CSYE 7245 – Big Data Systems & Intelligence Analytics

Research Paper

New York City Taxi Trip Duration

Ninad Gadre

Index

Abstract

Yellow Medallion Taxicabs are iconic to New York City. There are currently over 13,000 licensed taxicabs and over 50,000 taxicab drivers providing transportation for passengers in all five boroughs via street hails. Medallion taxicab drivers must follow a set of TLC requirements in order to be licensed to drive a yellow taxicab. In this analysis, we try to address some of the long-standing problems in the transportation industry i.e. to predict the duration at the beginning of the trip. With the advent of technology based cab services, this problem has become particularly important for the drivers and the customers. A good prediction mechanism can be instrumental for drivers in optimizing their returns, while also saving the customers from the uncertainties attached to a trip. In our analysis, we attempt to examine and understand the provided feature and also attempt to find the best possible ways to leverage those features. The millions of rides taken each month can provide insight into traffic patterns, road blockage, or large-scale events that attract many New Yorkers. With ridesharing apps gaining popularity, it is increasingly important for taxi companies to provide visibility to ride duration, since the competing apps provide these metrics upfront. Predicting duration of a ride can help passengers decide when is the optimal time to start their commute, or help drivers decide which of two potential rides will be more profitable, for example. Furthermore, this visibility into fare will attract customers during times when ridesharing services are implementing surge pricing.

Introduction

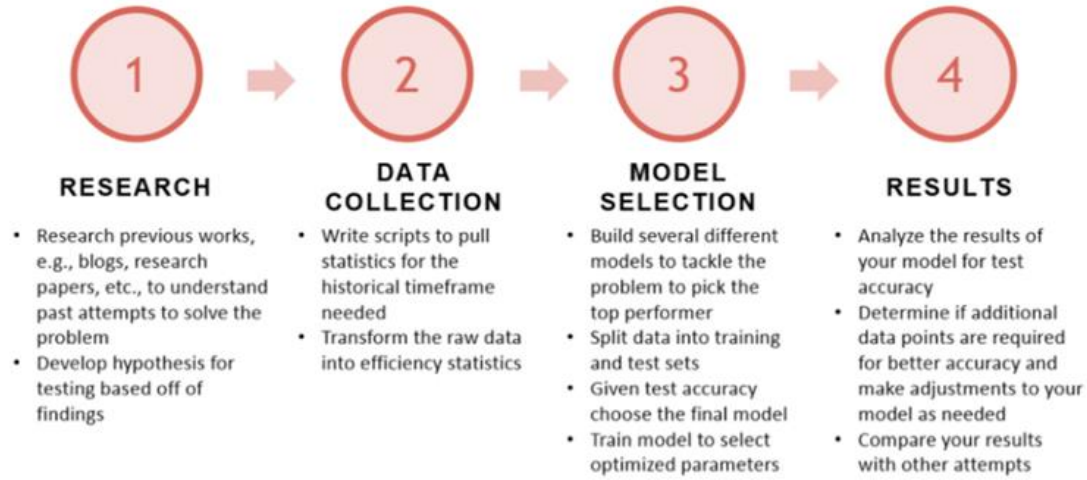


Fig 2. Approach for the Research Project

This may not come as much of a surprise, but according to the annual INRIX Global Traffic Scorecard, New York City is the third most congested city in the world in terms of traffic and the second worst in

the United States. Per the 2017 analysis, New York drivers averaged 91 peak hours stuck in traffic last year, tying with Moscow, Russia for second place for the most amount of hours spent in congestion. Additionally, New York drivers spent 13 percent of their time sitting in congestion, with 11 percent of that being attributed to daytime traffic. The city was only beat by Los Angeles, who ranks number one in the country and the world for most congested traffic. What's even worse, four out of ten of the worst U.S. corridors are found right here in New York. For the third consecutive year, the eastbound section of the Cross Bronx Expressway (I-95) tops the list[1]. INRIX reports that on average, driver waste 118 hours per year in congestion on the 4.7 mile stretch— an increase of 37 percent from the previous year. It might not be too hard to believe that traffic congestion will cost the city $100 billion over the next five years after all. In the meantime, Governor Andrew Cuomo's Fix NYC panel is scheduled to reveal a congestion pricing plan that could result in a charge for vehicles driving below 60th Street in Manhattan during specified hours. Over the last four years, Uber, Lyft and other app-based ride services have put 50,000 vehicles on the streets of New York City. Customers embraced these new services as offering a prompt, reliable and affordable option for traveling around town. Their growth also raises questions about their impact on traffic congestion and on public transit and taxi services that are essential components of urban transportation networks[1]. A dearth of factual information has made it difficult, however, to assess their role in the city's transportation network or decide whether a public policy is needed. This report presents a detailed analysis of the growth of app-based ride services in New York City, their impacts on traffic, travel patterns and vehicle mileage, and implications for achieving critical City goals for mobility, economic growth and environmental sustainability in New York and other major cities[2]. Findings are based on trip and mileage data that are uniquely available in New York City, providing the most detailed and comprehensive assessment of these new services in any U.S. city. One way to predict duration is by doing short term prediction with the help of real time data collection. In [3] the authors tackle the problem by using data from buses (GPS) and an algorithm based on Kalman filters. Using a similar approach, [4] uses real time data from smartphone placed inside vehicles. In [5] the authors use a combination of traffic modelling, real time data analysis and traffic history to predict travel time in congested freeways. In [6] the prediction is done using Support Vector Regression (SVR) while in [7] Neural Networks (SSNN) are used. We are trying to solve a similar problem: estimating ride duration without real time data, by analyzing data collected from taxis. The data provides the details of yellow taxi rides in the New York City from Jan 2016 to June 2016. This data is provided by the NYC Taxi and Limousine Commission (downloaded from Kaggle). The data for each month is about 1.8GB and consists of roughly over 10 million trips. Each trip records fields: pick-up and drop-off dates/times, pick-up and drop-off coordinates, trip distances, passenger counts, id, vendor id and store_and_fwd_flag.

