Udacity's Machine Learning Engineer Nanodegreee
Capstone Project Program
New York City Taxi Trip Duration

Luciano Falqueto Santana

January 15, 2019

Contents

1 Definition

1.1 Project Overview

The final goal for the Capstone Project is to solve one Kaggle Challenge, called **New York City Taxi Trip Duration**. The challenge proposal is to build a model capable of predicting the total duration of taxi trip in New York City using historic data provided in the Challenge page. The dataset includes pickup and drop-off time, pickup and drop-off coordinates, trip duration and number of passengers among other variables.

The challenge of solving similar problems using machine learning methods is not new [5][4], but the amount of data available and it's variety, resolution and precision is increasing now that smartphones provided with GPS and real-time tracking is available.

The dataset provided is already cleaned, allowing to direct focusing on the problem itself, the next step are analyze all the provided fields, make the feature engineering for the dataset, select the most relevant fields after preprocessing them, build the model and evaluate it.

1.2 Problem Statement

The competition dataset is based on the 2016 NYC Yellow Cab trip record data made available in Big Query on Google Cloud Platform. The data was originally published by the NYC Taxi and Limousine Commission (TLC). The data was originally published by the NYC Taxi and Limousine Commission (TLC). [2] The data was sampled and cleaned for the purposes of the challenge.

The challenge consists in finding the trip duration for taxis in New York City using only a sample of historical data from previous trips and the characteristics of those trips.

To solve this problem we will be using some Machine Learing methods called Regression Methods, but before training our models, we will have to understand the data, clean the data

removing outliers and spurious data, train and validate our models and finally predict the trip duration for the challenge's evaluation dataset.

1.3 Metrics

The same evaluation metric used to rank the competitors in the Kaggle competition will be used here, the metric will be the Root Mean Square Logarithmic Error (RMSLE), given by:

$$\epsilon = \frac{1}{n^s}\sqrt{\sum_{i=1}^{n}(\log(p_i+1) - \log(a_i+1))^2} \tag{1}$$

Where:

- ϵ is the RMSLE value (score);
- n is the total number of observations in the (public/private) data set;
- p_i is your prediction of trip duration;
- a_i is the actual trip duration for i;
- $log(x)$ is the natural logarithm of x.

The Score Function is implememented as follows:

```
def kaggle_score(y_true_exp, y_pred_exp):
            y_pred_exp = np.exp(y_pred_exp) - 1
            y_true_exp = np.exp(y_true_exp) - 1
            e_log_square = np.square( np.log(y_pred_exp + 1) - np.log(y_true_exp + 1))
            score = np.sqrt((1/len(y_true_exp)) * np.sum(e_log_square))
            return score
```

2 Analysis

2.1 The Dataset

Two datasets are provided by the Challenge, the **Train** dataset with 1,458,644 trip records and the **Test** dataset with 625,134 trip records. Both datasets are in CSV format and were originally published by the NYC Taxi and Limousine Commission (TLC)[2].

The data fields, as seen in the challenge's website:

- `id` - a unique identifier for each trip
- `vendor_id` - a code indicating the provider associated with the trip record
- `pickup_datetime` - date and time when the meter was engaged
- `dropoff_datetime` - date and time when the meter was disengaged
- `passenger_count` - the number of passengers in the vehicle (driver entered value)

- `pickup_longitude` - the longitude where the meter was engaged
- `pickup_latitude` - the latitude where the meter was engaged
- `dropoff_longitude` - the longitude where the meter was disengaged
- `dropoff_latitude` - the latitude where the meter was disengaged
- `store_and_fwd_flag` - This flag indicates whether the trip record was held in vehicle memory before sending to the vendor because the vehicle did not have a connection to the server - Y=store and forward; N=not a store and forward trip
- `trip_duration` - duration of the trip in seconds

The Target Variable:

The challenge objective is to find values for trip_duration for each trip in the **Test** dataset with a model trained using the information known from the **Train** dataset.

In more technical words as the origin data is already cleaned and prepared for processing, the first step to be done is the Feature Engineering, i.e. to analyze and prepare each field, select the most relevant ones. After this step, starts the data modeling phase and the model evaluation and fine tuning.

2.2 Data Exploration

For the data exploration we will be using Matplotlib and Seaborn packets, besisdes the Pandas packet, which will be used throughout the whole project.

We will be exploring the data present in all the columns, understanding how the data is distributed and considering

The data provided was already processed and there are no null values in any of the columns as can be seen:

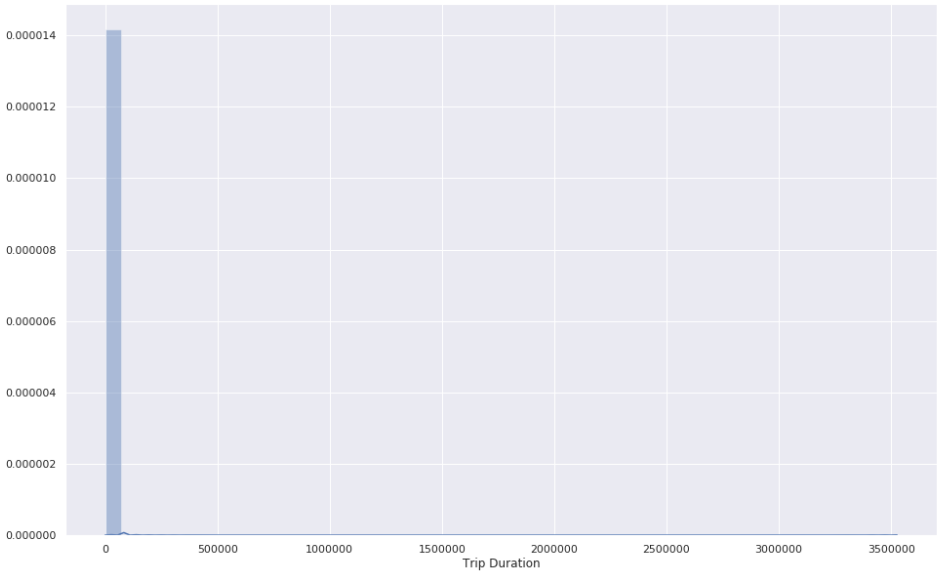
```
vendor_id            0
pickup_datetime      0
dropoff_datetime     0
passenger_count      0
pickup_longitude     0
pickup_latitude      0
dropoff_longitude    0
dropoff_latitude     0
store_and_fwd_flag   0
trip_duration        0
dtype: int64
```

2.2.1 Trip duration

First we will describe the `trip_duration` with simple statistical measurements like count, mean, standard deviation, quartlies and minimum and maximum values for the distribution:

```
count   1458644.000000
mean        959.492273
std        5237.431724
min           1.000000
25%         397.000000
50%         662.000000
75%        1075.000000
max     3526282.000000
Name: trip_duration, dtype: float64
```

