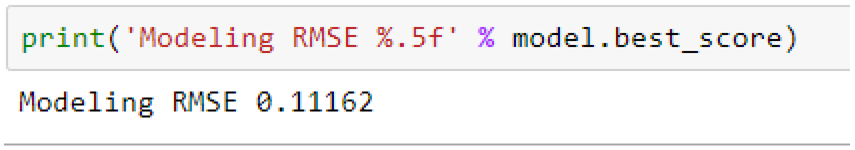


```
fig,ax = plt.subplots()
ax.plot(range(1, len(cv_lb) + 1), cv_lb['training_error'])
ax.plot(range(1, len(cv_lb) + 1), cv_lb['validation_error'])
ax.legend(loc=0)
ax.set_ylim(0.05, 0.9)
ax.set_xlabel("Number of iterations * 100")
ax.set_ylabel("RMSLE error")
ax.set_title('Training error vs Validation error')
```

```
<matplotlib.text.Text at 0x287aa4ae400>
```



The above claim is proved visually as we can see both training and validation RMSE are decreasing



```
print('Modeling RMSE %.5f' % model.best_score)
```

```
Modeling RMSE 0.11162
```

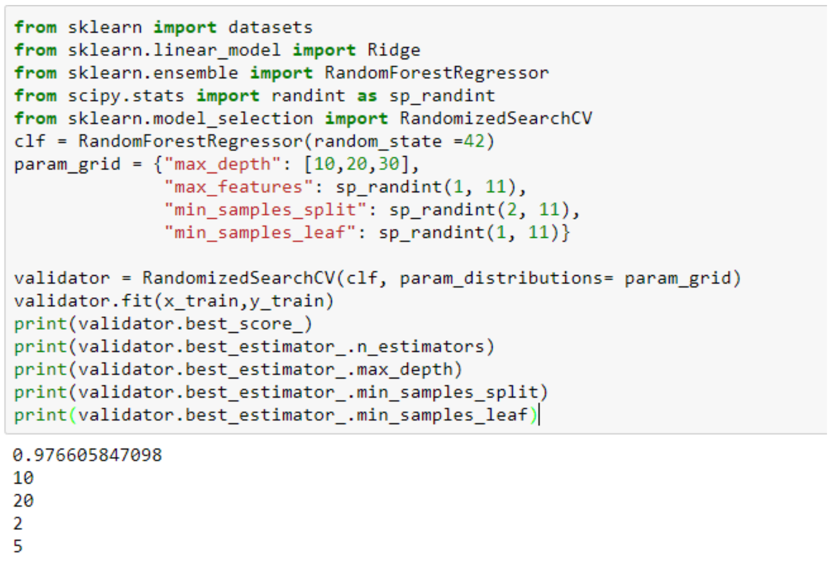
Finally the XGBoost RMSLE is 0.11162

Random Forest Regressor:

This is one of the powerful regression models available. It uses bagging technique

Hyperparameter Tuning:

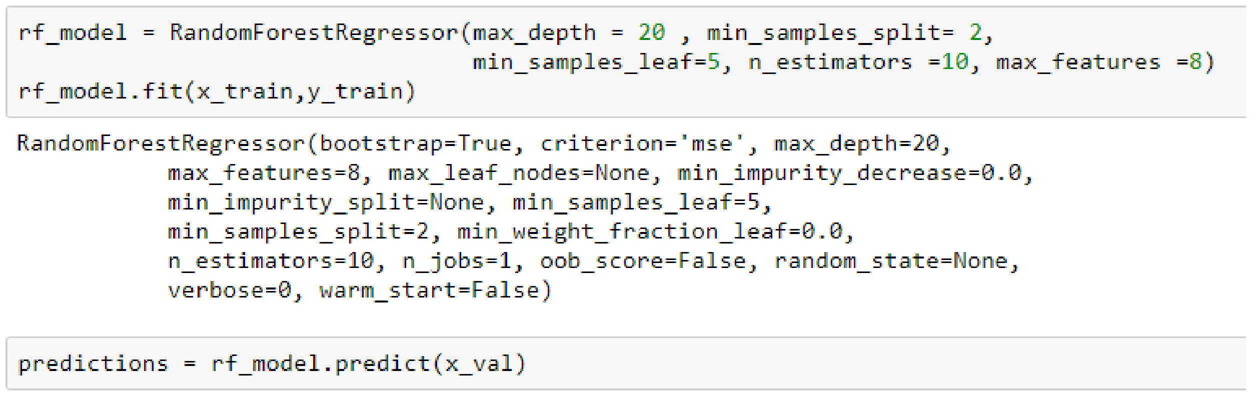
the algorithm must be fine tuned to get the optimal values of hyper-parameters. For this I have used RandomizedSearchCV package from Scikitlearn which takes the range of hyperparameters and gives out the best combination of parameters for which algorithm performed better



```
from sklearn import datasets
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor
from scipy.stats import randint as sp_randint
from sklearn.model_selection import RandomizedSearchCV
clf = RandomForestRegressor(random_state =42)
param_grid = {"max_depth": [10,20,30],
              "max_features": sp_randint(1, 11),
              "min_samples_split": sp_randint(2, 11),
              "min_samples_leaf": sp_randint(1, 11)}

validator = RandomizedSearchCV(clf, param_distributions= param_grid)
validator.fit(x_train,y_train)
print(validator.best_score_)
print(validator.best_estimator_.n_estimators)
print(validator.best_estimator_.max_depth)
print(validator.best_estimator_.min_samples_split)
print(validator.best_estimator_.min_samples_leaf)
```

```
0.976605847098
10
20
2
5
```