From a preliminary analysis, it was evident that there were few discrepancies in the dataset. For some cases the duration was zero or very low; for some the ratio of duration by distance was unreasonably high or low.

Weather data collected from the National Weather Service. It contains the first six months of 2016, for a weather station in central park. It contains for each day the minimum temperature, maximum temperature, average temperature, precipitation, new snow fall, and current snow depth.

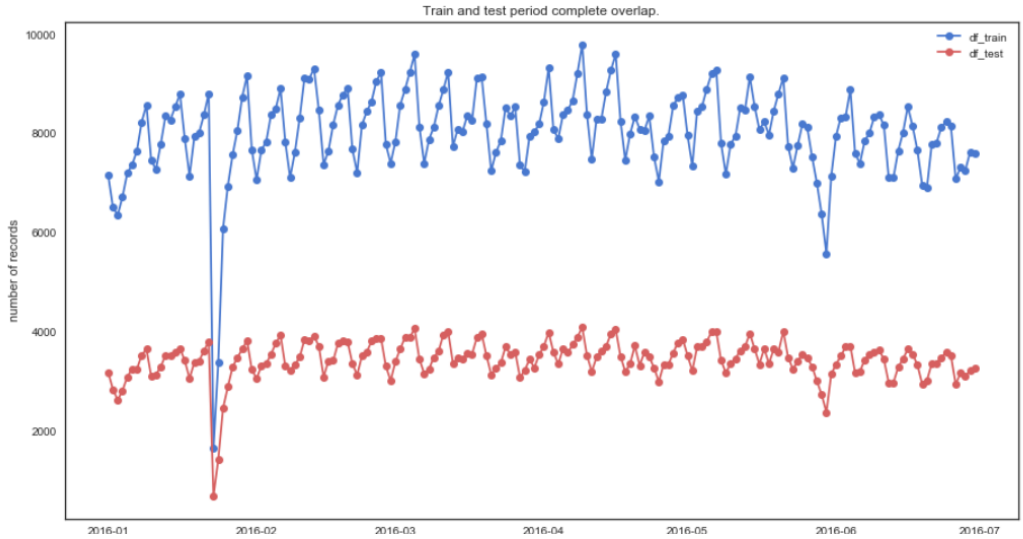
Code with Documentation

https://github.com/gadren/gadre_ninad_spring2018_BDSIA/tree/master/Final_project

Validation Strategy

First let's check the train test split. It helps to decide our validation strategy and gives ideas about feature engineering.

In [41]:
```
f = plt.figure(figsize=(15,8))
plt.plot(df_train.groupby('pickup_date').count()[['id']], 'o-', label='df_train')
plt.plot(df_test.groupby('pickup_date').count()[['id']], 'o-', label='df_test', color='r')
plt.title('Train and test period complete overlap.')
plt.legend(loc=0)
plt.ylabel('number of records')
plt.show()
```