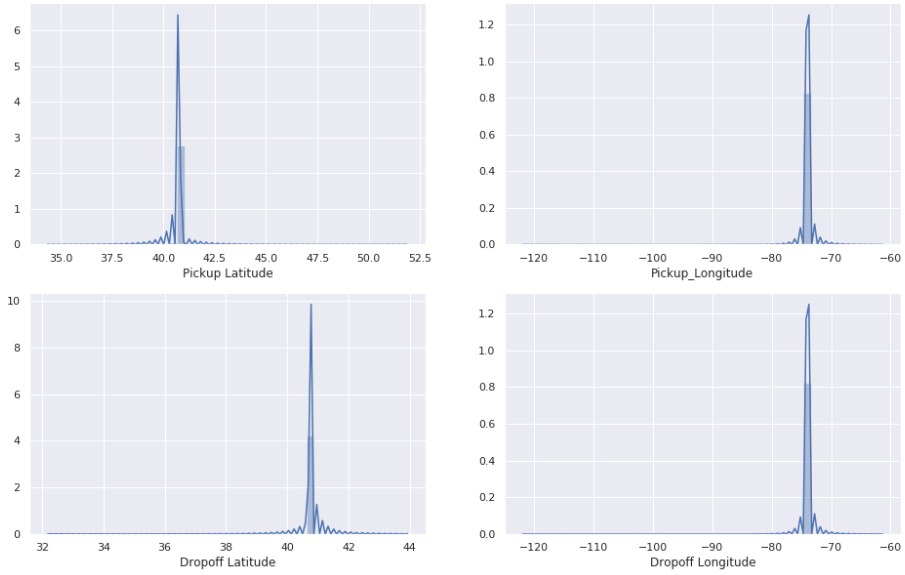
By the column description a few things come clear even though the data was pre-processed, no outliers were removed, that becomes quite clear by comparing the values of mean being much smaller than the std (standard deviation) and the maximum value for the distribution representing a trip almost 41 days long, which is not reasonable. A better way to confirm these assumptions is to plot the column values:



The graph confirm our assumption about the data, there are outliers in the distribution that must be removed before training the models, also as the distribution is skewed we will apply a log transform to the values in the column to achieve a better fit to the model.

2.2.2 Geo Location Coordinates

As we are speaking about coordinates, the best way to understanding the pickup and dropoff locations and how they are distributed in the dataset and check for outliers is a simple plot for each of the coordinates present in the data set:`pickup_latitude`, `pickup_longitude`, `dropoff_latitude` and `dropoff_longitude`:



The graph above show, as expected, most of the data is within the New York City bounding box, which is defined by the coordinates: (40.4774,-74.2589), (40.9176, -73.7004). However is also clear that there are a lot of outliers in the data, since there are latitude points ranging from around 32 to around 44 and longitude points from around -120 and around -60. Those values must be dropped before training the model

The lat-long coordinates can still be very useful, since some areas within the city have more traffic than others, impacting in the trip duration. To create zones inside the city, we will be using the lat-long for both the pickup place and dropoff place, but with the values rounded up to the third decimal place, with that we define areas in the city with the precision of approximately 111.32 m of radius [3]

2.2.3 Distance and Average Speed

Since the dataset provide a pickup point and a dropoff point, the obvious information that is possible to extract from that is the distance between the points, but since the distance is within a city and cities are formed by rectangular blocks, the distance between two points is better represented if we measure the taxicab metric or Manhattan distance, which is, in a simple way, the sum of the distance in the latitude axis summed with the distance in the longitude axis.

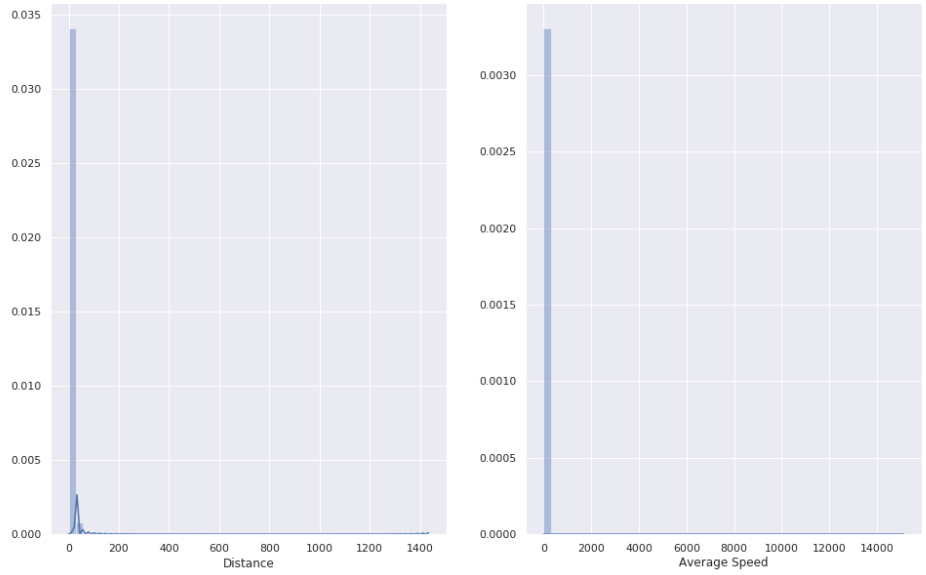
```
NYC_DEGREE_KM = 111.05938787411571

def calculate_city_block_distance(df_data):
    delta_lat = np.absolute(df_data.pickup_latitude - df_data.dropoff_latitude) * NYC_DEGREE_KM
    delta_lon = np.absolute(df_data.pickup_longitude - df_data.dropoff_longitude) * NYC_DEGREE_K
    return delta_lat + delta_lon
```

```
df_train['avg_speed'] = df_train['distance']/(df_train['trip_duration']/3600)
```

After calculating the distance and by having the trip duration, we are able to calculate the average speed for each trip. After converting the trip duration to hours and dividing the distance by that value, we find the average speed for the trip.

Since the two variables are extremely connected, we will shall analyze the graphs from both side by side.



As can be seen, here we will also find outliers, from the distance side the graph show values from zero to around 1300km and, by the average speed side, show speeds as high as 12,000 km/h, a speed well above the Land speed record of 1,227.985 km/h [1]

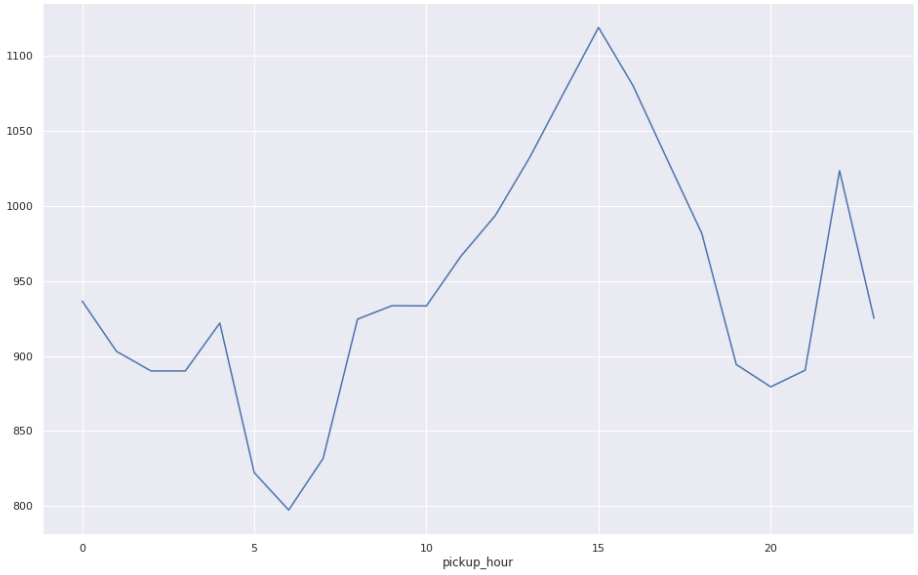
The average speed cannot be used to create a feature, once it depends on the trip duration, which we want to calculate, thus it can only be used to filter values that will add spurious information to our model. That said the average speed will be dropped before training the model.

