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

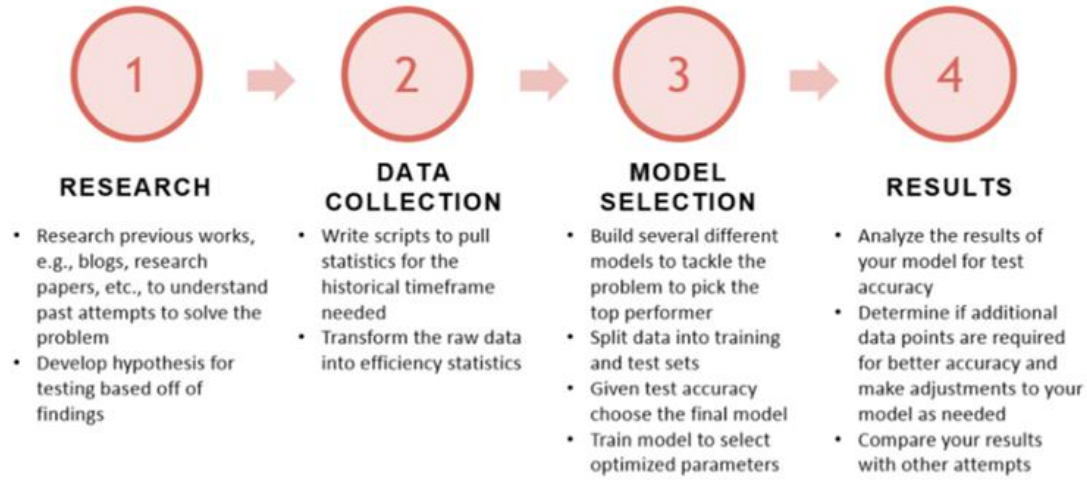


Fig 2. Approach for the Research Project

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

New York Taxi Trip Duration Analysis

by **Ninad Gadre**

Introduction

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

In [28]:
```
from IPython.display import Image
Image('/Users/ninad/Desktop/BDSIN/Final Project/Approach.jpg')
```

Out[28]:

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.

New York City Taxi Trip Duration Dataset

-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

As will be doing exploratory analysis in this notebook we will import all the libraries required to visualize the data

In [2]:
```
#First we will import all the required modules
import pandas as pd
import numpy as np
import seaborn as sns
```

Load train Data, we have used parse_dates function while loading the data as the train dataset has columns which can be date and time nut are in string format

In [3]:
```
import os
path = os.getcwd()
print(path)
df_train = pd.read_csv(path+'/train.csv', parse_dates=['pickup_datetime','dropoff_datetime'])
```

```
C:\Users\ninad\Desktop\BDSIN\Final Project\Scripts
```

Split the datetime columns into different columns having different information like date, day of the week, month, year etc

In [4]:
```
df_train['pickup_hour'] = df_train['pickup_datetime'].dt.hour
df_train['pickup_minute'] = df_train['pickup_datetime'].dt.minute
df_train['pickup_second'] = df_train['pickup_datetime'].dt.second
df_train['pickup_day'] = df_train['pickup_datetime'].dt.dayofweek
df_train['pickup_daymonth'] = df_train['pickup_datetime'].dt.day
df_train['pickup_month'] = df_train['pickup_datetime'].dt.month
df_train['pickup_year'] = df_train['pickup_datetime'].dt.year
df_train['pickup_date'] = df_train['pickup_datetime'].dt.date
```

In [5]:
```
df_train.head()
```

Out[5]:

	id	vendor_id	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd
0	id2875421	2	2016-03-14 17:24:55	2016-03-14 17:32:30	1	-73.982155	40.767937	-73.964630	40.765602	
1	id2377394	1	2016-06-12 00:43:35	2016-06-12 00:54:38	1	-73.980415	40.738564	-73.999481	40.731152	
2	id3858529	2	2016-01-19 11:35:24	2016-01-19 12:10:48	1	-73.979027	40.763939	-74.005333	40.710087	
3	id3504673	2	2016-04-06 19:32:31	2016-04-06 19:39:40	1	-74.010040	40.719971	-74.012268	40.706718	

Load the test dataset with the same technique used for loading the train dataset

In [6]:
```
df_test = pd.read_csv(path+'/test.csv', parse_dates=['pickup_datetime'])
df_test.head()
```

Out[6]:

	id	vendor_id	pickup_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd_flag
0	id3004672	1	2016-06-30 23:59:00	1	-73.988129	40.732029	-73.990173	40.756680	N
1	id3505355	1	2016-06-30 23:59:00	1	-73.964203	40.679993	-73.959808	40.655403	N
2	id1217141	1	2016-06-30 23:59:00	1	-73.997437	40.737583	-73.986160	40.729523	N
3	id2150126	2	2016-06-30 23:59:00	1	-73.956070	40.771900	-73.986427	40.730469	N
4	id1598245	1	2016-06-30 23:59:00	1	-73.970215	40.761475	-73.961510	40.755890	N

In [7]:
```
df_test['pickup_datetime'] = pd.to_datetime(df_test.pickup_datetime)
```

In [8]:
```
df_test['pickup_date'] = df_test['pickup_datetime'].dt.date
```

We will use .describe() on the train dataset which will generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

In [9]:
```
df_train.describe().T
```

Out[9]:

	count	mean	std	min	25%	50%	75%	max
vendor_id	1458644.0	1.534950	0.498777	1.000000	1.000000	2.000000	2.000000	2.000000e+00
passenger_count	1458644.0	1.664530	1.314242	0.000000	1.000000	1.000000	2.000000	9.000000e+00
pickup_longitude	1458644.0	-73.973486	0.070902	-121.933342	-73.991867	-73.981743	-73.967331	-6.133553e+01
pickup_latitude	1458644.0	40.750921	0.032881	34.359695	40.737347	40.754101	40.768360	5.188108e+01
dropoff_longitude	1458644.0	-73.973416	0.070643	-121.933304	-73.991325	-73.979752	-73.963013	-6.133553e+01
dropoff_latitude	1458644.0	40.751800	0.035891	32.181141	40.735885	40.754524	40.769810	4.392103e+01
trip_duration	1458644.0	959.492273	5237.431724	1.000000	397.000000	662.000000	1075.000000	3.526282e+06
pickup_hour	1458644.0	13.606484	6.399693	0.000000	9.000000	14.000000	19.000000	2.300000e+01

The output of .shape is a tuple that gives the number of rows or the number of records and the number of columns of the dataframe

In [10]:
```
print(df_train.shape)
```

```
(1458644, 19)
```

The .info() will display the information of all the columns in the dataframe with their NaN's and datatype

In [11]:
```
df_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1458644 entries, 0 to 1458643
Data columns (total 19 columns):
id                  1458644 non-null object
vendor_id           1458644 non-null int64
pickup_datetime     1458644 non-null datetime64[ns]
dropoff_datetime    1458644 non-null datetime64[ns]
passenger_count     1458644 non-null int64
pickup_longitude    1458644 non-null float64
pickup_latitude     1458644 non-null float64
dropoff_longitude   1458644 non-null float64
dropoff_latitude    1458644 non-null float64
store_and_fwd_flag  1458644 non-null object
trip_duration       1458644 non-null int64
pickup_hour         1458644 non-null int64
pickup_minute       1458644 non-null int64
pickup_second       1458644 non-null int64
pickup_day          1458644 non-null int64
pickup_daymonth     1458644 non-null int64
pickup_month        1458644 non-null int64
pickup_year         1458644 non-null int64
pickup_date         1458644 non-null object
dtypes: datetime64[ns](2), float64(4), int64(10), object(3)
memory usage: 211.4+ MB
```

Sanity check if there are any duplicate id's or duplicate records in the dataframe

In [12]:
```
print(df_train.id.duplicated().sum())
```

```
0
```

In [13]:
```
print(df_train.duplicated().sum())
```

```
0
```

Sanity check to cross check if all the trips are valid

In [14]:
```
sum(df_train.dropoff_datetime < df_train.pickup_datetime)
```

Out[14]:
```
0
```

Getting to know about the data

In [15]:
```
#We will try and analyze the target variable that is trip duration
print("Longest Trip Duration in secs: {}".format(np.max(df_train['trip_duration'].values)))
print("Shortest Trip Duration in secs: {}".format(np.min(df_train['trip_duration'].values)))
print("Average Trip Duration in secs: {}".format(np.mean(df_train['trip_duration'].values)))
```

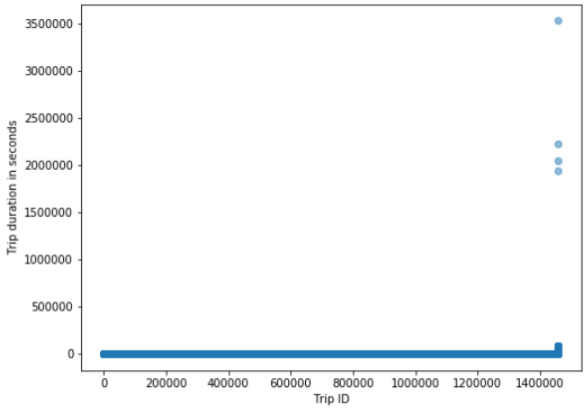
```
Longest Trip Duration in secs: 3526282
Shortest Trip Duration in secs: 1
Average Trip Duration in secs: 959.4922729603659
```

Outliers

An outlier is an observation point that is distant from other observations.An outlier may be due to variability in the measurement or it may indicate experimental error; the latter are sometimes excluded from the data set. An outlier can cause serious problems in statistical analyses. An outlier is an observation that appears to deviate markedly from other observations in the sample An outlier may indicate bad data. For example, the data may have been coded incorrectly or an experiment may not have been run correctly. If it can be determined that an outlying point is in fact erroneous, then the outlying value should be deleted from the analysis (or corrected if possible). In some cases, it may not be possible to determine if an outlying point is bad data. Outliers may be due to random variation or may indicate something scientifically interesting. In any event, we should not simply delete the outlying observation before a through investigation.