Clustering

K-means Clustering

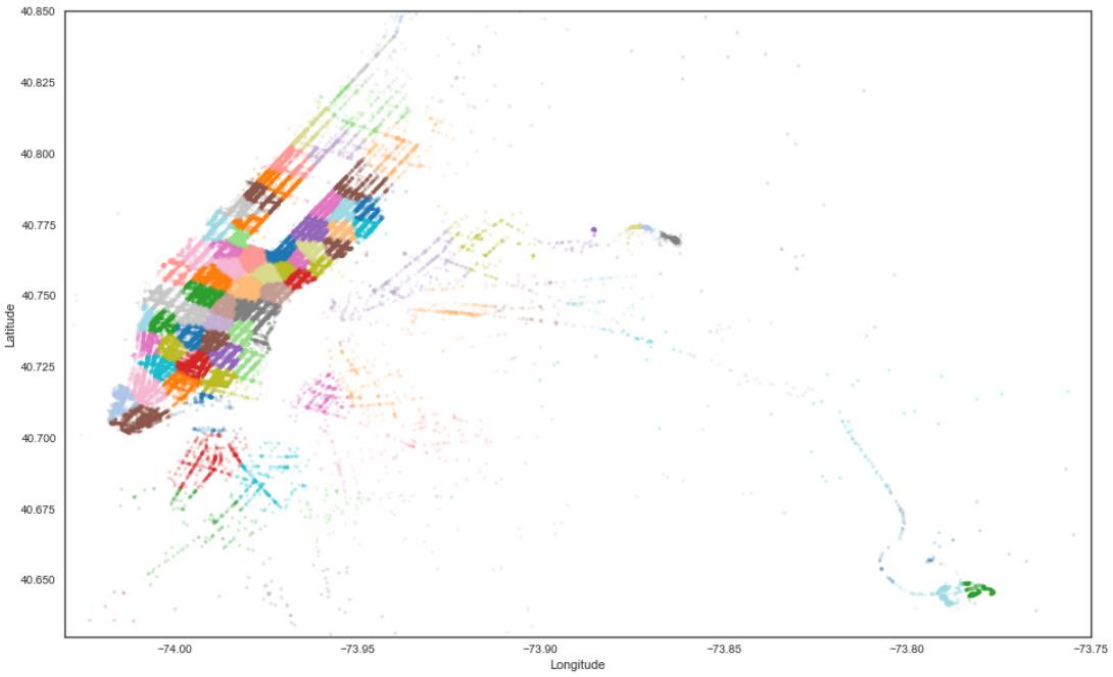
K-means clustering is a type of unsupervised learning, which is used when you have unlabeled data (i.e., data without defined categories or groups). The goal of this algorithm is to find groups in the data, with the number of groups represented by the variable K. The algorithm works iteratively to assign each data point to one of K groups based on the features that are provided. Data points are clustered based on feature similarity. Rather than defining groups before looking at the data, clustering allows you to find and analyze the groups that have formed organically. The "Choosing K" section below describes how the number of groups can be determined.Each centroid of a cluster is a collection of feature values which define the resulting groups. Examining the centroid feature weights can be used to qualitatively interpret what kind of group each cluster represents.

In [44]:
```
samples = np.random.permutation(len(coords))[:500000]
kmeans = MiniBatchKMeans(n_clusters=100, batch_size=10000).fit(coords[samples])
```

In [45]:
```
df_train.loc[:, 'pickup_cluster'] = kmeans.predict(df_train[['pickup_latitude', 'pickup_longitude']])
df_train.loc[:, 'dropoff_cluster'] = kmeans.predict(df_train[['dropoff_latitude', 'dropoff_longitude']])
df_test.loc[:, 'pickup_cluster'] = kmeans.predict(df_test[['pickup_latitude', 'pickup_longitude']])
df_test.loc[:, 'dropoff_cluster'] = kmeans.predict(df_test[['dropoff_latitude', 'dropoff_longitude']])
```

In [46]:
```
#************************************************************************************
#*    This was adapted from a post from
#*    Author: Louis Po-Wei Chen
#*    Date: 04/23/2018
#*    Availability: https://github.com/louispoweichen/New_York_City_Taxi_Trip_Duration
#*
#************************************************************************************/

N = 100000 # number of sample rows in plots
city_long_border = (-74.03, -73.75)
city_lat_border = (40.63, 40.85)
fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(16,10))
ax.scatter(df_train.pickup_longitude.values[:N], df_train.pickup_latitude.values[:N], s=10, lw=0,
           c=df_train.pickup_cluster[:N].values, cmap='tab20', alpha=0.2)
ax.set_xlim(city_long_border)
ax.set_ylim(city_lat_border)
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
plt.show()
```



In [38]:
```
#*****************************************************************************************
#*    This was adapted from a post from
#*    Author: Kulbear
#*    Date: 04/23/2018
#*    Availability: https://github.com/Kulbear/New-York-City-Taxi-Trip-Duration/blob/master/xgboost.ipynb
#*
#*****************************************************************************************/

import xgboost as xgb
y = np.log(train_df['trip_duration'].values + 1) #We will convert the trip duration into logarithmic values for the sake of ML
train_x, val_x, train_y, val_y = train_test_split(train_df[feature_names].values, y, test_size=0.2)

#The data is stored in a DMatrix object
#Missing values can be replaced by a default value in the DMatrix constructor
dtrain = xgb.DMatrix(train_x, label=train_y)
dvalid = xgb.DMatrix(val_x, label=val_y )
dtest = xgb.DMatrix(test_df[feature_names].values)
watchlist = [(dtrain, 'train'), (dvalid, 'valid')]

# We have used random search for XGBoost
xgb_pars = {'min_child_weight': 50, 'eta': 0.3, 'colsample_bytree': 0.3, 'max_depth': 10,
            'subsample': 0.8, 'lambda': 1., 'nthread': -1, 'booster' : 'gbtree', 'silent': 1,
            'eval_metric': 'rmse', 'objective': 'reg:linear'}
```