Training Random Forest on training set and evaluating on validation set:

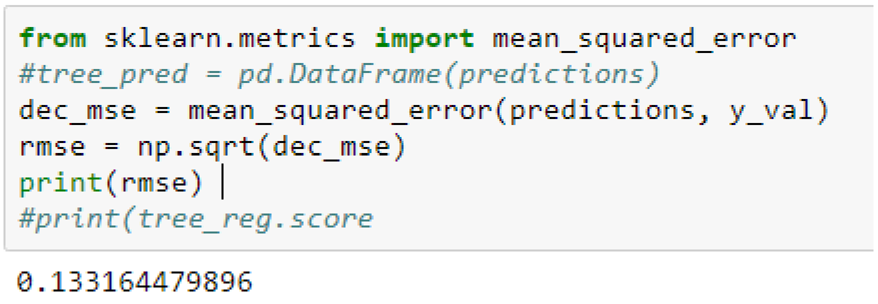


```
rf_model = RandomForestRegressor(max_depth = 20 , min_samples_split= 2,
                                 min_samples_leaf=5, n_estimators =10, max_features =8)
rf_model.fit(x_train,y_train)
```

```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=20,
           max_features=8, max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=5,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
           verbose=0, warm_start=False)
```

```
predictions = rf_model.predict(x_val)
```

Evaluating the model using RMSE



```
from sklearn.metrics import mean_squared_error
#tree_pred = pd.DataFrame(predictions)
dec_mse = mean_squared_error(predictions, y_val)
rmse = np.sqrt(dec_mse)
print(rmse)
#print(tree_reg.score
```

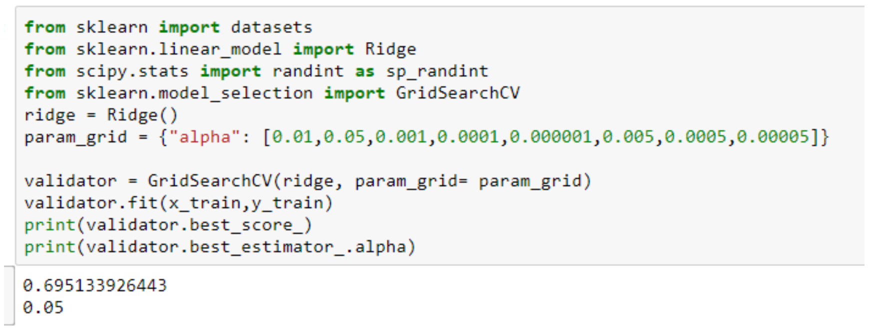
```
0.133164479896
```

Random Forest RMSLE is 0.13312

Ridge Regression:

Implemented Ridge regression as it is uses L2 regularization to reduce overfitting