As we can observe that the smallest trip duration was of 1 seconds which is highly impossible and hence we will try and plot the outliers

In [16]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from matplotlib.pyplot import *
from matplotlib import cm
from matplotlib import animation
f = plt.figure(figsize=(8,6))
plt.scatter(range(len(df_train['trip_duration'])), np.sort(df_train['trip_duration']), alpha=0.5)
plt.xlabel('Trip ID')
plt.ylabel('Trip duration in seconds')
plt.show()
```



From the above scatter plot we can see that there are 4 outliers in the target variable. As the count of outliers is small we will not eliminate them

In [17]:
```
# Getting to know about passengers column
print("Maximum number of passengers on a trip : ", np.max(df_train['passenger_count'].values))
print("Minimum number of passengers on a trip : ", np.min(df_train['passenger_count'].values))
print("Average number of passengers on a trip : ", np.mean(df_train['passenger_count'].values))
passenger_num = df_train['passenger_count'].value_counts()
print(passenger_num)
```

```
Maximum number of passengers on a trip :  9
Minimum number of passengers on a trip :  0
Average number of passengers on a trip :  1.6645295219395548
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
```

As from the above statistics, we can observe the number of rides decreses as the count of passengers goes on increasing

In [18]:
```
df_train[df_train['passenger_count'] == 0]
```

Out[18]:

	id	vendor_id	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store
62744	id3917283	2	2016-06-06 16:39:09	2016-06-07 16:30:50	0	-73.776367	40.645248	-73.776360	40.645260	
136519	id3645383	2	2016-01-01 05:01:32	2016-01-01 05:01:36	0	-73.993134	40.757473	-73.993294	40.757538	
194288	id2840829	2	2016-02-21 01:33:52	2016-02-21 01:36:27	0	-73.946243	40.772903	-73.946770	40.774841	
217765	id3762593	1	2016-01-04 12:24:17	2016-01-04 13:01:48	0	-73.815224	40.700081	-73.950699	40.755222	
263809	id2154895	1	2016-05-23 23:37:37	2016-05-23 23:37:45	0	-73.861633	40.705029	-73.861633	40.705029	

As from the cell previous to this we saw that there were rides having 0 passenger count, to investigate further I selected only the records with zero passenger count records. We can observe that all the columns with respect to those records look fine and hence we will keep them

Checking for null values in dataset

In [19]:
```
df_train.isnull().sum()
```

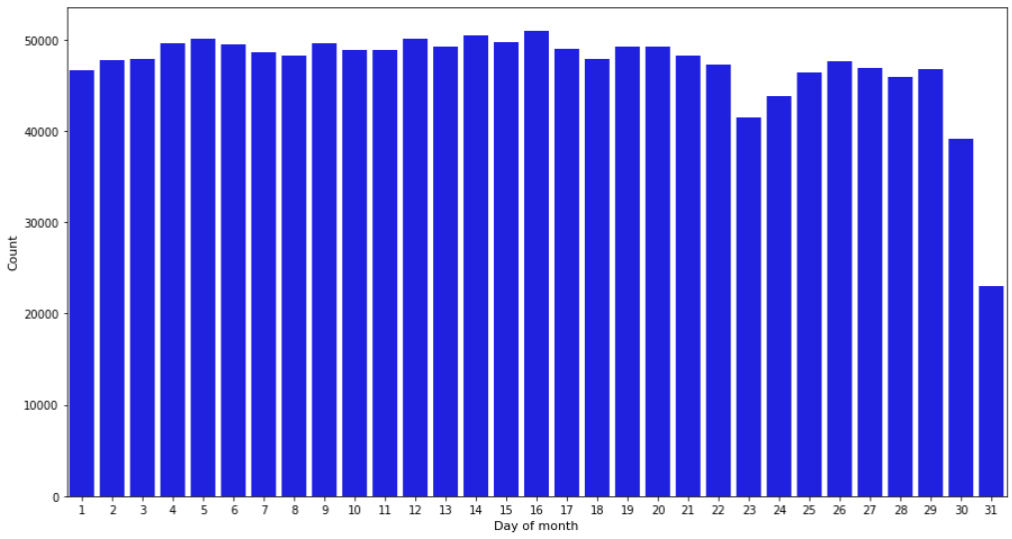
Out[19]:
```
id                    0
vendor_id             0
pickup_datetime       0
dropoff_datetime      0
passenger_count       0
pickup_longitude      0
pickup_latitude       0
dropoff_longitude     0
dropoff_latitude      0
store_and_fwd_flag    0
trip_duration         0
pickup_hour           0
pickup_minute         0
pickup_second         0
pickup_day            0
pickup_daymonth       0
pickup_month          0
pickup_year           0
pickup_date           0
dtype: int64
```

We see that training data contains 14 columns and ~1.4 Million data records with No nulls are present in out dataset.

In [20]:
```
#Count based on the day of the week
print(df_train['pickup_day'].value_counts())
```

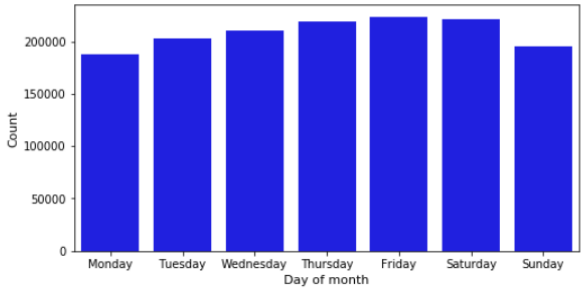
```
4    223533
5    220868
3    218574
2    210136
1    202749
```

In [21]:
```
#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 [22]:
```
#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()
```

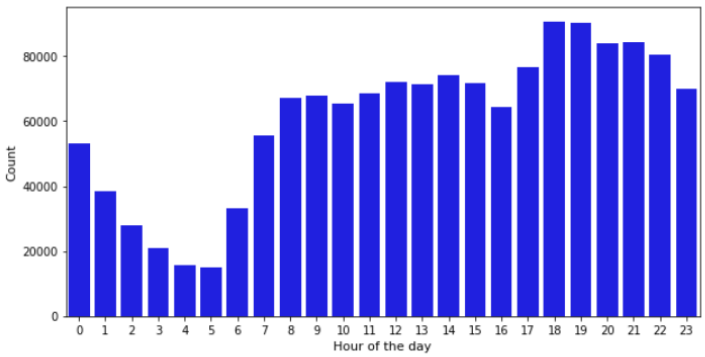


In the above figure I have plotted number of taxi trips during a certain day of the week. As we can see from above figure, the number of trips usually spikes, especially during the weekend i.e. Friday and Saturday, while it stays low on Monday.

In [23]:
```
print(df_train['pickup_month'].value_counts())
```

```
3    256189
4    251645
5    248487
2    238300
6    234316
1    229707
Name: pickup_month, dtype: int64
```

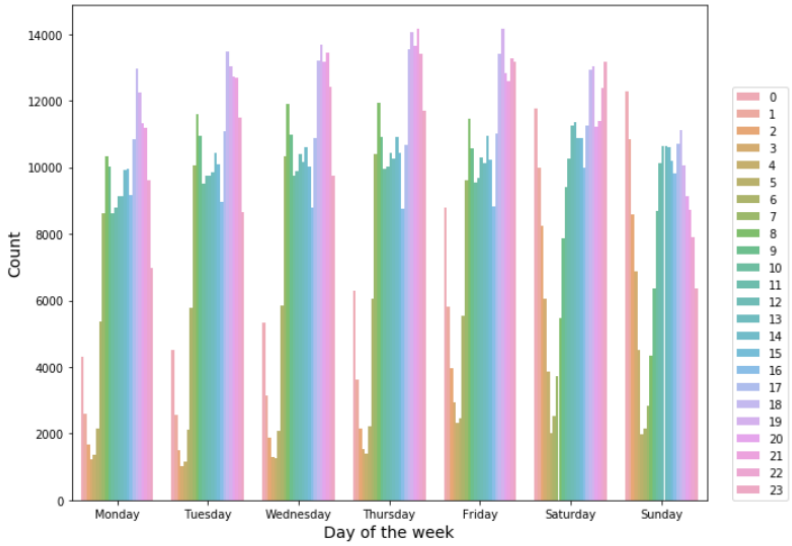
In [24]:
```
f = plt.figure(figsize=(10,5))
sns.countplot(x='pickup_hour', data=df_train, color = 'blue')
plt.xlabel('Hour of the day', fontsize=11)
plt.ylabel('Count', fontsize=11)
plt.show()
```



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

It is evident from the above plot that number of trips spike after 6pm till midnight

In [25]:
```
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

Reading the external data

In [26]:
```
#df_train_new = df_train.copy()
fast_route_1 = pd.read_csv(path+'/fastest_routes_train_part_1.csv')

fast_route_2 = pd.read_csv(path+'/fastest_routes_train_part_2.csv')

fast_route = pd.concat([fast_route_1,fast_route_2])
fast_route.head()
```

Out[26]:

	id	starting_street	end_street	total_distance	total_travel_time	number_of_steps	street_for_each_step	distance_per_step	travel_time
0	id2875421	Columbus Circle	East 65th Street	2009.1	164.9	5	Columbus Circle|Central Park West|65th Street ...	0|576.4|885.6|547.1|0	0|61.1|6
1	id2377394	2nd Avenue	Washington Square West	2513.2	332.0	6	2nd Avenue|East 13th Street|5th Avenue|Washing...	877.3|836.5|496.1|164.2|139.1|0	111.7|109|69.9|2
2	id3504673	Greenwich Street	Broadway	1779.4	235.8	4	Greenwich Street|Park Place|Broadway|Broadway	644.2|379.9|755.3|0	80.5|5(
3	id2181028	Broadway	West 81st Street	1614.9	140.1	5	Broadway|West 86th Street|Columbus Avenue|West...	617|427.4|412.2|158.3|0	56|36|3
4	id0801584	Lexington Avenue	West 31st Street	1393.5	189.4	5	Lexington Avenue|East 27th Street|Madison Aven...	18.9|311.9|313.3|749.4|0	6.3|42.9|4

We will check if the OSRM data is clean or not. First step is we will check for duplicate id's ad records

In [27]:
```
print(fast_route.id.duplicated().sum())
```