2.2.4 Picukp Datetime

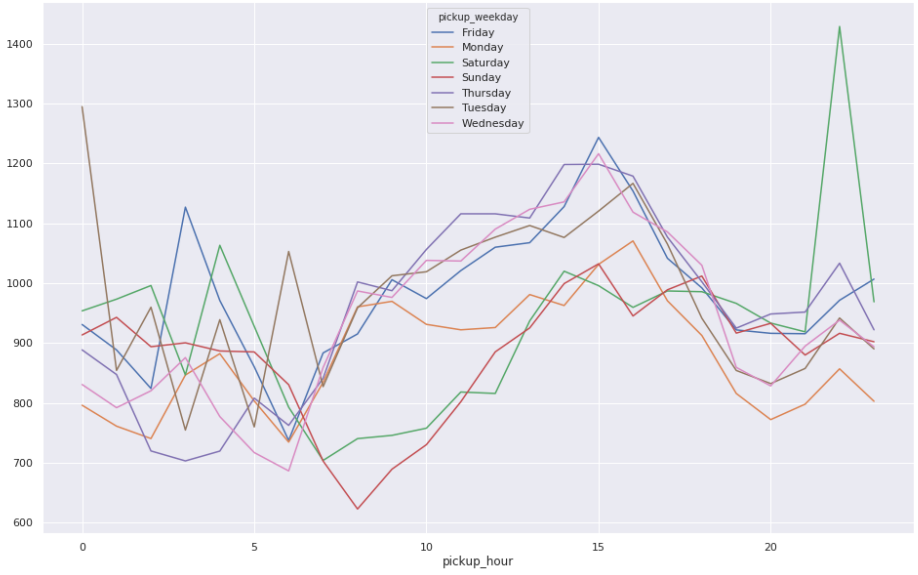
The date and time that the passenger was picked up is also a very important feature.

```
df_train['pickup_date'] = df_train['pickup_datetime'].dt.date
df_train['pickup_hour'] = df_train['pickup_datetime'].dt.hour
df_train['pickup_weekday'] = df_train['pickup_datetime'].dt.day_name()
```

Regarding the hours, we can apply the same line of reasoning, as the hours of the day present a typical behavior regarding traffic, for example, all big cities have their "Rush Hour" during which you can find a lot of traffic. The graph bellow illustrate that well.

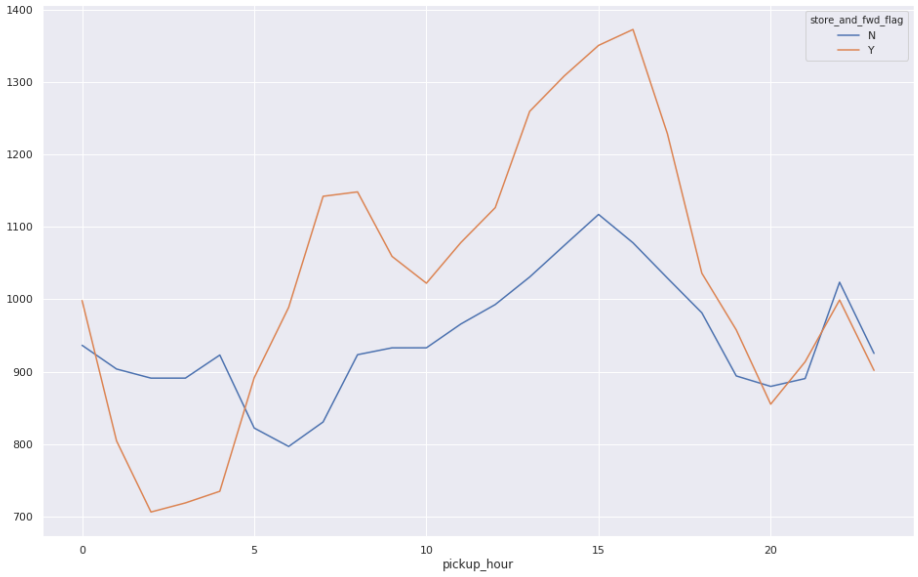


Speaking about the days each day of the week presents a typical behaviour is of common sense, for example that in weekdays the traffic is heavier than in weekends, what affects the trip duration. In that same line, in holidays a lighter traffic is expected. Bellow are the graph for average trip duration each day of the week and a graph comparing the holidays to normal days.



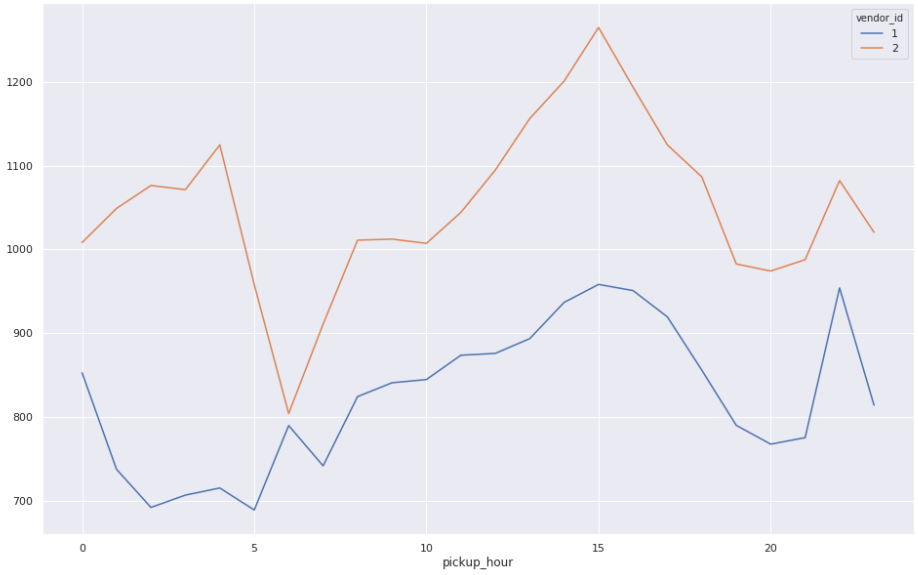
2.2.5 Store and Forward

The store and forward is a Boolean that at first look may not give any information about the trip duration, but looking the graph bellow, it becomes clear there is a difference between the two. That difference may be explained by the technologies available to drivers like navigation apps, Internet or more modern cars.