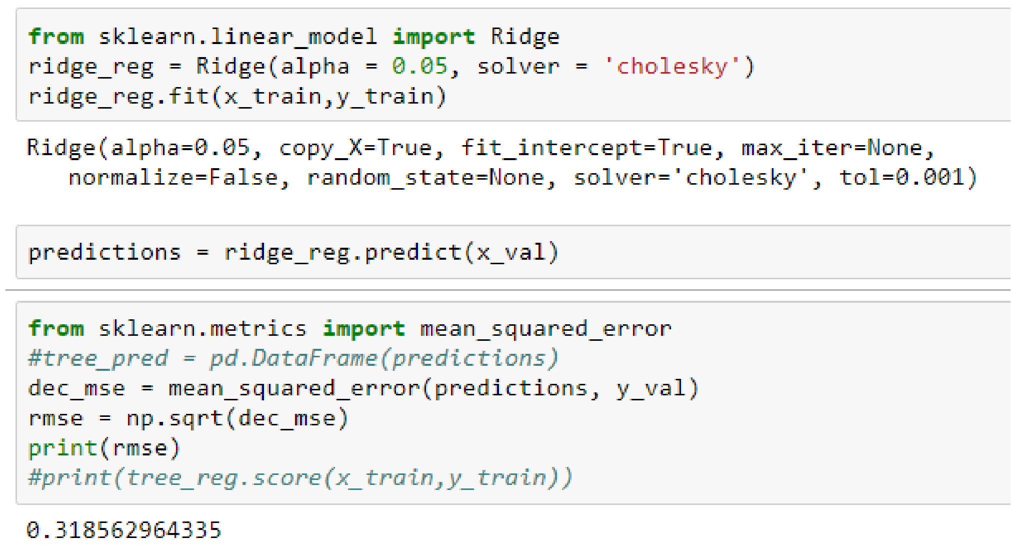
Hyperparameter tuning:



```
from sklearn import datasets
from sklearn.linear_model import Ridge
from scipy.stats import randint as sp_randint
from sklearn.model_selection import GridSearchCV
ridge = Ridge()
param_grid = {"alpha": [0.01,0.05,0.001,0.0001,0.000001,0.005,0.0005,0.00005]}

validator = GridSearchCV(ridge, param_grid= param_grid)
validator.fit(x_train,y_train)
print(validator.best_score_)
print(validator.best_estimator_.alpha)
```

```
0.695133926443
0.05
```

Training and predicting the RMSLE on validation set:



```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha = 0.05, solver = 'cholesky')
ridge_reg.fit(x_train,y_train)
```

```
Ridge(alpha=0.05, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='cholesky', tol=0.001)
```

```
predictions = ridge_reg.predict(x_val)
```

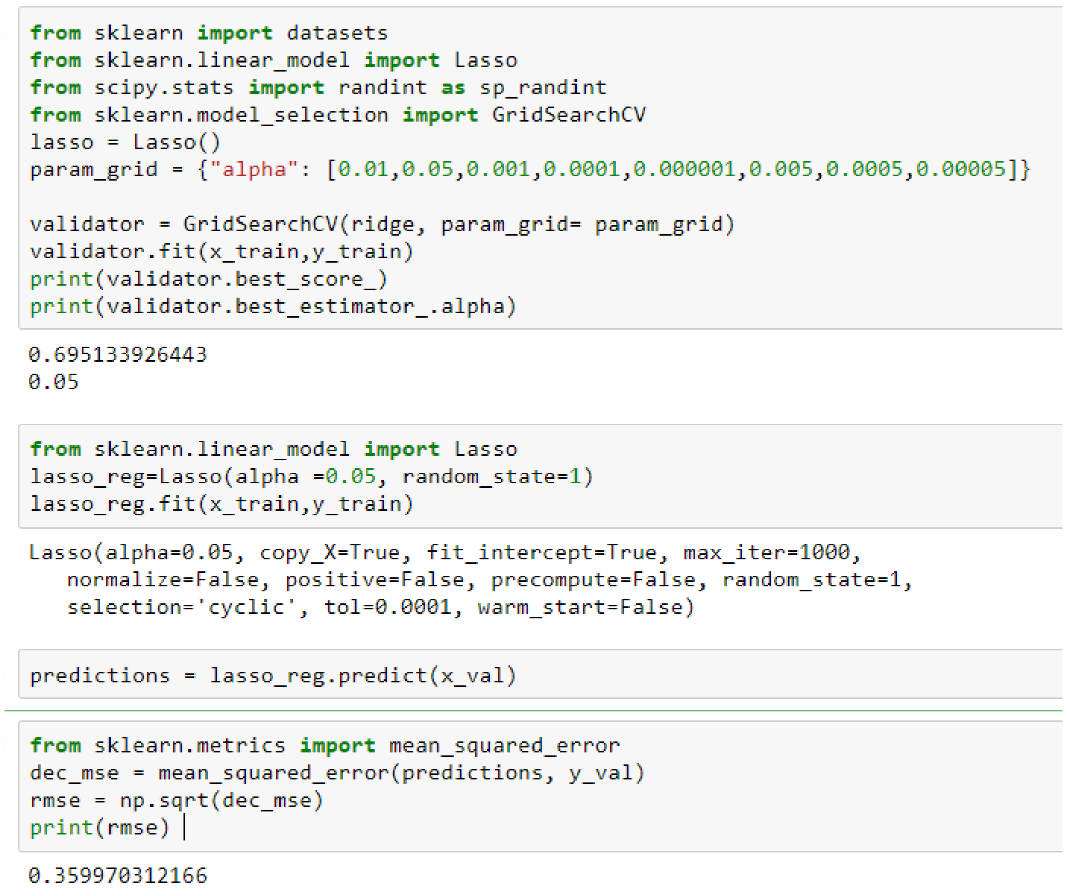
```
from sklearn.metrics import mean_squared_error
#tree_pred = pd.DataFrame(predictions)
dec_mse = mean_squared_error(predictions, y_val)
rmse = np.sqrt(dec_mse)
print(rmse)
#print(tree_reg.score(x_train,y_train))
```

```
0.318562964335
```