```
0
```

In [28]:
```
fast_route_new = fast_route[['id','total_distance','total_travel_time','number_of_steps']]
fast_route_new.head()
```

Out[28]:

	id	total_distance	total_travel_time	number_of_steps
0	id2875421	2009.1	164.9	5
1	id2377394	2513.2	332.0	6
2	id3504673	1779.4	235.8	4
3	id2181028	1614.9	140.1	5
4	id0801584	1393.5	189.4	5

We will join the the fastest route data with the original dataframe

In [29]:
```
df_train_new = df_train.copy()
train_df = pd.merge(df_train_new, fast_route_new, on='id', how='left')
train_df.head()
```

Out[29]:

	id	vendor_id	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd
0	id2875421	2	2016-03-14 17:24:55	2016-03-14 17:32:30	1	-73.982155	40.767937	-73.964630	40.765602	
1	id2377394	1	2016-06-12 00:43:35	2016-06-12 00:54:38	1	-73.980415	40.738564	-73.999481	40.731152	
2	id3858529	2	2016-01-19 11:35:24	2016-01-19 12:10:48	1	-73.979027	40.763939	-74.005333	40.710087	
3	id3504673	2	2016-04-06 19:32:31	2016-04-06 19:39:40	1	-74.010040	40.719971	-74.012268	40.706718	
4	id2181028	2	2016-03-26 13:30:55	2016-03-26 13:38:10	1	-73.973053	40.793209	-73.972923	40.782520	

5 rows × 22 columns

In [30]:
```
print(train_df.shape)
```

```
(1458644, 22)
```

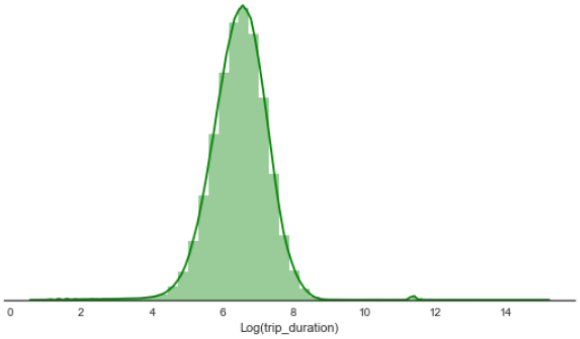
In [31]:
```
train_df.isnull().sum()
```

Out[31]:
```
id                    0
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
pickup_hour           0
pickup_minute         0
pickup_second         0
pickup_day            0
pickup_daymonth       0
pickup_month          0
pickup_year           0
pickup_date           0
total_distance        1
total_travel_time     1
number_of_steps       1
dtype: int64
```

Visualize the trip duration given using log-scale distplot in sns

We want to predict trip_duration of the test set, so we first check what kind of trips durations are present in the dataset. First, I plotted it on a plain scale and not on a log scale, and some of the records have very long trip durations ~100 hours. Such long trips are making all another trip invisible in the histogram on plain scale => We go ahead with the log scale. Another reason of using the log scale for visualizing trip-duration on the log scale is that this competition uses RMSLE matrix so it would make sense to visualize the target variable in log scale only.

In [32]:
```
import warnings
warnings.filterwarnings('ignore')
sns.set(style="white")
f, axes = plt.subplots(1, 1, figsize=(9, 5), sharex=True)
sns.despine(left=True)
sns.distplot(np.log(train_df['trip_duration'].values+1),
             axlabel = 'Log(trip_duration)',
             label = 'log(trip_duration)',
             bins = 50,
             color="g")
plt.setp(axes, yticks=[])
plt.show()
```



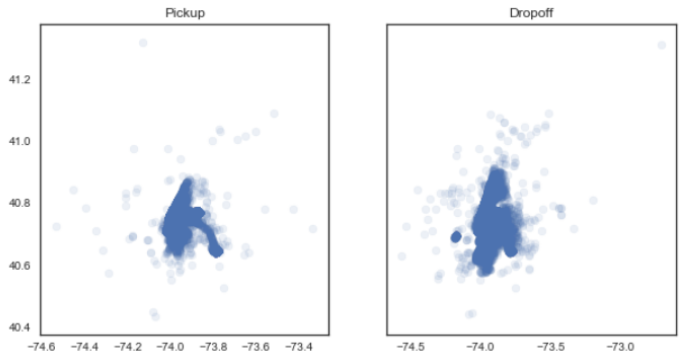
Note It is clear with the above histogram and kernel density plot that the trip-durations are like Gaussian and few trips have very large duration.

In [34]:
```
n=100000
f, (ax1,ax2) = plt.subplots(1, 2, sharey=True, figsize=(10,5))

ax1.scatter(df_train.pickup_longitude[:n], df_train.pickup_latitude[:n], alpha = 0.1)
ax1.set_title('Pickup')

ax2.scatter(df_train.dropoff_longitude[:n], df_train.dropoff_latitude[:n], alpha = 0.1)
ax2.set_title('Dropoff')
```

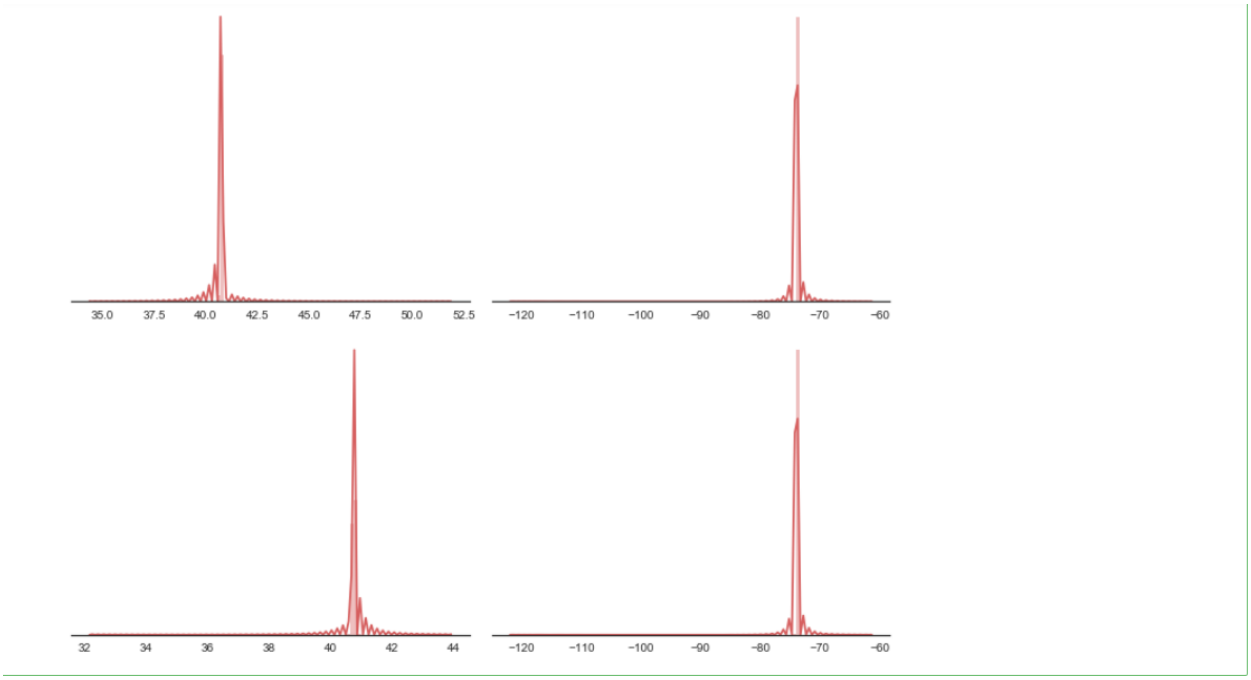
Out[34]: Text(0.5,1,'Dropoff')



While most of the trips are e^4 = 1 minute to e^8 ~ 60 minutes. and probably are taken inside Manhattan or in new york only. Let's check the lat-long distributions are then used them to have a heat map kind of view of given lat-longs.

In [35]:
```
#*********************************************************************************
#*    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
#*
#*********************************************************************************/