2.2.6 Vendor ID

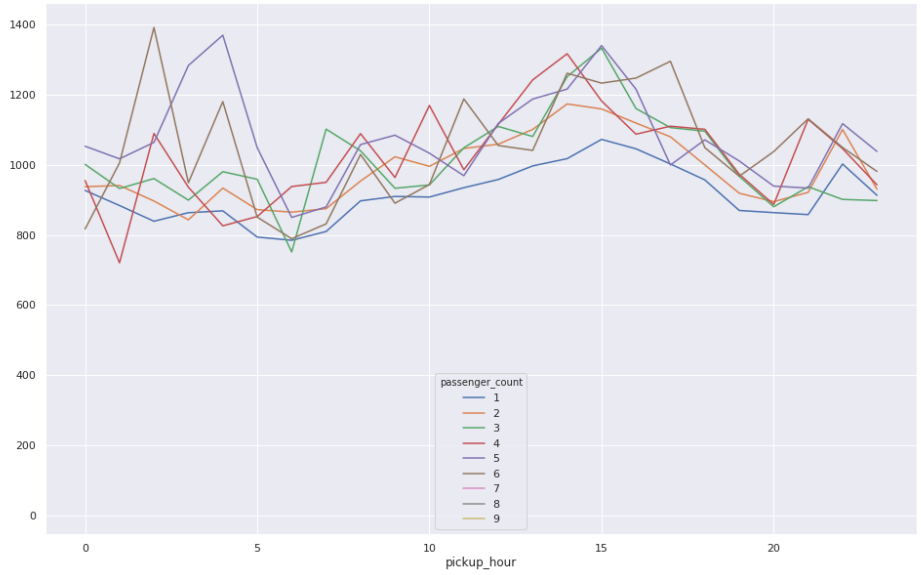
A naive analysis could lead to a misconception regarding the Vendor ID as a not useful information, but given that technology has changed the private transportation market, a good routing or driver selection algorithm or payment method can make a difference in the trip_duration. In the whole dataset, there are only 2 different vendors and both present a very different behaviour in trip duration as can be seen in the graph bellow.



2.2.7 Number of Passengers

In the same way as the previous analysis, it's worth to check the behavior of the number of passengers transported and if it might give more information about the trip duration.

As can be seen bellow, the graph of number of passengers shows a similar behavior for each number of passengers.



2.3 Benchmark

Regarding the Benchmark, as the capstone project was taken from a Kaggle's challenge, the obvious choice for the benchmark is the winner team, called **L2F** who achieved the score of 0.28976. However, given they're probably a well experienced team, a more conservative benchmark would be to achieve a score bellow 0.79807 achieved by using the naive prediction of determining the trip duration by the mean trip duration saw in the past, using the evaluation page provided by the challenge page.

3 Methodology

3.1 Data Preprocessing

After knowing better our data, it's time to process it filtering the outliers and transforming the data to a more adequate form to train the models.

After creating some other features in the previous section,besides having the original set of columns vendor id, pickup datetime, dropoff datetime, passenger count, pickup longitude, pickup latitude, dropoff longitude, dropoff latitude, store and fwd flag and trip duration, we now have some other features like distance, average speed, day of the week, hour of day and holiday.

3.1.1 Trip Duration

As observed we have a relevant number of outliers in the dataset for trip duration, so first, we must filter those values and to do that we will apply the Interquartile Range technique. As the minimum values are still very low, we will also filter any values less than 60 seconds.

Other features that can be easily filtered are the pickup and dropoff coordinates for that we define the bouncing box coordinates for New York City, here we define only two points to define that the top left point (north-east) and the bottom right (south-west) the coordinates are, respectively, (40.4774,-74.2589) and (40.9176, -73.7004). Bellow we filter the datapoints that are out of that box.

```
NYC_BOUNDING_BOX = [(40.4774,-74.2589), ( 40.9176, -73.7004)]

filter_lat_long = df_train['pickup_latitude'] < NYC_BOUNDING_BOX[1][0]
filter_lat_long &= df_train['pickup_latitude'] > NYC_BOUNDING_BOX[0][0]
filter_lat_long &= df_train['pickup_longitude'] < NYC_BOUNDING_BOX[1][1]
filter_lat_long &= df_train['pickup_longitude'] > NYC_BOUNDING_BOX[0][1]

filter_lat_long &= df_train['dropoff_latitude'] < NYC_BOUNDING_BOX[1][0]
filter_lat_long &= df_train['dropoff_latitude'] > NYC_BOUNDING_BOX[0][0]
filter_lat_long &= df_train['dropoff_longitude'] < NYC_BOUNDING_BOX[1][1]
filter_lat_long &= df_train['dropoff_longitude'] > NYC_BOUNDING_BOX[0][1]
```

Now we can search for outliers in the speed distribution speed, but now, we won't be using the interquartile range technique, since small values for average speed can be expected in some hours of the day. To filter the data, we will only use our understanding from the nature of the problem, thus we will filter any values less than 1 km/h or grater than 100 Km/h.

```
df_train = df_train[df_train['avg_speed'] < 100]
df_train = df_train[df_train['avg_speed'] > 1]
```

3.1.2 Passenger Count

There are some spurious values for passenger count, as can be seen bellow:

```
Out[3]: 1    1033540
        2     210318
        5      78088
        3      59896
        6      48333
        4      28404
        0         60
        7          3
        9          1
        8          1
        Name: passenger_count, dtype: int64
```

So we must filter any values of Passenger count equal to zero or greater than 6;

```
df_train = df_train[df_train['passenger_count']>0]
df_train = df_train[df_train['passenger_count']<7]
```

```
       travel_distance_km
count        1.458644e+06
mean         5.102859e+00
std          6.637813e+00
min          0.000000e+00
25%          1.788463e+00
50%          3.043757e+00
75%          5.610539e+00
max          1.435183e+03
```

3.1.3 One-hot enconding the Categorical Variables

To be properly used in supervised learning models the categorical fields independent of its type, must be transformed, since wrong information can be added to the model. One classical example of that are numerical categories where there is a logic relation between the numbers (greater than, sequences, , ratios, etc.) that might not exist in the category itself.

We will apply the pandas function get_dummies to transform the columns vendor_id, passenger_count,store_and_fwd_flag, pickup_weekday, pickup_hour, holiday.

```
cols = ['vendor_id', 'passenger_count','store_and_fwd_flag',
                'pickup_weekday', 'pickup_hour', 'holiday']
df_train = pd.get_dummies(df_train, columns=cols)
```

3.1.4 Dropping Columns

Now we have all our features we must drop the columns that we will no use to train the model. They are as follows: pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, pickup_datetime, pickup_date.

```
cols = ['pickup_longitude', 'pickup_latitude', 'dropoff_longitude',
                'dropoff_latitude', 'pickup_datetime', 'pickup_date', 'avg_speed']
df_train.drop(cols, axis=1, inplace=True)
```

3.1.5 Log Transformation

As the last step, since we found two skewed columns in the previous section we will make the log transform of those columns. Log transformation can reduce the skewness and make a non-linear distribution become linear.

```
df_train['trip_duration'] = np.log(df_train['trip_duration'] + 1)
df_train['distance'] = np.log(df_train['distance'] + 1)
```

3.1.6 The Train/Test Split

The first step to train a model is to define a Train dataset and a test dataset, for that we will split the original train dataset provided. We will be using 70% of the original dataset for training the model and the other 30% to validate the model and calculate the score

To split the data we use the pandas function train_test_split as follows:

```
X_train, X_test, y_train, y_test = train_test_split(df_X_train,
                df_y_train,
                test_size = 0.3,
                random_state = 3)
```

3.2 Implementation

In this section we will cover each step made to train the models and evaluate the best ones. To understand better how the model perform with the dataset, we will first run multiple models all with the default parameters

3.2.1 Linear Regression

To understand better the performance and the capabilities of a model and have quick first insights about how the data is being pre processed and is being predicted we will run one simple model, the chosen model for that is the multivariate Linear Regression. The linear regression model fit each feature with a linear functions with the least amount of error.