Ridge Regression RMSLE is 0.31856

Lasso Regression:

Hyperparameter tuning and evaluating the model



```
from sklearn import datasets
from sklearn.linear_model import Lasso
from scipy.stats import randint as sp_randint
from sklearn.model_selection import GridSearchCV
lasso = Lasso()
param_grid = {"alpha": [0.01,0.05,0.001,0.0001,0.000001,0.005,0.0005,0.00005]}

validator = GridSearchCV(ridge, param_grid= param_grid)
validator.fit(x_train,y_train)
print(validator.best_score_)
print(validator.best_estimator_.alpha)
```

```
0.695133926443
0.05
```

```
from sklearn.linear_model import Lasso
lasso_reg=Lasso(alpha =0.05, random_state=1)
lasso_reg.fit(x_train,y_train)
```

```
Lasso(alpha=0.05, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=1,
   selection='cyclic', tol=0.0001, warm_start=False)
```

```
predictions = lasso_reg.predict(x_val)
```

```
from sklearn.metrics import mean_squared_error
dec_mse = mean_squared_error(predictions, y_val)
rmse = np.sqrt(dec_mse)
print(rmse)
```

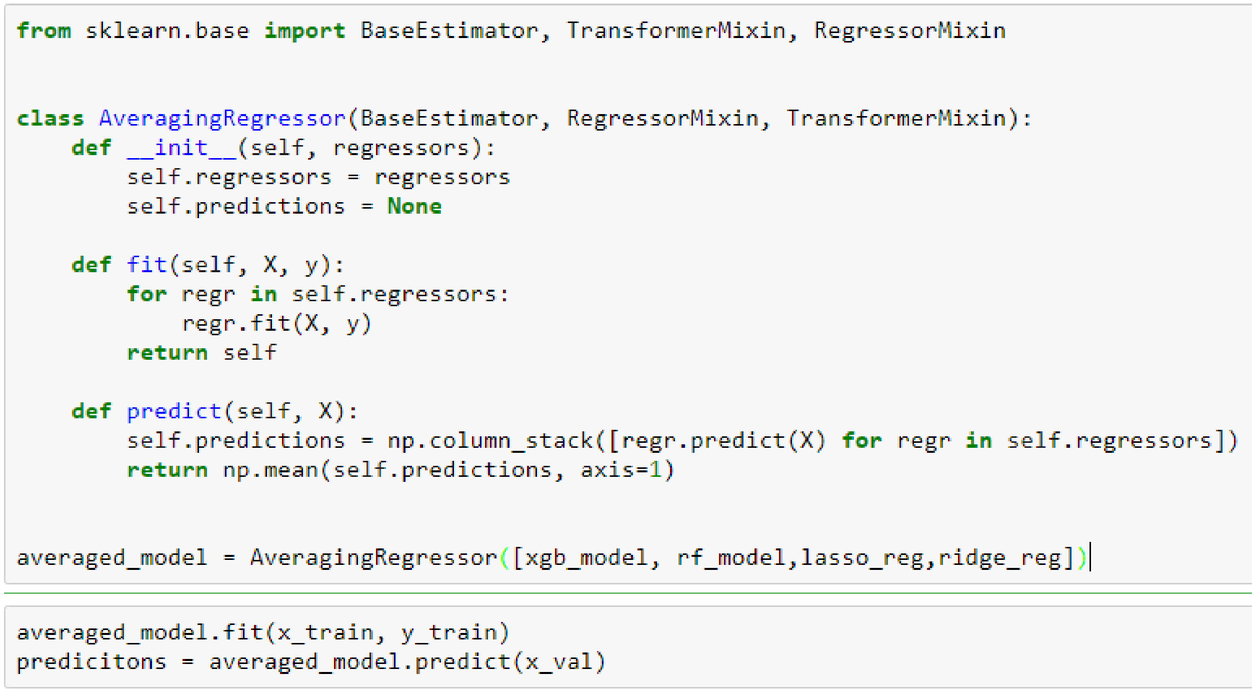
```
0.359970312166
```

Lasso regression RMSLE is about 0.35998

Ensemble Learning:

Ensemble learning is often used to further improve the model efficiency in predicting the target variable given set of features.

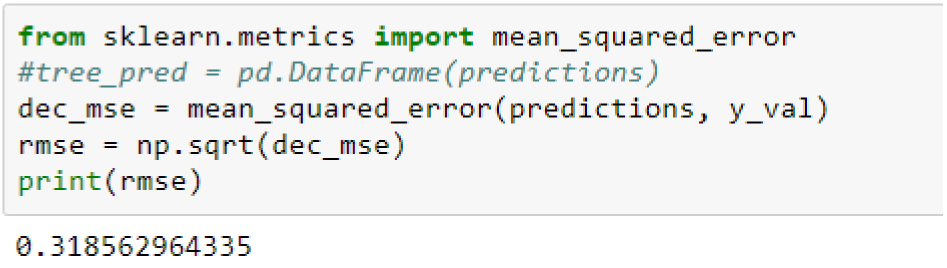
I have used Averaging Regressor where it takes the outcome of different regressors and average the result



```
from sklearn.base import BaseEstimator, TransformerMixin, RegressorMixin


class AveragingRegressor(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self, regressors):
        self.regressors = regressors
        self.predictions = None

    def fit(self, X, y):
        for regr in self.regressors:
            regr.fit(X, y)
        return self

    def predict(self, X):
        self.predictions = np.column_stack([regr.predict(X) for regr in self.regressors])
        return np.mean(self.predictions, axis=1)


averaged_model = AveragingRegressor([xgb_model, rf_model,lasso_reg,ridge_reg])
```

```
averaged_model.fit(x_train, y_train)
prediciton = averaged_model.predict(x_val)
```

Evaluating the model on validation set:



```
from sklearn.metrics import mean_squared_error
#tree_pred = pd.DataFrame(predictions)
dec_mse = mean_squared_error(predictions, y_val)
rmse = np.sqrt(dec_mse)
print(rmse)
```

```
0.318562964335
```

The RMSLE for averaged regressor is about 0.31856

Evaluation of models:

Model	RMSLE
XGBoost	0.11162
Random Forest Regressor	0.13312
Ridge Regression	0.31856
Lasso Regression	0.35988
Averaging Models	0.13237

- Among all the models XGBoost performed exceptionally well when compared to other algorithms
- It is not a rule that Ensemble learning would give better results sometimes it may perform worse than independent models

Further Improvements:

- The dataset given doesn’t contain the sufficient features to predict the target variable efficiently because:
  - I have observed different trip durations for the same distance on different days this might be due to different factors such as
    1. There might be a traffic jam on that route and on that date
    2. The weather might not be suitable to travel, it might be raining or snowing
    3. The dataset contained only the coordinates of pickup and drop-off. It doesn’t tell us the route the taxi used to get to the destination. (The Taxi may have used the longer route instead of shorter route)