In [39]:
```
# You could try to train with more epoch,the rmse value might stall after few epochs sometimes
model = xgb.train(xgb_pars, dtrain, 100, watchlist, early_stopping_rounds=50,
                  maximize=False, verbose_eval=10)

#If you have a validation set, you can use early stopping to find the optimal number of boosting rounds.
#Early stopping requires at least one set in evals. If there's more than one, it will use the last.
#train(..., evals=evals, early_stopping_rounds=10)
```

```
[0]     train-rmse:4.21862      valid-rmse:4.21747
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.
[10]    train-rmse:0.176757     valid-rmse:0.181808
[20]    train-rmse:0.108635     valid-rmse:0.118087
[30]    train-rmse:0.10046      valid-rmse:0.112307
[40]    train-rmse:0.095502     valid-rmse:0.110124
[50]    train-rmse:0.0919       valid-rmse:0.10861
[60]    train-rmse:0.088545     valid-rmse:0.107903
[70]    train-rmse:0.086194     valid-rmse:0.107953
[80]    train-rmse:0.083958     valid-rmse:0.107904
[90]    train-rmse:0.080878     valid-rmse:0.108227
[99]    train-rmse:0.07929      valid-rmse:0.108135
```

In [40]:
```
print('Modeling RMSLE %.5f' % model.best_score)
```

```
Modeling RMSLE 0.10775
```

In [41]:
```
predictions = model.predict(dvalid)
```

In [42]:
```
#Predicted values
y_pred = np.exp(predictions)-1
print(y_pred[:10])
y_pred = pd.DataFrame(y_pred, columns = ['y_pred'])
```

```
[ 399.8948   222.12178  916.3345  1128.9053   241.03188 3114.3386
 2042.732    916.4408  1773.5358   429.2949 ]
```

In [43]:
```
#Original values
y_org = np.exp(val_y)-1
print(y_org[:10])
y_org= pd.DataFrame(y_org, columns = ['y_org'])
```

```
[ 393.  216.  940. 1136.  239. 3140. 2227.  990. 1683.  434.]
```

In [44]:
```
#Comparing the predicted and original values
result = pd.concat([y_org,y_pred], axis=1)
result.head()
```

Out[44]:

Out[44]:

	y_org	y_pred
0	393.0	399.894806
1	216.0	222.121780
2	940.0	916.334473
3	1136.0	1128.905273
4	239.0	241.031876

Random Forest Regressor

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

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if bootstrap=True (default)

Randomized search on hyper parameters.

RandomizedSearchCV implements a “fit” and a “score” method. It also implements “predict”, “predict_proba”, “decision_function”, “transform” and “inverse_transform” if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings.

In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by n_iter.

If all parameters are presented as a list, sampling without replacement is performed. If at least one parameter is given as a distribution, sampling with replacement is used. It is highly recommended to use continuous distributions for continuous parameters.

In [45]:
```
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)#random_state is the seed used by the random number generator;

#We will try and select the best parameters for Random Forest to give good accuracy
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) #clf is the estimator and
validator.fit(train_x,train_y)
#param_distributions : dict
#Dictionary with parameters names (string) as keys and distributions or lists of parameters to try

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)
print(validator.best_estimator_.max_features)
```

```
0.9826610120711197
10
20
5
1
10
```

Ridge

In [49]:
```
#Ridge
from sklearn import datasets
from sklearn.linear_model import Ridge
from scipy.stats import randint as sp_randint
from sklearn.model_selection import GridSearchCV
import warnings
warnings.filterwarnings('ignore')

#This model solves a regression model
#where the loss function is the linear least squares function and regularization is given by the l2-norm
ridge = Ridge()

#param_grid will be give a range of values for alpha, which provides regularization strength in Ridge function
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(train_x,train_y)
print(validator.best_score_)
print(validator.best_estimator_.alpha)
```

```
0.49843961136619813
0.05
```

In [50]:
```
#'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution.
ridge_reg = Ridge(alpha = 0.05, solver = 'cholesky')
ridge_reg.fit(train_x,train_y)
```

Out[50]:
```
Ridge(alpha=0.05, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='cholesky', tol=0.001)
```

In [51]:
```
predictions_ridge = ridge_reg.predict(val_x)
```

In [52]:
```
dec_mse_ridge = mean_squared_error(predictions_ridge, val_y)
rmse_ridge = np.sqrt(dec_mse_ridge)
print(rmse_ridge)
```

```
0.5108069344777587
```

Lasso

In [53]:
```
from sklearn.linear_model import Lasso
lasso = Lasso()
param_grid = {"alpha": [0.01,0.05,0.001,0.0001,0.000001,0.005,0.0005,0.00005]}

validator = GridSearchCV(lasso, param_grid= param_grid)
validator.fit(train_x,train_y)
print(validator.best_score_)
print(validator.best_estimator_.alpha)
```

```
0.5017025114087248
0.0001
```