The default parameters are: `fit_intercept=True`, `normalize=False`, `copy_X=True` and `n_jobs=None`.

```
%%time
from sklearn.linear_model import LinearRegression
model = LinearRegression()
clf = model.fit(X_train, y_train)
y_pred = clf.predict(X_test)
final_score = kaggle_score( y_test, y_pred)

print(f'Linear Regression Score: {final_score}')
```

3.2.2 Lasso

Changing a little the approach, we'll use another linear regression model, using L1 regularization. The L1 regularization also try to fit a linear function to each feature, but differently from the previous method, it limits the size of the coefficients by applying a penalty equals to the absolute value of the magnitude of coefficients.

The default parameters are: `alpha=1.0`, `fit_intercept=True`, `normalize=False`, `precompute=False`, `copy_X=True`, `max_iter=1000`, `tol=0.0001`, `warm_start=False`, `positive=False`, `random_state=None`, `selection='cyclic'`

```
%%time
from sklearn.linear_model import Lasso
model = Lasso(random_state=3)
clf = model.fit(X_train, y_train)
y_pred = clf.predict(X_test)
final_score = kaggle_score( y_test, y_pred)

print(f'Lasso Score: {final_score}')
```

3.2.3 Ridge

Ridge is another linear regression model that also try to fit all features of the dataset with linear functions, but using a L2 regularization. The algorithm applies L2 penalty to the coefficients, the penalty is equal to the square root of the coefficients magnitudes.

The default parameters are: `alpha=1.0`, `fit_intercept`=True, `normalize`=False, `copy_X`=True, `max_iter`=None, `tol`=0.001, `solver`='auto', `random_state`=None

```
%%time
from sklearn.linear_model import Ridge
model = Ridge(random_state=3)
clf = model.fit(X_train, y_train)
y_pred = clf.predict(X_test)
final_score = kaggle_score( y_test, y_pred)

print(f'Ridge Score: {final_score}')
```

3.2.4 Decision Tree Regressor

Basically, the decision tree model works like a series of if-else questions applied for different features presents in the dataset that leads to a understanding of the target variable.

Decision trees intuitively are works well with the classification problem with categorical classes as features, but can also be used for regression models with continuous variables such as ours. For that the algorithm makes assumptions relating to the variable itself, like "is feature k larger than value x?". To build the final model the algorithm calculates all the possible relations and find the one that gives the most accurate information.

As the algorithm cover all the possible values, the Decision Trees are prone to overfitting the training data. As that happens when we have nodes with a single leaf to every possibility in the tree, the name for the technique to prevent that is called pruning and there are two different methods:

- Post-pruning: stop the creation of nodes preventing the creation of single nodes
- Pre-pruning: after building the tree, removes or collapse nodes with little information

The pruning technique can be determined by defining the depth of the tree, using the parameter `max_depth` or limiting the number of leaves present in a node with the parameter `max_leaf_nodes`.

To achieve a good precision, there are other parameters that can be tuned, like `min_samples_split`, which controls the minimum number of samples to split a node and `max_features` which control the maximum number of features that each tree will use during the train phase, among others

The default parameters are: `criterion`='mse', `splitter`='best', `max_depth`=None, `min_samples_split`=2, `min_samples_leaf`=1, `min_weight_fraction_leaf`=0.0, `max_features`=None, `random_state`=None, `max_leaf_nodes`=None, `min_impurity_decrease`=0.0, `min_impurity_split`=None, `presort`=False

```
%%time
from sklearn.tree import DecisionTreeRegressor
model = DecisionTreeRegressor(random_state=3)
```

```
clf = model.fit(X_train, y_train)
y_pred = clf.predict(X_test)
final_score = kaggle_score( y_test, y_pred)

print(f'Decision Tree Regressor Score: {final_score}')
```

3.2.5 Random Forest Regressor

In a few words, Random Forest is an ensemble method that uses a set of slightly different approach to the decision trees method. The Random Forest, as the name suggests uses a number of different trees (Forest) that differs from each other by the number of features that they takes in consideration and by the features itself, the features and the number of features are randomly chosen.

To predict or classify, the trained model averages the result from the trees.

To build each tree of the forest, a random set of features and a booststrap[6] sample of the original data are chosen. Basically the bootstrap sample contains have the same size of the original sample, but some items might be repeated and some others of the original may not be present. That helps prevent overfitting.

In the Scikit-Learn's implementation of the Random Forest, some parameters are available and must be chosen carefully, like `max_features`, which defines the max number of features that a single tree can use, `n_estimators` which defines the number of treens in the forest and as explained for the Decision Tree Regressor, `max_depth`, `max_leaf_nodes`, `min_samples_split`, `max_features`

The default parameters are: `n_estimators='warn'`, `criterion='gini'`, `max_depth=None`, `min_samples_split=2`, `min_samples_leaf=1`, `min_weight_fraction_leaf=0.0`, `max_features='auto'`, `max_leaf_nodes=None`, `min_impurity_decrease=0.0`, `min_impurity_split=None`, `bootstrap=True`, `oob_score=False`, `n_jobs=None`, `random_state=None`, `verbose=0`, `warm_start=False`, `class_weight=None`

```
%%time
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor(random_state=3)
clf = model.fit(X_train, y_train)
y_pred = clf.predict(X_test)
final_score = kaggle_score( y_test, y_pred)

print(f'Random Forest Regressor Score: {final_score}')
```

3.2.6 Gradient Boosting Regressor

Gradient Boosted trees is another ensemble method that combines multiple decision trees to compose a better and more precise final model.

Differently from the Random Forest, the decision trees are not created in an random way, instead the trees are created in a more sequential manner, with one tree trying to correct mistakes from the previous one. The algorithm uses strong pre-pruning, leading to a final model composed of shallow trees. This characteristic contributes to a faster training that consumes less memory.

As the name suggests, the Gradient Boosting Regressor uses the gradient of a loss function to evaluate how to improve in each iteration, the loss function can be defined by the `loss` parameter, also related to the learning through iteration there is the `learning_rate`, which defines the

amount of contribution of each tree to the model. Here we can also find other parameters found in Random Forest and Decision Tree methods like `max_depth`, `max_leaf_nodes`, `min_samples_split`, `max_features`.

The default parameters are: loss='ls', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, min_impurity_split=None, init=None, random_state=None, max_features=None, alpha=0.9, verbose=0, max_leaf_nodes=None, warm_start=False, presort='auto', validation_fraction=0.1, n_iter_no_change=None, tol=0.0001

```
%%time
from sklearn.ensemble import GradientBoostingRegressor
model = GradientBoostingRegressor(random_state=3)
clf = model.fit(X_train, y_train)
y_pred = clf.predict(X_test)
final_score = kaggle_score( y_test, y_pred)

print(f'Gradient Boosting Regressor Score: {final_score}')
```

3.2.7 XGBoost: Extreme Gradient Boosting

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.[7]

It has been chosen due to its large use in industry to solve real life problems with a great amount of data and . XGBoost, as the Gradient Boosting Regressor, also uses small trees and implements a gradient boosting algorithm applied to a loss function to find the best learning method.