sns.set(style="white", palette="muted", color_codes=True)
f, axes = plt.subplots(2,2,figsize=(10, 8), sharex=False, sharey = False)
sns.despine(left=True)
sns.distplot(df_train['pickup_latitude'].values, label = 'pickup_latitude',color="r",bins = 100, ax=axes[0,0])
sns.distplot(df_train['pickup_longitude'].values, label = 'pickup_longitude',color="r",bins =100, ax=axes[0,1])
sns.distplot(df_train['dropoff_latitude'].values, label = 'dropoff_latitude',color="r",bins =100, ax=axes[1, 0])
sns.distplot(df_train['dropoff_longitude'].values, label = 'dropoff_longitude',color="r",bins =100, ax=axes[1, 1])
plt.setp(axes, yticks=[])
plt.tight_layout()
plt.show()
```

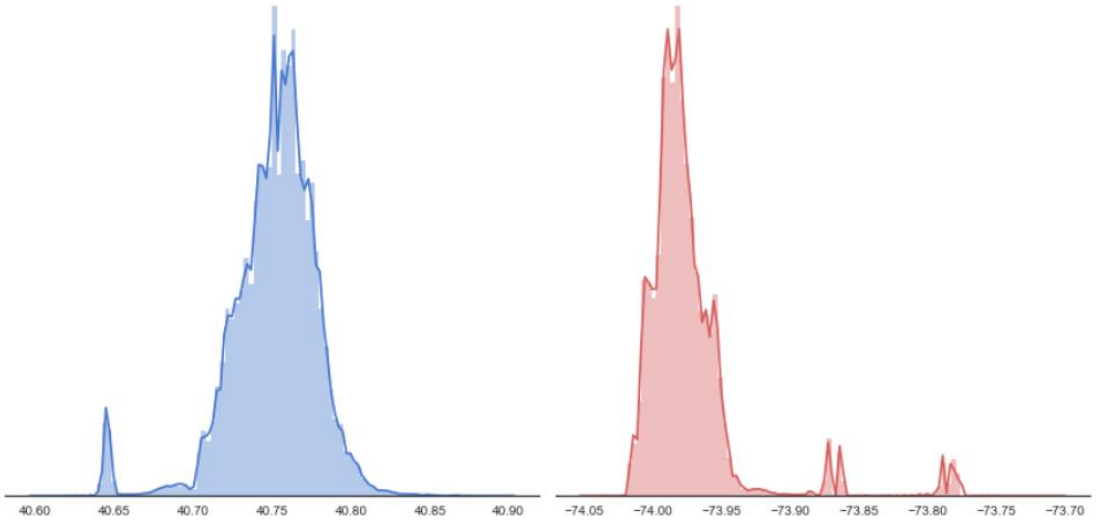


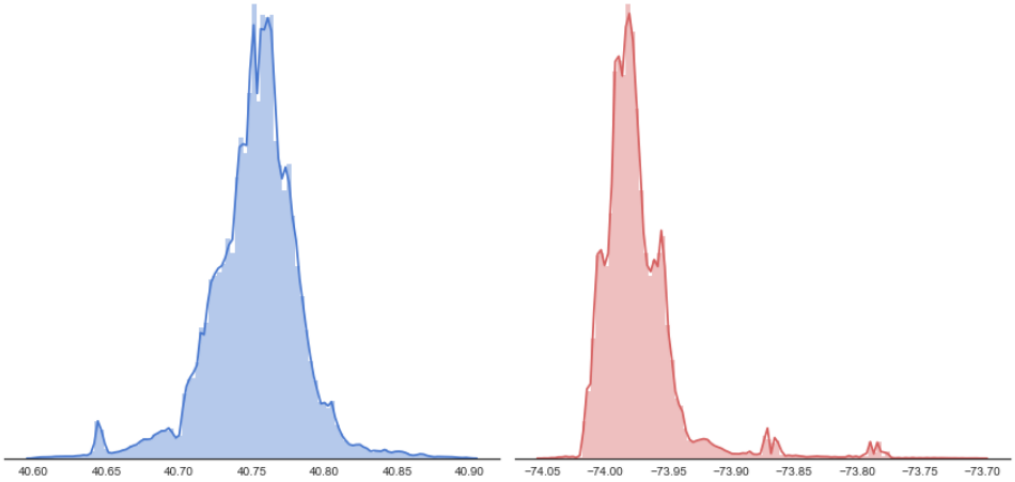
Note From the plot above it is clear that pick and drop latitude are centered around 40 to 41, and longitude are situated around -74 ton-73. We are not getting any histogram kind of plots when we are plotting lat-long as the distplot function of sns is getting affected by outliers, trips which are very far from each other like lat 32 to lat 44, are taking very long time, and have affected this plot such that it is coming off as a spike. Let's remove those large duration trip by using a cap on lat-long and visualize the distributions of latitude and longitude given to us.

In [36]:
```
#*********************************************************************************
#*    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
#*
#*********************************************************************************/

df = df_train.loc[(df_train.pickup_latitude > 40.6) & (df_train.pickup_latitude < 40.9)]
df = df.loc[(df.dropoff_latitude>40.6) & (df.dropoff_latitude < 40.9)]
df = df.loc[(df.dropoff_longitude > -74.05) & (df.dropoff_longitude < -73.7)]
df = df.loc[(df.pickup_longitude > -74.05) & (df.pickup_longitude < -73.7)]
df_new = df.copy()
sns.set(style="white", palette="muted", color_codes=True)
f, axes = plt.subplots(2,2,figsize=(12, 12), sharex=False, sharey = False)#
sns.despine(left=True)
sns.distplot(df_new['pickup_latitude'].values, label = 'pickup_latitude',color="b",bins = 100, ax=axes[0,0])
sns.distplot(df_new['pickup_longitude'].values, label = 'pickup_longitude',color="r",bins =100, ax=axes[0,1])
sns.distplot(df_new['dropoff_latitude'].values, label = 'dropoff_latitude',color="b",bins =100, ax=axes[1, 0])
sns.distplot(df_new['dropoff_longitude'].values, label = 'dropoff_longitude',color="r",bins =100, ax=axes[1, 1])
plt.setp(axes, yticks=[])
plt.tight_layout()
print(df.shape[0], df_new.shape[0])
plt.show()
```

```
1452385 1452385
```





We put the following caps on lat-long -

-latitude should be between 40.6 to 40.9

-longitude should be between -74.05 to -73.70

We get that the distribution spikes becomes as distribution in distplot (distplot is a histogram plot in seaborn package), we can see that most of the trips are getting concentrated between these lat-long only. Let's plot them on an empty image.

Heatmap of coordinates

We have taken an empty image and make it a color it black so that we can see colors where the lat-longs are falling. To visualize we need to consider each point of this image as a point represented by lat-long, to achieve that we will bring the lat-long to image coordinate range and then take a summary of lat-long and their count, assign a different color for different count range.

Why 255?
RGB (Red, Green, Blue) are 8 bit each. The range for each individual colour is 0-255 (as 2^8 = 256 possibilities). The combination range is 256256256. It really comes down to math and getting a value between 0-1. Since 255 is the maximum value, dividing by 255 expresses a 0-1 representation. Each channel (Red, Green, and Blue are each channels) is 8 bits, so they are each limited to 256, in this case 255 since 0 is included

00000000 - 00000000 - 00000000 would be the RGB value of black; none of any colour.

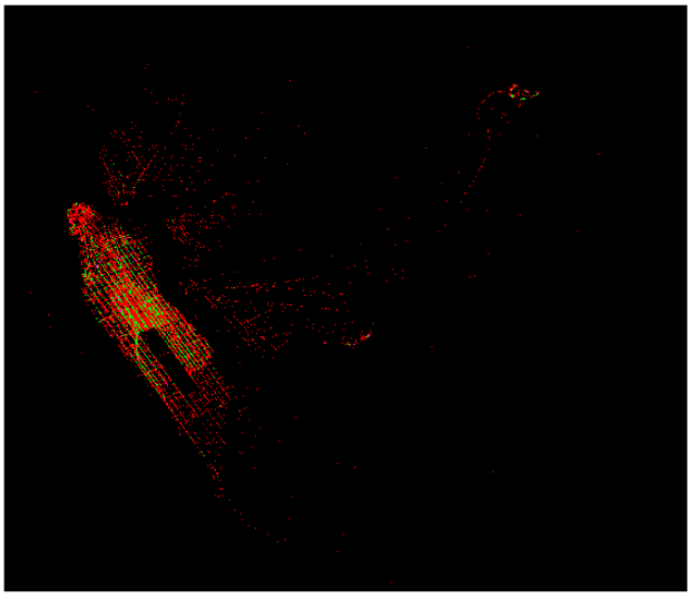
11111111 - 11111111 - 11111111 would be white; all of everything.

In [37]:
```
#*********************************************************************************
#*    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
#*
#*********************************************************************************/

rgb = np.zeros((3000, 3500, 3), dtype=np.uint8)
rgb[..., 0] = 0
rgb[..., 1] = 0
rgb[..., 2] = 0
df_new['pick_lat_new'] = list(map(int, (df_new['pickup_latitude'] - (40.6000))*10000))
df_new['drop_lat_new'] = list(map(int, (df_new['dropoff_latitude'] - (40.6000))*10000))
df_new['pick_lon_new'] = list(map(int, (df_new['pickup_longitude'] - (-74.050))*10000))
df_new['drop_lon_new'] = list(map(int,(df_new['dropoff_longitude'] - (-74.050))*10000))

summary_plot = pd.DataFrame(df_new.groupby(['pick_lat_new', 'pick_lon_new'])['id'].count())

summary_plot.reset_index(inplace = True)
summary_plot.head(120)
lat_list = summary_plot['pick_lat_new'].unique()
for i in lat_list:
    lon_list = summary_plot.loc[summary_plot['pick_lat_new']==i]['pick_lon_new'].tolist()
    unit = summary_plot.loc[summary_plot['pick_lat_new']==i]['id'].tolist()
    for j in lon_list:
        a = unit[lon_list.index(j)]
        if (a//50) >0:
            rgb[i][j][0] = 255
            rgb[i,j, 1] = 0
            rgb[i,j, 2] = 255
        elif (a//10)>0:
            rgb[i,j, 0] = 0
            rgb[i,j, 1] = 255
            rgb[i,j, 2] = 0
        else:
            rgb[i,j, 0] = 255
            rgb[i,j, 1] = 0
            rgb[i,j, 2] = 0
fig, ax = plt.subplots(nrows=1,ncols=1,figsize=(10,10))
ax.imshow(rgb, cmap = 'hot')
ax.set_axis_off()
```



To get an idea about the actual pickup and dropoff locations, we will take only the location co-ordinates of pickup and dropoff from the original dataframe into another dataframe

Scatter Plot

A scatter plot is a type of plot or mathematical diagram using Cartesian coordinates to display values for typically two variables for a set of data. If the points are color-coded, one additional variable can be displayed. The data are displayed as a collection of points, each having the value of one variable determining the position on the horizontal axis and the value of the other variable determining the position on the vertical axis. A scatter plot can be used either when one continuous variable that is under the control of the experimenter and the other depends on it or when both continuous variables are independent.

We will use scatter plot to plot the co-ordinates, to get an clear idea about the pickup and dropoff

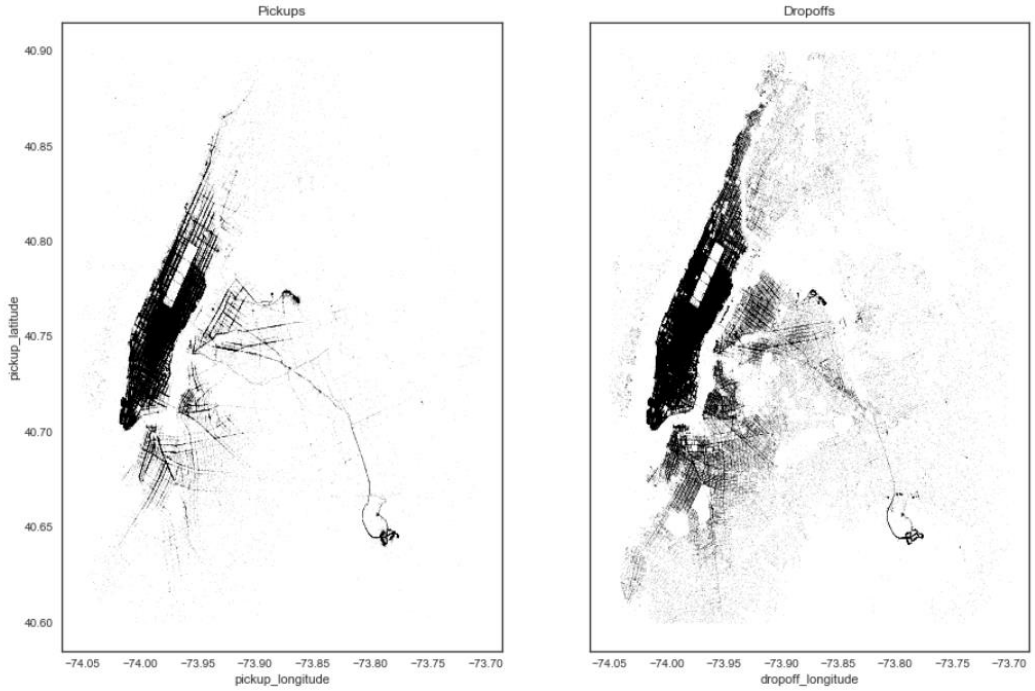
In [40]:
```
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True,figsize=(15,10))