In [54]:
```
#alpha: Constant that multiplies the L1 term
#random_state: The seed of the pseudo random number generator that selects a random feature to update
#The parameters can be changed
lasso_reg=Lasso(alpha =0.05, random_state=1)
lasso_reg.fit(train_x,train_y)
```

Out[54]:
```
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)
```

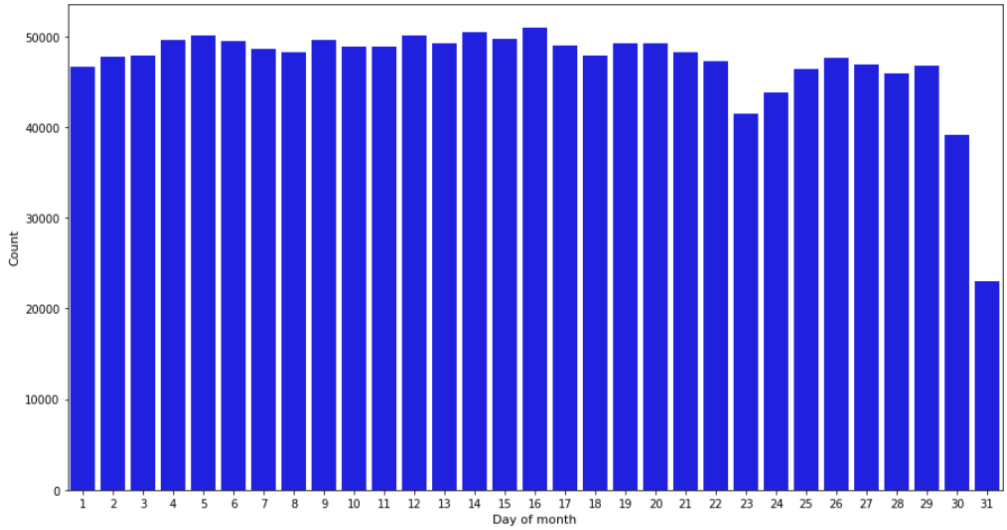
In [55]:
```
predictions_lasso = lasso_reg.predict(val_x)
```

In [56]:
```
dec_mse_lasso = mean_squared_error(predictions_lasso,val_y)
rmse_lasso = np.sqrt(dec_mse_lasso)
print(rmse_lasso)
```

```
0.529727936744906
```

Results

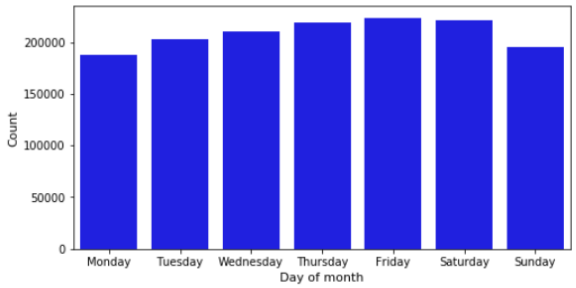
```
In [53]: #Checking at what date of a month the number of trips are more
         f = plt.figure(figsize=(15,8))
         sns.countplot(x='pickup_daymonth', data=df_train, color = 'blue')
         plt.xlabel('Day of month', fontsize=11)
         plt.ylabel('Count', fontsize=11)
         plt.show()
```



In the above figure I have plotted number of taxi trips during a certain day of the month. As we can see from above figure, the number of trips stays uniform throughout the month except for the last few days of the month.

In the above figure I have plotted number of taxi trips during a certain day of the month. As we can see from above figure, the number of trips stays uniform throughout the month except for the last few days of the month.

In [54]:
```
#Checking at what date of the week the number of trips are more
f = plt.figure(figsize=(8,4))
day_of_week = [i for i in range(7)]
sns.countplot(x='pickup_day', data=df_train, color = 'blue')
plt.xlabel('Day of month', fontsize=11)
plt.ylabel('Count', fontsize=11)
plt.xticks(day_of_week, ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'))
plt.show()
```



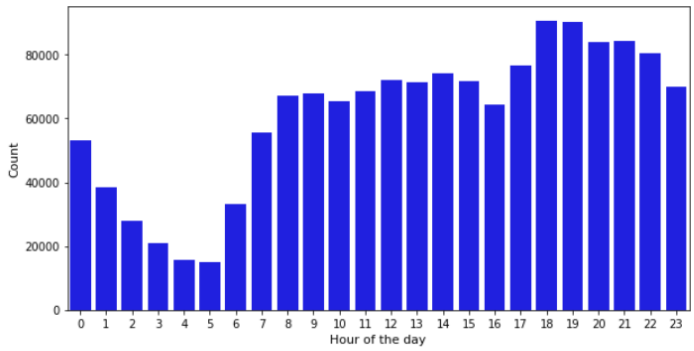
In the above figure I have plotted number of taxi trips during a certain day of the week. As we can see from above figure, the number of trips usually spikes, especially during the weekend i.e. Friday and Saturday, while it stays low on Monday.