The XGBoost implementation is very configurable, providing a great number of parameters to be tuned, for that reason we recomend that the default parameter should be checked in the XGBoost Parameters page

```
%%time
import xgboost as xgb
model = xgb.XGBRegressor(random_state=3)
clf = model.fit(X_train, y_train)
y_pred = clf.predict(X_test)
final_score = kaggle_score( y_test, y_pred)

print(f'XGBoost Score: {final_score}')
```

3.2.8 Adaboost Regressor

Adaboost is another ensemble method that uses a weighted combination of weak learners (models slightly better then a random guess) to produce the final result.

It's a simple iterative method that uses a sample of the data with the same weight to all the points to train weak learners, those learners are used to predict the target value, which is compared

to the true one. In the next iterations the points with the bigger error receives the higher weight and the process starts again.

The training ceases when the error functions is the same between interactions or the number o the total number of estimators is reached.

As other ensemble methods, it has some parameters already covered like `n_estimators`, `loss` function and `learning_rate`.

The default parameters are: base_estimator=None, n_estimators=50, learning_rate=1.0, loss='linear', random_state=None

```
%%time
from sklearn.ensemble import AdaBoostRegressor
model = AdaBoostRegressor(random_state=3)
clf = model.fit(X_train, y_train)
y_pred = clf.predict(X_test)
final_score = kaggle_score( y_test, y_pred)

print(f'Gradient Boosting Regressor Score: {final_score}')
```

3.3 Refinement

As describe before the scoring function measures the error, thus a smaller error will mean a better score. Bellow are all the models evaluated in order from the best performing to the worst performing:

1. **Gradient Boosting:** 0.40877528042762556
2. **XGBoost:** 0.40888857095046693
3. **Ridge:** 0.4277240533226883
4. **Linear Regression:** 0.42772463461077576
5. **AdaBoost:** 0.43804008122671456
6. **Random Forest:**0.45598979449492705
7. **Decision Tree:** 0.568131730069767
8. **Lasso:** 0.6725745338011018

For this section we'll only work with the 3 best performing models, that is the models with the lower values for the scores.

To fine tune we will use simultaneously cross validation and Grid Search, provided by scikit-learn in a single function. GridSearch implements an exhaustive search method through all the different combinations of Hyperparameters that the user wants to test in the model and with the help of cross validation evaluate which combination of the hyperparameters gives the best result for the dataset.

Cross-validation is a technique to verify how robust and general is the model when making predictions to different datasets.

In cross-validation the original data set is divided in k parts of the same size, in each interaction the model to be validated is trained using $k-1$ parts and later predicts the target variable in the remaining part of the original dataset and a score is calculated with a determined score function.

After calculating the score for the configuration of train/test datasets, the dataset which was used to test becomes part of the training dataset and one of the parts that belonged to the train dataset becomes the test dataset and the process repeats until all the k parts were used as test dataset. The final score for the model is the average of the scores found in each interaction

3.3.1 Ridge

The score for Ridge using the default parameters was 0.4277240533226883.

The fine tuning tested the combination of the following parameters for the model using 3-fold cross-validation to evaluate the best result:

- alpha: 0.5, 1.0, 3.0, 4.0, 5.0, 10.0
- solver: *auto*, *least-squares* (lsqr), *Stochastic Average Gradient descent* (SAG), *Singular Value Decomposition* (SVD)

The final set of parameters chosen after running the GridSearch are *alpha* = 5.0 and *solver* = *svd*, which had no practical improvement over the previous run of the Rdge model using the default parameters. The final score achieved is: 0.4277241321921506

Bellow the implementation used to find the best parameters:

```
%%time
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer

model = Ridge(random_state=3)
scorer = make_scorer(kaggle_score, greater_is_better= False)

parameters = {
    'alpha': [0.5, 1.0, 3.0, 4.0, 5.0, 10.0],
    'solver': ['auto', 'lsqr', 'sag', 'svd']
    }

clf = GridSearchCV(model, param_grid=parameters, scoring=scorer, verbose=10, n_jobs=-1, cv=3)
grid_fit = clf.fit(X_train, y_train)

best_params = grid_fit.best_params_
best_clf = grid_fit.best_estimator_
final_score = kaggle_score(y_test, best_clf.predict(X_test))
    best_models['Ridge'] = {
    'final_score':final_score,
    'clf':best_clf,
    'params':best_params
    }

print(f'Ridge best parameters: {best_params}')
print(f'Ridge Score: {final_score}')
```

3.3.2 Gradient Boosting Regressor

The score for Gradient Boosting Regressor using the default parameters was 0.40877528042762556.

The fine tuning tested the combination of the following parameters for the model using 2-fold cross-validation to evaluate the best result:

- max_depth: 3, 5
- n_estimators: 100, 200
- min_samples_split: 2, 6
- learning_rate : 0.1, 1.0

The final set of parameters chosen after running the GridSearch are *learning_rate*: 0.1, *max_depth*: 5, *min_samples_split*: 6, *n_estimators*: 200, which improved the score to 0.3985323203593734

Bellow the implementation used to find the best parameters:

```
%%time
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer

model = GradientBoostingRegressor(random_state=3)
scorer = make_scorer(kaggle_score, greater_is_better= False)

parameters = {
    'max_depth': [3, 5],
    'n_estimators': [100, 200],
    'min_samples_split': [2, 6],
    'learning_rate': [0.1, 1.0]
    }

clf = GridSearchCV(model, param_grid=parameters, scoring=scorer, verbose=10, cv=2, n_jobs=4)
grid_fit = clf.fit(X_train, y_train)

best_params = grid_fit.best_params_
best_clf = grid_fit.best_estimator_
final_score = kaggle_score(y_test, best_clf.predict(X_test))
best_models['GradientBoost'] = {
    'final_score':final_score,
    'clf':best_clf,
    'params':best_params
    }

print(f'Gradient Boosting Regressor best parameters: {best_params}')
print(f'Gradient Boosting Regressor  Score: {final_score}')
```

3.3.3 XGBoost

The score for XGBoost Regressor using the default parameters was 0.409344079071674.

The fine tuning tested the combination of the following parameters for the model using 2-fold cross-validation to evaluate the best result:

- max_depth: 5, 8, 10
- n_estimators: 200, 300
- learning_rate: 0.05, 0.1

- reg_lambda: 1.0, 5

The final set of parameters chosen after running the GridSearch are *learning_rate*: 0.1, *max_depth*: 5, *n_estimators*: 300, *reg_lambda*: 5, which reasonably improved the score to 0.3981089784145823

Bellow the implementation used to find the best parameters:

```
%%time
import xgboost as xgb
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer

model = xgb.XGBRegressor(random_state=3)
scorer = make_scorer(kaggle_score, greater_is_better=False)

parameters = {
    'max_depth': [5, 8, 10],
    'n_estimators': [200, 300],
    'learning_rate': [0.05, 0.1,],
    'reg_lambda': [1.0, 5] }

clf = GridSearchCV(model, param_grid=parameters, scoring=scorer, verbose=10, cv=2, n_jobs=4)
grid_fit = clf.fit(X_train, y_train)

best_params = grid_fit.best_params_
best_clf = grid_fit.best_estimator_
final_score = kaggle_score(y_test, best_clf.predict(X_test))
best_models['XGBoost'] = {
    'final_score':final_score,
    'clf':best_clf,
    'params':best_params
    }

print(f'XGBoost best parameters: {best_params}')
print(f'XGBoost Score: {final_score}')
```

3.4 Hyperparameters Fine Tuning Analysis

After fine tuning the 3 best performing models, the on that presented the best score was once again the XGBoost. One thing worth noticing is the average time used to validate each of the parameters:

- Ridge: 72 fits run in 3 minutes and 35 seconds
- Gradient Boosting: 32 fits in 58 minutes and 6 seconds
- XGBoost: 48 fits in 2 hours, 4 minutes and 29 second

Also, the gain obtained in this phase was quite small, specially in the Ridge Model:

- XGBoost: 0.010779592535884619

- Gradient Boosting: 0.01024296006825215
- Ridge: 7.886946229440639e-08

The final score after tuning the Hyperparamenters are:

1. XGBoost: 0.39852413102654594
2. Gradient Boosting: 0.3985323203593734
3. Ridge: 0.4277241321921506

4 Results

The Kaggle challenge was already finished, but still accepts late submissions, so we'll use our 3 best models after the fine tuning and make the late submission to get the real submission score.