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

Out[40]:
```
Text(0.5,1,'Dropoffs')
```

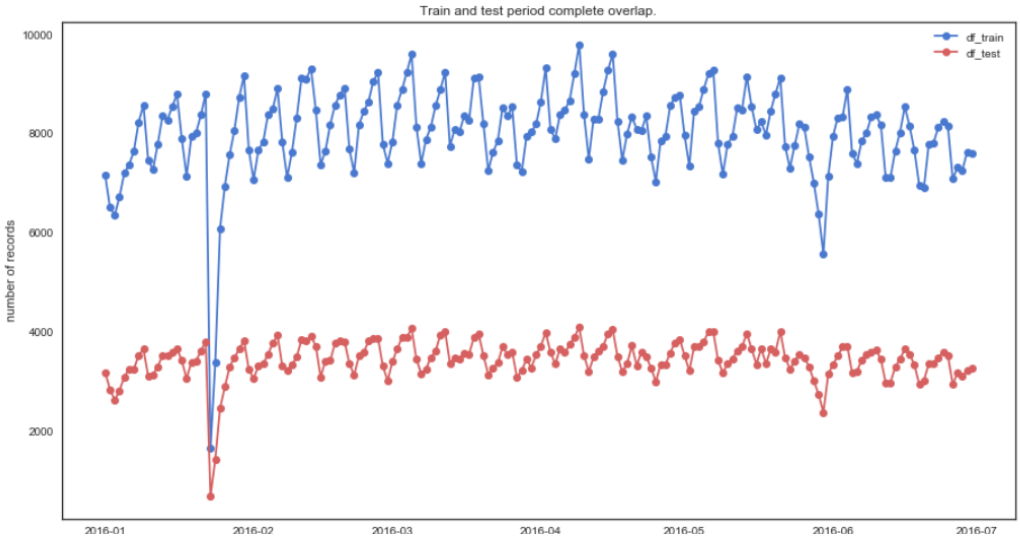
Out[40]: Text(0.5,1,'Dropoffs')



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

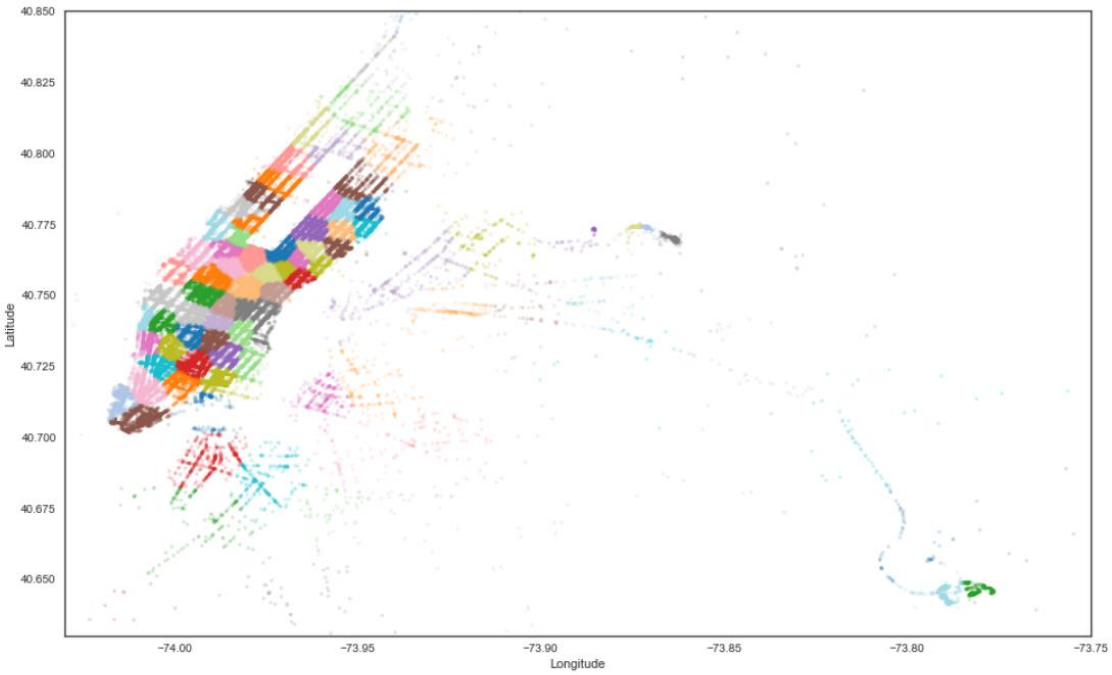
K-means clustering is a type of unsupervised learning, which is used when you have unlabeled data (i.e., data without defined categories or groups). The goal of this algorithm is to find groups in the data, with the number of groups represented by the variable K. The algorithm works iteratively to assign 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()
```



XGBoost

XGBoost is an algorithm that has recently been dominating applied machine learning for structured or tabular data. XGBoost is an implementation of gradient boosted decision trees designed for speed and performance. Why Use XGBoost? The two reasons to use XGBoost are also the two goals of the project:

Execution Speed. Model Performance.

1. XGBoost Execution Speed Generally, XGBoost is fast. Really fast when compared to other implementations of gradient boosting.
2. XGBoost Model Performance XGBoost dominates structured or tabular datasets on classification and regression predictive modeling problems.

How to install xgboost module in Python

1. Download the Appropriate .whl file for your environment from https://www.lfd.uci.edu/~gohlke/pythonlibs/#xgboost for me as I am using 64-bit Python 3.6, I downloaded xgboost-0.71-cp36-cp36m-win_amd64.whl file. My python is 64-bit and not my system, so make sure you check that before downloading the .whl file
2. Open your command prompt and cd into the downloaded folder.
3. Now simply issue the pip install command to the downloaded .whl file like so:

   pip install xgboost-0.71-cp36-cp36m-win_amd64.whl
4. If you have downloaded the wrong file it will give you an error saying whl file not supported for this platform

In [1]:
```
import pandas as pd
import numpy as np
import datetime
from haversine import haversine
import math
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from matplotlib.pyplot import *
from matplotlib import cm
from matplotlib import animation
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
```

Feature Extraction

In machine learning, pattern recognition and in image processing, feature extraction starts from an initial set of measured data and builds derived values (features) intended to be informative and non-redundant, facilitating the subsequent learning and generalization steps, and in some cases leading to better human interpretations. Feature extraction is related to dimensionality reduction.

When the input data to an algorithm is too large to be processed and it is suspected to be redundant (e.g. the same measurement in both feet and meters, or the repetitiveness of images presented as pixels), then it can be transformed into a reduced set of features (also named a feature vector). Determining a subset of the initial features is called feature selection.The selected features are expected to contain the relevant information from the input data, so that the desired task can be performed by using this reduced representation instead of the complete initial data

Principal Component Analysis

Sometimes data are collected on a large number of variables from a single population. With a large number of variables, the dispersion matrix may be too large to study and interpret properly. There would be too many pairwise correlations between the variables to consider. Graphical displays may also not be particularly helpful when the data set is very large. With 12 variables, for example, there will be more than 200 three-dimensional scatterplots. To interpret the data in a more meaningful form, it is necessary to reduce the number of variables to a few, interpretable linear combinations of the data. Each linear combination will correspond to a principal component.**We use PCA to transform longitude and latitude coordinates. In this case it is not about dimension reduction since we transform 2D-> 2D. The rotation could help for decision tree splits.**

In [9]:
```
#Stack arrays in sequence vertically (row wise).
coords = np.vstack((train_df[['pickup_latitude', 'pickup_longitude']].values,
                    train_df[['dropoff_latitude', 'dropoff_longitude']].values,
                    train_df[['pickup_latitude', 'pickup_longitude']].values,
                    train_df[['dropoff_latitude', 'dropoff_longitude']].values))

pca = PCA().fit(coords)

train_df['pickup_pca0'] = pca.transform(train_df[['pickup_latitude', 'pickup_longitude']])[:,0]
train_df['pickup_pca1'] = pca.transform(train_df[['pickup_latitude', 'pickup_longitude']])[:,1]
train_df['dropoff_pca0'] = pca.transform(train_df[['dropoff_latitude', 'dropoff_longitude']])[:,0]
train_df['dropoff_pca1'] = pca.transform(train_df[['dropoff_latitude', 'dropoff_longitude']])[:,1]