In [55]:
```
print(df_train['pickup_month'].value_counts())
```

```
3    256189
4    251645
5    248487
2    238300
```

In the above figure I have plotted number of taxi trips during a certain day of the week. As we can see from above figure, the number of trips usually spikes, especially during the weekend i.e. Friday and Saturday, while it stays low on Monday.

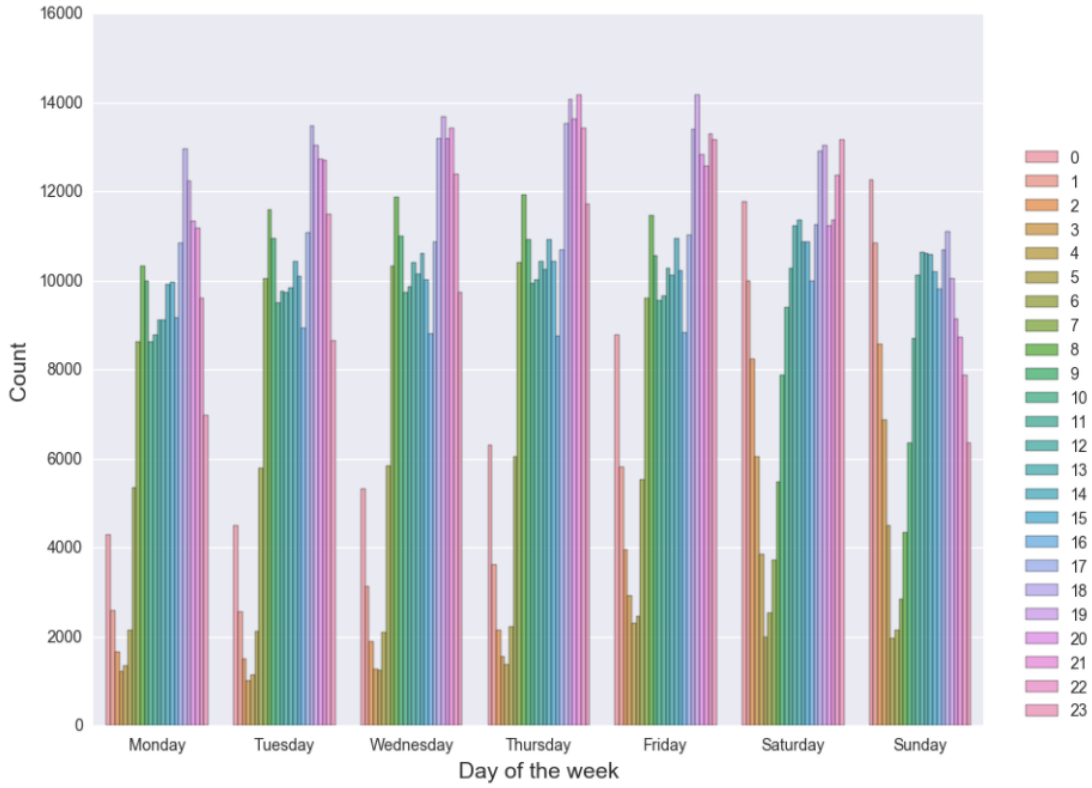
In [56]:
```
f = plt.figure(figsize=(10,5))
sns.countplot(x='pickup_hour', data=df_train, color = 'blue')
plt.xlabel('Hour of the day', fontsize=11)
plt.ylabel('Count', fontsize=11)
plt.show()
```



In the above figure I have plotted number of taxi trips during a certain hour of the day. As we can observe from the above figure that the number of trips usually rises after 6pm till midnight

In the above figure I have plotted number of taxi trips during a certain hour of the day. As we can observe from the above figure that the number of trips usually rises after 6pm till midnight.

In [21]:
```
f = plt.figure(figsize=(10,8))
days = [i for i in range(7)]
sns.countplot(x='pickup_day', data=df_train, hue='pickup_hour', alpha=0.8)
plt.xlabel('Day of the week', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.xticks(days, ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'))
plt.legend(loc=(1.04,0))
plt.show()
```