4.1 Model Evaluation

4.1.1 Preparing the Test Dataset

Before applying the model to the original test dataset, we must create the same features that were created in the model exploration and fine tuning.

Also, the dimensions for train and test dataset must be the same, with the same column names. As we dropped any data for trips with 0 or 9 passengers, the columns generated with the *pd.get_dummies* (passenger_count_0 and passenger_count_9) must also be dropped in the test dataset

```
df_test = pd.read_csv(PATH_TEST_DATASET, infer_datetime_format=True,
                          parse_dates=['pickup_datetime'], index_col='id')

df_test['pickup_date'] = df_test['pickup_datetime'].dt.date
df_test['pickup_hour'] = df_test['pickup_datetime'].dt.hour
df_test['pickup_weekday'] = df_test['pickup_datetime'].dt.day_name()

holidays = [day.date() for day in calendar().holidays(
                                      start=df_test['pickup_date'].min(),
                                      end=df_test['pickup_date'].max())]
df_test['holiday'] = df_test['pickup_date'].isin(holidays)
df_test.drop('pickup_date', axis=1, inplace=True)

df_test['distance'] = calculate_city_block_distance(df_test)
df_test['distance'] = np.log(df_test['distance'] + 1)

df_test = pd.get_dummies(df_test, columns=['vendor_id', 'passenger_count',
                                      'store_and_fwd_flag', 'pickup_weekday',
                                      'pickup_hour', 'holiday'])

df_test.drop(['pickup_datetime', 'pickup_longitude',
                'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude'],
                axis=1, inplace=True)
```

4.1.2 Preparing the Submission File

The submission file must be formatted to be accepted by the automated submission scoring system. A sample of the final format is given with the datasets in the Challenge's page.

```
import os
BASE_PATH_KAGGLE_SUBMISISON = './out'

def create_txt_file_for_submission(df_data, file_name):

    final_path = os.path.abspath(os.path.join(BASE_PATH_KAGGLE_SUBMISISON,file_name))
    final_path += '.txt'
    df_data.index.name = 'id'
    print(final_path)
    df_data.to_csv(final_path,
                   sep=',',
                   header=True,
                   na_rep=df_data['trip_duration'].quantile(0.5)
                   )
```

4.1.3 Predicting the trip duration for the original test dataset

As we stored the best models we'll now use an empty copy of them with the same parameters and train that again now with the whole original train dataset to get a better model for the final submission.

After predicting the values for trip duration one submission file will be prepared for each model.

```
for model_name, model_specs in best_models.items():
    clf = sklearn.base.clone(model_specs['clf'])
    clf.fit(df_X_train, df_y_train)
    y_predict = clf.predict(df_test)
    y_predict = np.exp(y_predict) - 1
    df_result = pd.DataFrame(y_predict, columns=['trip_duration'], index=df_test.index.values)
    create_txt_file_for_submission(df_result, model_name)
```

4.1.4 The final results

After submiting the prediction for each model, the final Scores are as follows:

1. XGBoost: 0.49571
2. Gradient Boost: 0.49750
3. Ridge: 0.51109

4.2 Model Validation

As the XGBoost model achieved the best performance, we will validate only that. In order to do that we must show how robust the solution, using multiple values for `random_state` parameter

present both in the model definition and in the `train_test_split` function that divide the test and train dataset.

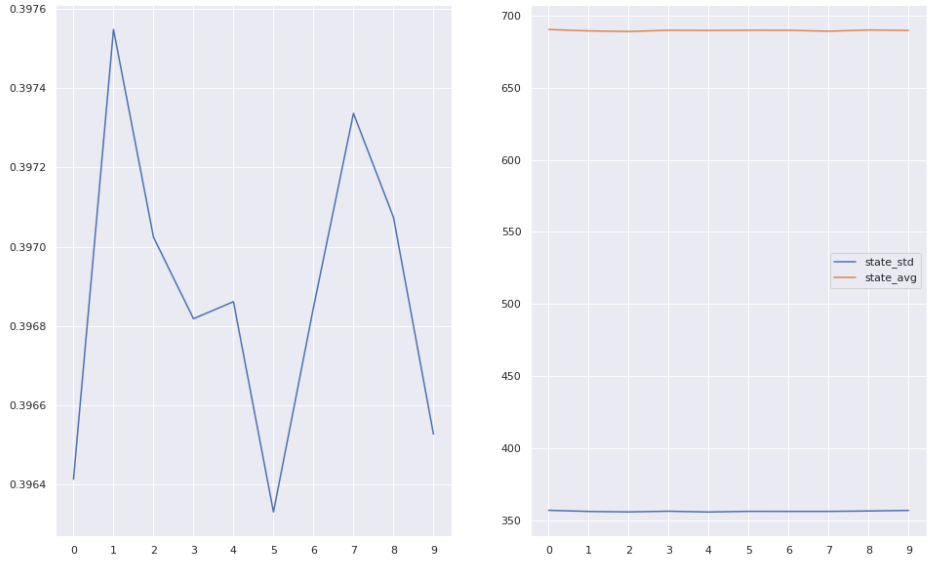
To validate that, the following piece of code was used:

```
best_overall_model = best_models['XGBoost']
state_avg = []
state_std = []
state_score = []
for i in range(10):
    print(f'Runing Iteration {i}')
    random_state = np.random.randint(low=0, high=100)
    X_train, X_test, y_train, y_test = train_test_split(df_X_train,
                                                        df_y_train,
                                                        test_size = 0.3,
                                                        random_state = random_state)
    clf = sklearn.base.clone(best_overall_model['clf'])
    clf.random_state = random_state
    clf.nthread=6
    clf.fit(df_X_train, df_y_train)
    y_predict = clf.predict(X_test)

    state_score.append(kaggle_score(y_test, y_predict))
    y_predict = np.exp(y_predict) - 1
    state_avg.append(y_predict.mean())
    state_std.append(y_predict.std())

df_result_free = pd.DataFrame(
    {'state_score': state_score,
        'state_avg': state_avg,
        'state_std': state_std
    })
```

The code generates the graph bellow:



As can be seen in the graph above, even by totally changing the train and test datasets by setting 10 different values for `random_state` and applying those values both for the `train_test_split` function and the model and training the model, the values for average of trip duration keep between 688.913 and 690.306. The values for standard deviation and and score for each run also vary little staying between 355.8198 and 357.0060 and 0.39633 and 0.39754 respectively.