test_df['pickup_pca0'] = pca.transform(test_df[['pickup_latitude', 'pickup_longitude']])[:,0]
test_df['pickup_pca1'] = pca.transform(test_df[['pickup_latitude', 'pickup_longitude']])[:,1]
test_df['dropoff_pca0'] = pca.transform(test_df[['dropoff_latitude', 'dropoff_longitude']])[:,0]
test_df['dropoff_pca1'] = pca.transform(test_df[['dropoff_latitude', 'dropoff_longitude']])[:,1]
```

Haversine Distance: The haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitudes.

Manhattan Distance: The distance between two points in a grid based on a strictly horizontal and/or vertical path (that is, along the grid lines), as opposed to the diagonal or "as the crow flies" distance.

Bearing Distance: The bearing (or azimuth) is the compass direction to travel from the starting point, and must be within the range 0 to 360. 0 represents north, 90 is east, 180 is south and 270 is west.

In [11]:
```
def haversine_array(lat1, lng1, lat2, lng2):
    lat1, lng1, lat2, lng2 = map(np.radians, (lat1, lng1, lat2, lng2)) #applies a function to all the items in an input_list
    AVG_EARTH_RADIUS = 6371  # in km
    lat = lat2 - lat1
    lng = lng2 - lng1
    d = np.sin(lat * 0.5) ** 2 + np.cos(lat1) * np.cos(lat2) * np.sin(lng * 0.5) ** 2
    h = 2 * AVG_EARTH_RADIUS * np.arcsin(np.sqrt(d))
    return h

def dummy_manhattan_distance(lat1, lng1, lat2, lng2):
    a = haversine_array(lat1, lng1, lat1, lng2)
    b = haversine_array(lat1, lng1, lat2, lng1)
    return a + b

def bearing_array(lat1, lng1, lat2, lng2):
    AVG_EARTH_RADIUS = 6371  # in km
    lng_delta_rad = np.radians(lng2 - lng1)
    lat1, lng1, lat2, lng2 = map(np.radians, (lat1, lng1, lat2, lng2)) #applies a function to all the items in an input_list
    y = np.sin(lng_delta_rad) * np.cos(lat2)
    x = np.cos(lat1) * np.sin(lat2) - np.sin(lat1) * np.cos(lat2) * np.cos(lng_delta_rad)
    return np.degrees(np.arctan2(y, x))
```

We will apply the above three functions to the columns in train and test dataframe

We will apply the above three functions to the columns in train and test dataframe

In [12]:
```
train_df.loc[:, 'distance_haversine'] = haversine_array(train_df['pickup_latitude'].values, train_df['pickup_longitude'].values,
train_df.loc[:, 'distance_manhattan'] = dummy_manhattan_distance(train_df['pickup_latitude'].values, train_df['pickup_longitude']
train_df.loc[:, 'direction'] = bearing_array(train_df['pickup_latitude'].values, train_df['pickup_longitude'].values, train_df['d
train_df.loc[:, 'pca_manhattan'] = np.abs(train_df['dropoff_pca1'] - train_df['pickup_pca1']) + np.abs(train_df['dropoff_pca0'] -
```

In [13]:
```
test_df.loc[:, 'distance_haversine'] = haversine_array(test_df['pickup_latitude'].values, test_df['pickup_longitude'].values, tes
test_df.loc[:, 'distance_manhattan'] = dummy_manhattan_distance(test_df['pickup_latitude'].values, test_df['pickup_longitude'].va
test_df.loc[:, 'direction'] = bearing_array(test_df['pickup_latitude'].values, test_df['pickup_longitude'].values, test_df['dropo
test_df.loc[:, 'pca_manhattan'] = np.abs(test_df['dropoff_pca1'] - test_df['pickup_pca1']) + np.abs(test_df['dropoff_pca0'] - tes
```

In [14]:
```
train_df.loc[:, 'center_latitude'] = (train_df['pickup_latitude'].values + train_df['dropoff_latitude'].values)/2
train_df.loc[:, 'center_longitude'] = (train_df['pickup_longitude'].values + train_df['dropoff_longitude'].values)/2
test_df.loc[:, 'center_latitude'] = (test_df['pickup_latitude'].values + test_df['dropoff_latitude'].values)/2
test_df.loc[:, 'center_longitude'] = (test_df['pickup_longitude'].values + test_df['dropoff_longitude'].values)/2
```

Extract all the datetime features

In [15]:
```
train_df.loc[:, 'pickup_weekday'] = train_df['pickup_datetime'].dt.weekday
train_df.loc[:, 'pickup_hour_weekofyear'] = train_df['pickup_datetime'].dt.weekofyear
train_df.loc[:, 'pickup_hour'] = train_df['pickup_datetime'].dt.hour
train_df.loc[:, 'pickup_minute'] = train_df['pickup_datetime'].dt.minute
train_df.loc[:, 'pickup_dt'] = (train_df['pickup_datetime'] - train_df['pickup_datetime'].min()).dt.total_seconds()
train_df.loc[:, 'pickup_week_hour'] = train_df['pickup_weekday'] * 24 + train_df['pickup_hour']
```

In [16]:
```
test_df.loc[:, 'pickup_weekday'] = test_df['pickup_datetime'].dt.weekday
test_df.loc[:, 'pickup_hour_weekofyear'] = test_df['pickup_datetime'].dt.weekofyear
test_df.loc[:, 'pickup_hour'] = test_df['pickup_datetime'].dt.hour
test_df.loc[:, 'pickup_minute'] = test_df['pickup_datetime'].dt.minute
test_df.loc[:, 'pickup_dt'] = (test_df['pickup_datetime'] - test_df['pickup_datetime'].min()).dt.total_seconds()
test_df.loc[:, 'pickup_week_hour'] = test_df['pickup_weekday'] * 24 + test_df['pickup_hour']
```

Extract the Speed feature using the Haversine Distance and trip duration

We will use the basic speed is equal to distance upon time formula

In [17]:
```
train_df.loc[:, 'avg_speed_h'] = 1000 * train_df['distance_haversine']/train_df['trip_duration']
train_df.loc[:, 'avg_speed_m'] = 1000 * train_df['distance_manhattan']/train_df['trip_duration']
```

In [18]:
```
test_df.loc[:, 'avg_speed_h'] = 1000 * train_df['distance_haversine']/train_df['trip_duration']
test_df.loc[:, 'avg_speed_m'] = 1000 * train_df['distance_manhattan']/train_df['trip_duration']
```

In [19]:
```
train_df.head()
```

Out[19]:

	id	vendor_id	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd
0	id2875421	2	2016-03-14 17:24:55	2016-03-14 17:32:30	1	-73.982155	40.767937	-73.964630	40.765602	
1	id2377394	1	2016-06-12 00:43:35	2016-06-12 00:54:38	1	-73.980415	40.738564	-73.999481	40.731152	
2	id3858529	2	2016-01-19 11:35:24	2016-01-19 12:10:48	1	-73.979027	40.763939	-74.005333	40.710087	
3	id3504673	2	2016-04-06 19:32:31	2016-04-06 19:39:40	1	-74.010040	40.719971	-74.012268	40.706718	
4	id2181028	2	2016-03-26 13:30:55	2016-03-26 13:38:10	1	-73.973053	40.793209	-73.972923	40.782520	

5 rows × 34 columns

Adding holiday as a feature

We will add all the US holidays as holiday feature from the available library

In [20]:
```
import holidays
from datetime import date, datetime
us_holidays = holidays.UnitedStates()
```

In [21]:
```
#If that particular day is a holiday then it will be marked as 1 otherwise 0
train_df.loc[:, 'holiday'] = train_df.pickup_datetime.apply(lambda x: 1 if x in us_holidays else 0)
```

In [22]:
```
#Same for the test data
test_df.loc[:, 'holiday'] = train_df.pickup_datetime.apply(lambda x: 1 if x in us_holidays else 0)
```

Adding weather as a feature

First we will import the weather data into a dataframe

In [23]:
```
import os
path = os.getcwd()+'\weather_data_nyc_centralpark_2016.csv'
weather_df = pd.read_csv(path)
weather_df.head()
```

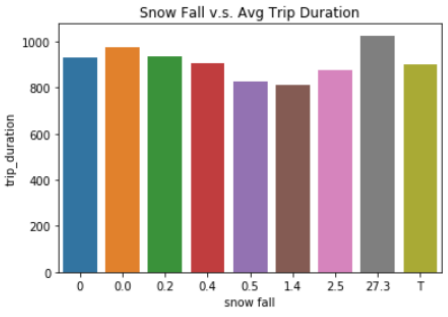
Out[23]:

	date	maximum temperature	minimum temperature	average temperature	precipitation	snow fall	snow depth
0	1-1-2016	42	34	38.0	0.00	0.0	0
1	2-1-2016	40	32	36.0	0.00	0.0	0
2	3-1-2016	45	35	40.0	0.00	0.0	0
3	4-1-2016	36	14	25.0	0.00	0.0	0
4	5-1-2016	29	11	20.0	0.00	0.0	0

In [24]:
```
weather_df['DATE'] = pd.to_datetime(weather_df.date) #as the format of date in data varies we will convert it into one format
weather_df['pickup_date'] = weather_df['DATE'].dt.date #rename the column name
del weather_df['date']
```

In [26]:
```
#We will try and visualize how weather affected the trip duration
a = train_df.groupby(['snow fall']).mean()['trip_duration'].reset_index()
print(a)
sns.barplot(x="snow fall", y='trip_duration', data=a)
plt.xlabel('snow fall')
plt.ylabel('trip_duration')
plt.title('Snow Fall v.s. Avg Trip Duration')
plt.show()
```