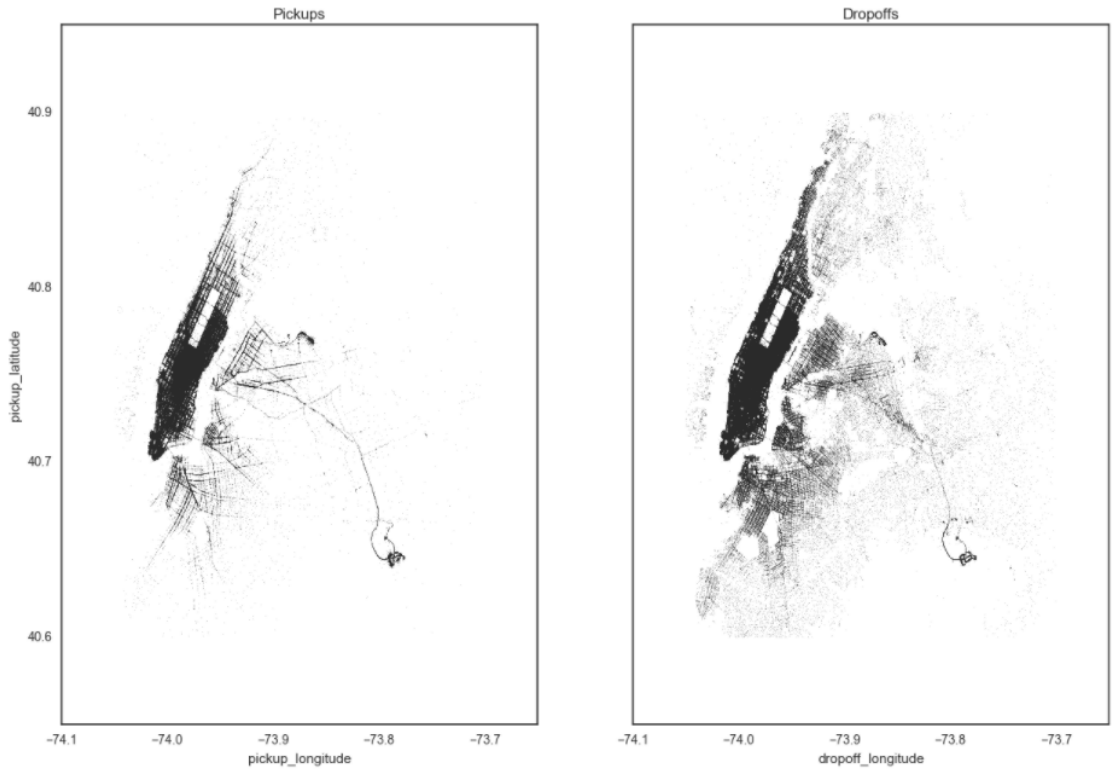


In the above figure I have tried to combine all the results I got above into one single plot. On the Y-axis we have the count of trips and on the X-axis, we have day of the week combined with the hour of the day. As we can observe, count of trips is more on weekend with the count spikes in peak hours from 6pm till midnight.

```
location_df.plot(kind='scatter', x='pickup_longitude', y='pickup_latitude',
                 color='white',
                 s=.02, alpha=.6, subplots=True, ax=ax1)
ax1.set_title("Pickups")
#ax1.set_facecolor('black')

location_df.plot(kind='scatter', x='dropoff_longitude', y='dropoff_latitude',
                 color='white',
                 s=.02, alpha=.6, subplots=True, ax=ax2)
ax2.set_title("Dropoffs")
#ax2.set_facecolor('black')
```

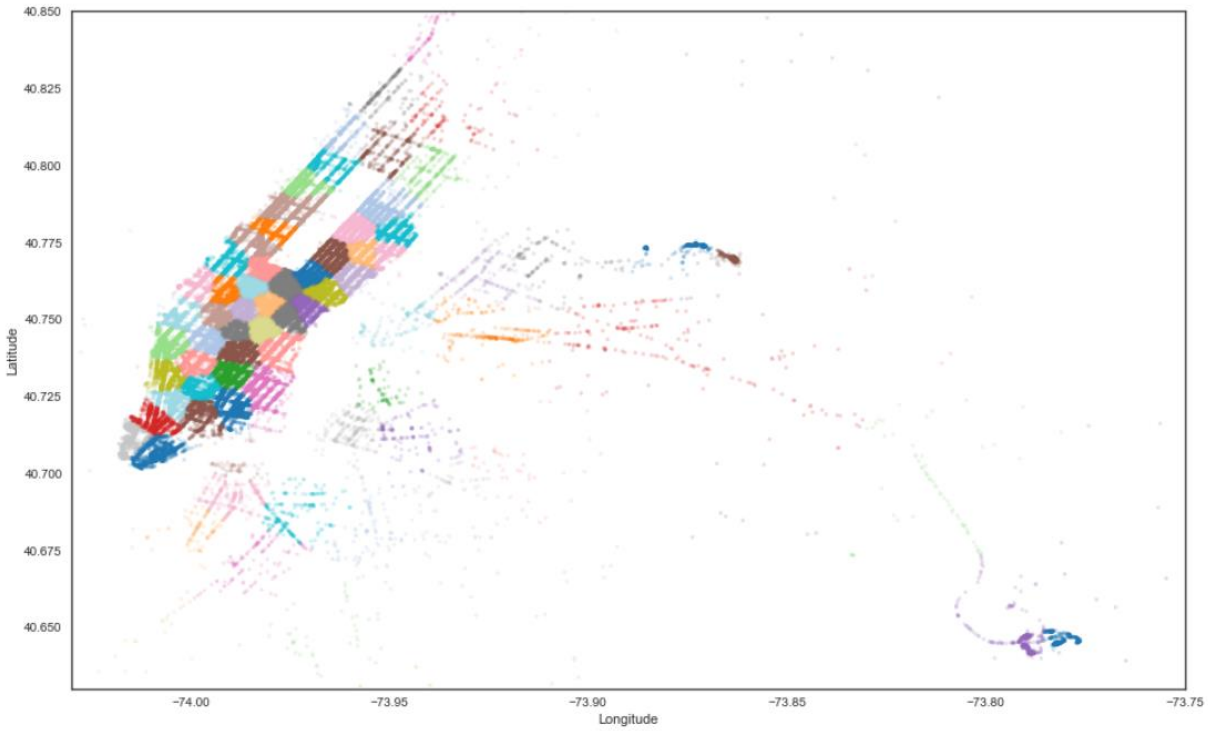
Out[44]: <matplotlib.text.Text at 0x274bc3e8668>