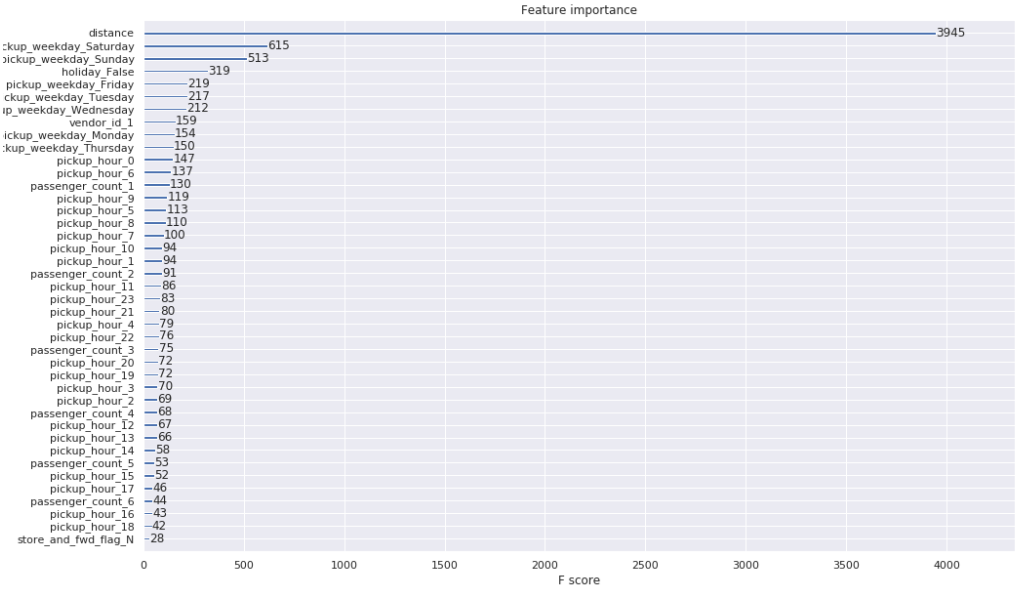
4.3 Justification

Besides been a robust model, XGBoost performed well, with the score of 0.49571, and achieved a notable improvement in score over our naive model benchmark proposition, of 0.79807.

5 Conclusion

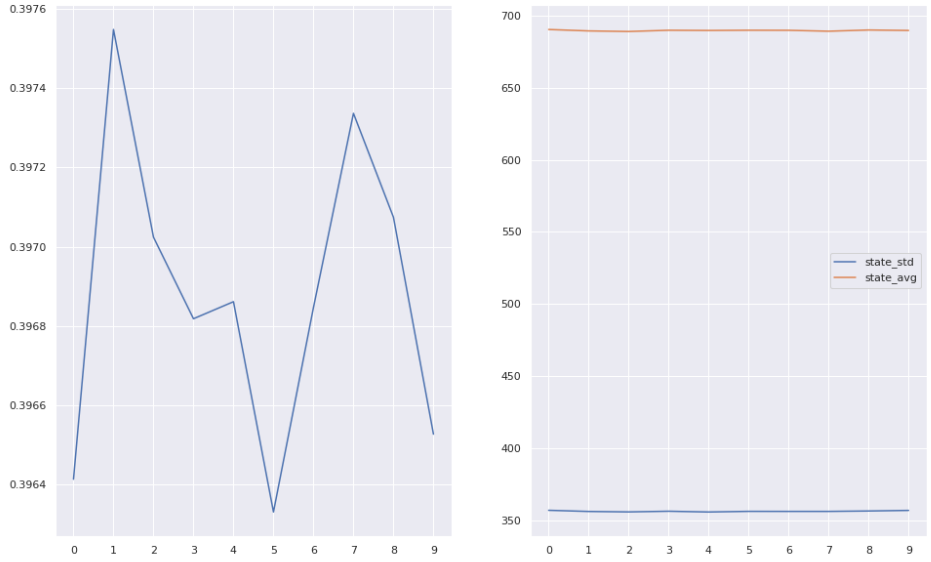
5.1 Free Form Visualization

One of the most interesting information that we can extract from models is the importance that each column have in the predicting process, i.e. tho plot the values in `feature_importances_`. Bellow we have a graph showing the importance that each feature have in the prediction, the most important, higher is the value.



As can be seen in the graph, it makes total sense to have features like `distance`, `pickup_weekday_saturday`, `pickup_weekday_sunday` and `holiday_false` as the most important for the trip_duration, since it's from common sense that if you have more distance to cover, you will take longer to arrive, and in weekends and holidays to traffic changes drastically.

The feature importance also can give more insights about the distribution, for example, here we see that distance feature was created from the pickup and dropoff coordinates is the major contributor for trip duration prediction, thus we can assume that those coordinates have a big impact in the problem and more features could be extracted from them, or the distance between the two coordinates in the city could be measured with a higher precision, using APIs from Google Maps, for example



5.2 Reflection

Throughout the project, many topics related do data manipulation and visualization were covered.

The first step was to properly load the data, adjusting the data types, and understand the data, not only what meant each of the columns, but understand how the data was distributed in each column, which of the features presented outliers and how to get rid of them without changing or loosing too much data.

The cleaning step involved a lot of data visualization, even though we could make the basic statistical description of each column and analyze its quartiles, standard deviation, mean, maximum and minimum values visualization gives us a better perspective of which kind of distribution we are working with.

We also had to create a few features more, not only to train the data, but also to clean some of the spurious information in the data. In that step we prepare all the data, making log transformation to make the `Distance` and `trip_duration` columns more suitable to train the model.

To finish, we trained simple models to verify which of the one could lead to a more accurate result, fine tuned many parameters from the 3 best models and finally evaluated them against the Kaggles Leaderboard, achieving the score of 0.49571 as our best score.

5.3 Improvements

Regarding improvements, the first step would be to play more with the hyperparameters to fine tune them even more and make our model more accurate. Another point of improvement is to understand better in what areas the traffic is more intense by using clusterization, at the end we missed a lot of information given by the pickup and dropoff coordinates.

We saw in the free form visualization that the distance is the most important feature for the model, so the best approach would be calculate the most precise distance possible by using external sources of data like Google Maps API.

Another way to improve the models is to add more sources of data, like weather data in New York City around as a whole, since weather and traffic are intimately related.

References

[1] 1997: Land Speed Record,
http://www.guinnessworldrecords.com/news/60at60/2015/8/1997-land-speed-record-392880

[2] New York City Taxi Trip Duration,
https://www.kaggle.com/c/nyc-taxi-trip-duration

[3] Decimal Degrees,
https://en.wikipedia.org/wiki/Decimal_degrees

[4] Learning to Predict the Duration of an Automobile Trip,
https://www.aaai.org/Papers/KDD/1998/KDD98-037.pdf

[5] Dynamic O-D travel time estimation using an artificial neural network,
https://ieeexplore.ieee.org/document/518845

[6] Bootstrap: A Statistical Method ,
http://stat.rutgers.edu/home/mxie/RCPapers/bootstrap.pdf

[7] XGBoost Documentation,
https://xgboost.readthedocs.io