```
  snow fall  trip_duration
0         0     931.256312
1       0.0     973.658589
2       0.2     934.412834
3       0.4     906.369139
4       0.5     828.804774
5       1.4     813.655093
6       2.5     875.932462
7      27.3    1026.549757
8         T     899.071886
```



From the above graph it is clearly visible that during snowfall the drip duration of the taxi ride increased

From the above graph it is clearly visible that during snowfall the drip duration of the taxi ride increased

In [27]:
```
#We will create a lookup for the 'T' value in snowfall column so as to convert it into float 0.01
lookup = {"snow fall": {"T": 0.01},
          "snow depth": {"T": 0.01},
          'precipitation': {"T": 0.01}}
train_df.replace(lookup, inplace=True)
train_df['snow fall'] = train_df['snow fall'].astype('float')
train_df['snow depth'] = train_df['snow depth'].astype('float')
train_df['precipitation'] = train_df['precipitation'].astype('float')
```

In [28]:
```
#We will do the same thing for test data
test_df.replace(lookup, inplace=True)
test_df['snow fall'] = test_df['snow fall'].astype('float')
test_df['snow depth'] = test_df['snow depth'].astype('float')
test_df['precipitation'] = test_df['precipitation'].astype('float')
```

We will add the names of all the columns in train into a list and the list of not required columns to exclude list

In [33]:
```
feature_names = list(train_df.columns)
exclude = ['id','pickup_datetime','dropoff_datetime','trip_duration','pickup_date','dropoff_date','DATE']
feature_names = [f for f in train_df.columns if f not in exclude]
train_df[feature_names].count()
```

Out[33]:

vendor_id	1458644
passenger_count	1458644
pickup_longitude	1458644
pickup_latitude	1458644
dropoff_longitude	1458644
dropoff_latitude	1458644
store_and_fwd_flag	1458644
total_distance	1458644
total_travel_time	1458644
number_of_steps	1458644
pickup_pca0	1458644
pickup_pca1	1458644
dropoff_pca0	1458644
dropoff_pca1	1458644
distance_haversine	1458644
distance_manhattan	1458644
direction	1458644
pca_manhattan	1458644
center_latitude	1458644
center_longitude	1458644

In [34]:
```
print(feature_names)
```

```
['vendor_id', 'passenger_count', 'pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude', 'store_and_fwd
_flag', 'total_distance', 'total_travel_time', 'number_of_steps', 'pickup_pca0', 'pickup_pca1', 'dropoff_pca0', 'dropoff_pca1',
'distance_haversine', 'distance_manhattan', 'direction', 'pca_manhattan', 'center_latitude', 'center_longitude', 'pickup_weekda
y', 'pickup_hour_weekofyear', 'pickup_hour', 'pickup_minute', 'pickup_dt', 'pickup_week_hour', 'avg_speed_h', 'avg_speed_m', 'h
oliday', 'maximum temperature', 'minimum temperature', 'average temperature', 'precipitation', 'snow fall', 'snow depth']
```

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
```

RandomForestRegressor

max_depth
: integer or None, optional (default=None) The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split
int, float, optional (default=2) The minimum number of samples required to split an internal node:

min_samples_leaf
int, float, optional (default=1) The minimum number of samples required to be at a leaf node:

n_estimators
: The number of trees in the forest.

max_features
: The number of features to consider when looking for the best split

You can vary the features and see it the rmse value decreases

In [46]:
```
rf_model = RandomForestRegressor(max_depth = 20 , min_samples_split= 2,
                                 min_samples_leaf=5, n_estimators =10, max_features =8)
rf_model.fit(train_x,train_y)
```

Out[46]:
```
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)
```

In [47]:
```
#Predicting on validation set
predictions = rf_model.predict(val_x)
```

In [48]:
```
#RMSLE error
from sklearn.metrics import mean_squared_error
#tree_pred = pd.DataFrame(predictions)
dec_mse = mean_squared_error(predictions, val_y)
rmse = np.sqrt(dec_mse)
print(rmse)
```

```
0.11033131704917128
```

Lasso & Ridge Regularization

Ridge and Lasso regression are powerful techniques generally used for creating parsimonious models in presence of a 'large' number of features. Here 'large' can typically mean either of two things: 1.Large enough to enhance the tendency of a model to overfit (as low as 10 variables might cause overfitting) 2.Large enough to cause computational challenges. With modern systems, this situation might arise in case of millions or billions of features

Though Ridge and Lasso might appear to work towards a common goal, the inherent properties and practical use cases differ substantially. If you've heard of them before, you must know that they work by penalizing the magnitude of coefficients of features along with minimizing the error between predicted and actual observations. These are called 'regularization' techniques. The key difference is in how they assign penalty to the coefficients:

Ridge Regression: Performs L2 regularization, i.e. adds penalty equivalent to square of the magnitude of coefficients Minimization objective = LS Obj + α * (sum of square of coefficients)

Lasso Regression: Performs L1 regularization, i.e. adds penalty equivalent to absolute value of the magnitude of coefficients Minimization objective = LS Obj + α * (sum of absolute value of coefficients)

param_grid :
dict of string to sequence, or sequence of such

The parameter grid to explore, as a dictionary mapping estimator parameters to sequences of allowed values.

An empty dict signifies default parameters.

A sequence of dicts signifies a sequence of grids to search, and is useful to avoid exploring parameter combinations that make no sense or have no effect. See the examples below.

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
```

Ensemble Learning

An ensemble contains a number of learners which are usually called base learners. The generalization ability of an ensemble is usually much stronger than that of base learners. Actually, ensemble learning is appealing because that it is able to boost weak learners which are slightly better than random guess to strong learners which can make very accurate predictions. So, "base learners" are also referred as "weak learners". It is noteworthy, however, that although most theoretical analyses work on weak learners, base learners used in practice are not necessarily weak since using not-so-weak base learners often results in better performance.

Base learners are usually generated from training data by a base learning algorithm which can be decision tree, neural network or other kinds of machine learning algorithms. Most ensemble methods use a single base learning algorithm to produce homogeneous base learners, but there are also some methods which use multiple learning algorithms to produce heterogeneous learners. In the latter case there is no single base learning algorithm and thus, some people prefer calling the learners individual learners or component learners to "base learners", while the names "individual learners" and "component learners" can also be used for homogeneous base learners.

To see if we get better performance we will combine Random Forest, Lasso, Ridge and XGBRegressor and check

XGBRegressor

objective : string or callable Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below)

n_estimators : int Number of boosted trees to fit

subsample : float Subsample ratio of the training instance.

max_depth : int Maximum tree depth for base learners

min_child_weight : int Minimum sum of instance weight(hessian) needed in a child.

reg_alpha : float (xgb's alpha) L1 regularization term on weights reg_lambda : float (xgb's lambda) L2 regularization term on weights

In [57]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Ridge, Lasso
from xgboost.sklearn import XGBRegressor

xgb_model = XGBRegressor(objective='reg:linear', n_estimators=10, subsample=0.9, max_depth = 6, min_child_weight = 3
                         ,reg_lambda = 2, learning_rate = 0.05)
rf_model = RandomForestRegressor(max_depth = 20 , min_samples_split= 2, min_samples_leaf=5, n_estimators =10, max_features =8)
```

In [58]:
```
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])
```

In [59]:
```
averaged_model.fit(train_x,train_y)
predicitons_ensemble = averaged_model.predict(val_x)
```

In [60]:
```
dec_mse_ensemble = mean_squared_error(predicitons_ensemble, val_y)
rmse_ensemble = np.sqrt(dec_mse_ensemble)
print(rmse_ensemble)
```

```
0.9652056875334333
```

Neural Network

An artificial neural network is an interconnected group of nodes, akin to the vast network of neurons in a brain. Here, each circular node represents an artificial neuron and an arrow represents a connection from the output of one artificial neuron to the input of another.

I have used keras module instead of tensorflow

In [61]:
```
from keras.models import Model
from keras.layers import Dense, Activation, Dropout
from keras.layers import Input
```

```
Using TensorFlow backend.
```

In [62]:
```
from keras.optimizers import Adam, RMSprop
from keras.layers import BatchNormalization
```

In [63]:
```
from keras import backend as K
def root_mean_squared_logarithmic_error(y_true, y_pred):
    #y_pred = K.round(y_pred)
    first_log = K.log(K.clip(y_pred, K.epsilon(), None) + 1.)
    second_log = K.log(K.clip(y_true, K.epsilon(), None) + 1.)
    return K.sqrt(K.mean(K.square(first_log - second_log)))

rmsle = root_mean_squared_logarithmic_error
learning_rate = 0.001
```

In [64]:
```
num_features = train_x.shape[-1]
_input = Input(shape=(num_features,))
#_norm = BatchNormalization()(_input)
layer1 = Dense(150, activation='linear')(_input)
layer1 = Dropout(0.2)(layer1)
layer2 = Dense(20, activation='relu')(layer1)
_output = Dense(1, activation='relu')(layer2)

model = Model(inputs=[_input], outputs=[_output])
optimizer = Adam(learning_rate)
model.compile(optimizer=optimizer,
              loss=rmsle)
```

In [65]:
```
from keras.callbacks import ModelCheckpoint
from keras.callbacks import ReduceLROnPlateau, LearningRateScheduler

checkpoint = ModelCheckpoint('weights1.h5', save_best_only=True,
                             monitor='val_loss', verbose=0)
reduce_lr = ReduceLROnPlateau(monitor='loss', factor=0.2,
              patience=10, verbose=1)

def scheduler(epoch):
    if epoch < 40:
        return learning_rate
    if epoch > 40 and epoch < 80:
        return learning_rate * 0.1
    else:
        return learning_rate * 0.05

lr_scheduler = LearningRateScheduler(scheduler)

results = model.fit(train_x, train_y,
                    batch_size=256,
                    epochs=50, callbacks=[checkpoint],
                    validation_split=0.2)
```

```
Train on 933532 samples, validate on 233383 samples
```

In [66]:
```
y_test_hat = model.predict(val_x)
print('Test MSE: {}'.format(mean_squared_error(val_y, y_test_hat)))
```

```
Test MSE: 42.43811166912354
```

In [67]:
```
test_loss = []
for yt, yth in zip(val_y, y_test_hat):
    diff = np.log(yt + 1.) - np.log(yth + 1.)
    test_loss.append(diff ** 2)
print(np.sqrt(np.mean(test_loss)))
```

```
2.00744
```

Conclusion

Evaluation of models XGBoost: 0.1077 RandomForest: 0.1103 Ridge: 0.5107 Lasso: 0.5297 EnsembleLearning: 0.9652 NeraulNetwork: 2.0074