In the plot above, I have used the pickup and drop off latitude and longitude data to get a clear view of the busy locations. The pickup location is dense in the central New York area like Manhattan and Central Park, while the drop off location is almost spread throughout New York

Discussion

We obtained very interesting insight when we plotted our cluster centroids over the map of New York. As cluster centroids are representative of the areas from where most of the pickups and drop offs of the rides occur.



We got RMSLE value of 0.39 for Gradient boosting algorithm. The value is low which means the algorithm performed well in predicting the trip duration values. To get a good accuracy score, a lot of new features were added to the original dataset, features such as haversine distance, manhattan distance and bearing array. Speed for all these distances was calculated to give a better insight. The dataset given doesn't contain sufficient features to predict the target variable efficiently, I have observed different trip durations for the same distance on different days.

RMSLE values before adding extra features

Name of Algorithm	**RMSLE value**
XGBoost	0.3894
Random Forest	0.4022
Ridge	0.6357
Lasso	0.6377
Ensemble Learning	1.039
Neural Network	2.0074

RMSLE value after adding holiday and weather as features

Name of Algorithm	**RMSLE value**
XGBoost	0.1077
Random Forest	0.1103
Ridge	0.5107
Lasso	0.5297
Ensemble Learning	0.9652
Neural Network	2.0074

From the above 2 tables, we get the lowest RMSLE value for Gradient Boosting algorithm for prediction of trip duration. Lasso Regression, Ridge Regression, Random Forest, Gradient Boosting, Neural Network and an ensemble method (combination of gradient boosting, random forest, lasso and ridge) were trained to predict ride duration. The results show that Random Forest and Gradient Boosting performed better than the rest. Amongst all the algorithm Neural Network has the worst performance, which was expected. The gradient boosting model outperform all other models used, although the model accounts for the effect of pickup and drop off locations, it has no way of modeling the effects of the locations along the route. A ride between two locations with high traffic can still be relatively fast if it goes through high-speed areas with little or no traffic. Although using rides in hour and the average speed in hour improves the models and hence works as proxies for traffic modeling, rotating the location coordinates does not yield any significant improvement in prediction accuracy. Significant improvement in the accuracy is achieved when external features are added such as weather and holidays. The dataset doesn't contain sufficient amount of information to predict the target variable. To be able to predict and get good accuracy for prediction external data was needed to add to the original data. There very instances where the trip duration was very large for smaller distance, the reason behind them must be different like traffic jam, weather or increase number of tourist etc. The dataset gives only the pickup and drop off co-ordinates and not the route which was taken by the cab (The taxi might have used a longer route instead of a shorter one). Inclusion of all such factors will help in predicting the right trip duration.

References

[1] Josh Grinberg, Arzav Jain, Vivek Choksi. Predicting Taxi Pickups in New York City. Final Paper for CS221, Autumn 2014

[2] Yunrou Gong, Bin Fang, shuo zhang and Jingyu Zhang Predict New York City Taxi Demand

[3] Vanajakshi, L., S. C. Subramanian, and R. Sivanandan. "Travel time prediction under heterogeneous traffic conditions using global positioning system data from buses." IET intelligent transport systems 3.1 (2009): 1-9

[4] Biagioni, James, et al. "Easytracker: automatic transit tracking, mapping, and arrival time prediction using smartphones." Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems. ACM, 2011.

[5] Yildirimoglu, Mehmet, and Nikolas Geroliminis. "Experienced travel time prediction for congested freeways." Transportation Research Part B: Methodological 53 (2013): 45-63.

[6] Wu, Chun-Hsin, Jan-Ming Ho, and Der-Tsai Lee. "Travel-time prediction with support vector regression." IEEE transactions on intelligent transportation systems 5.4 (2004): 276-281.

[7] Van Lint, J. W. C., S. P. Hoogendoorn, and Henk J. van Zuylen. "Accurate freeway travel time prediction with state-space neural networks under missing data." Transportation Research Part C: Emerging Technologies 13.5 (2005): 347-369.