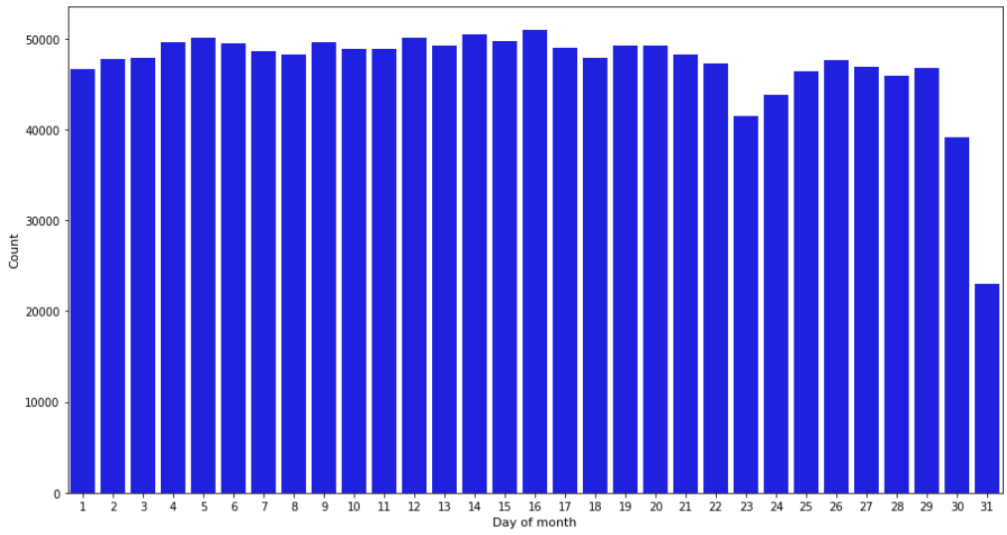
Among all the models XGBoost performed exceptionally well when compared to other algorithms It is not a rule that Ensemble Learning would give better results, sometimes it may perform worse than independent models

After adding holiday, weather, fast route features to the original data, the rmse value of all the algorithms improved which means it gave good accuracy.

Further Improvements 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

Results

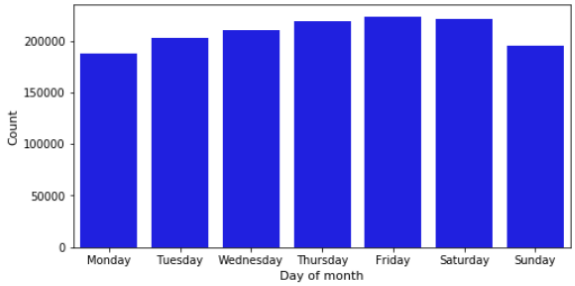
In [53]:
```
#Checking at what date of a month the number of trips are more
f = plt.figure(figsize=(15,8))
sns.countplot(x='pickup_daymonth', data=df_train, color = 'blue')
plt.xlabel('Day of month', fontsize=11)
plt.ylabel('Count', fontsize=11)
plt.show()
```



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()
```



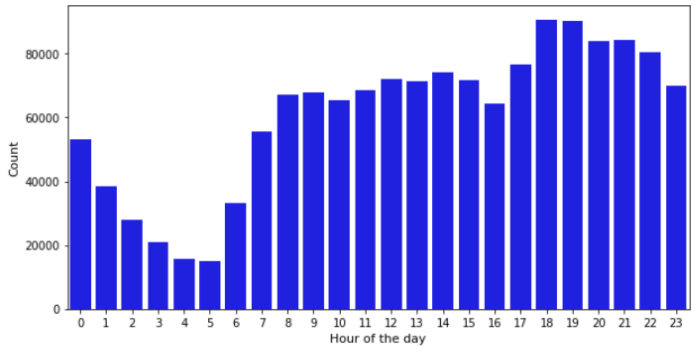
In the above figure I have plotted number of taxi trips during a certain day of the week. As we can see from above figure, the number of trips usually spikes, especially during the weekend i.e. Friday and Saturday, while it stays low on Monday.

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.

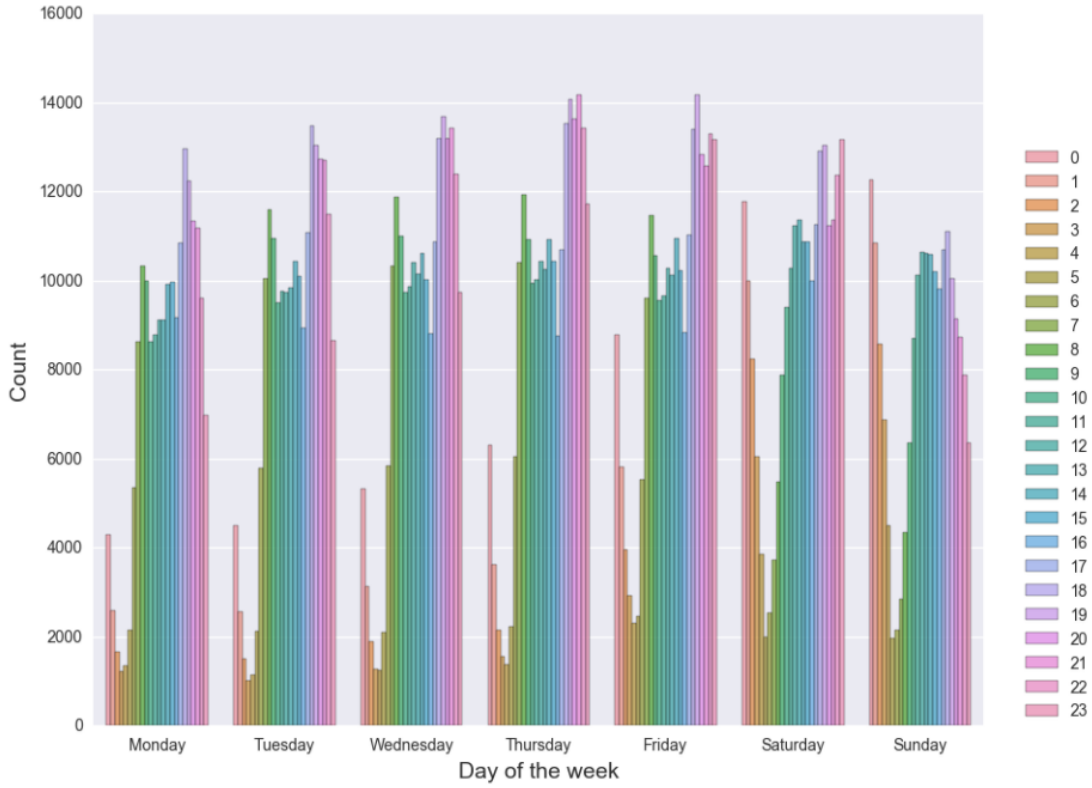
In [56]:
```
f = plt.figure(figsize=(10,5))
sns.countplot(x='pickup_hour', data=df_train, color = 'blue')
plt.xlabel('Hour of the day', fontsize=11)
plt.ylabel('Count', fontsize=11)
plt.show()
```



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()
```

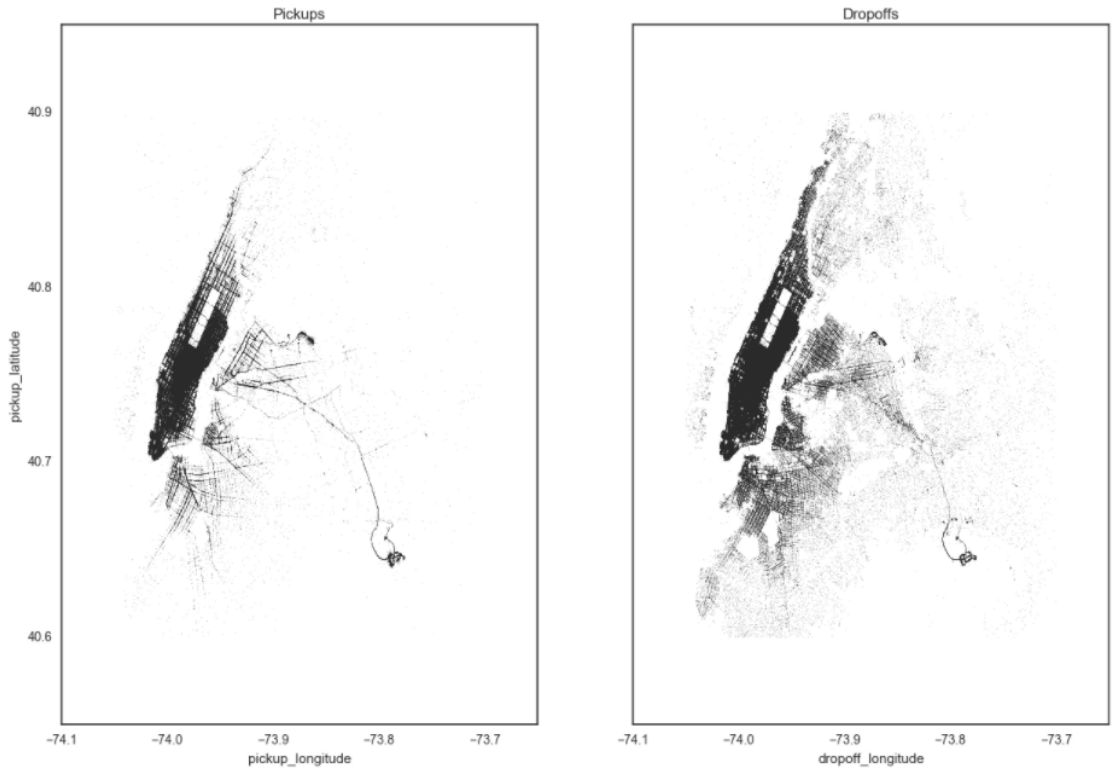


In the above figure I have tried to combine all the results I got above into one single plot. On the Y-axis we have the count of trips and on the X-axis, we have day of the week combined with the hour of the day. As we can observe, count of trips is more on weekend with the count spikes in peak hours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

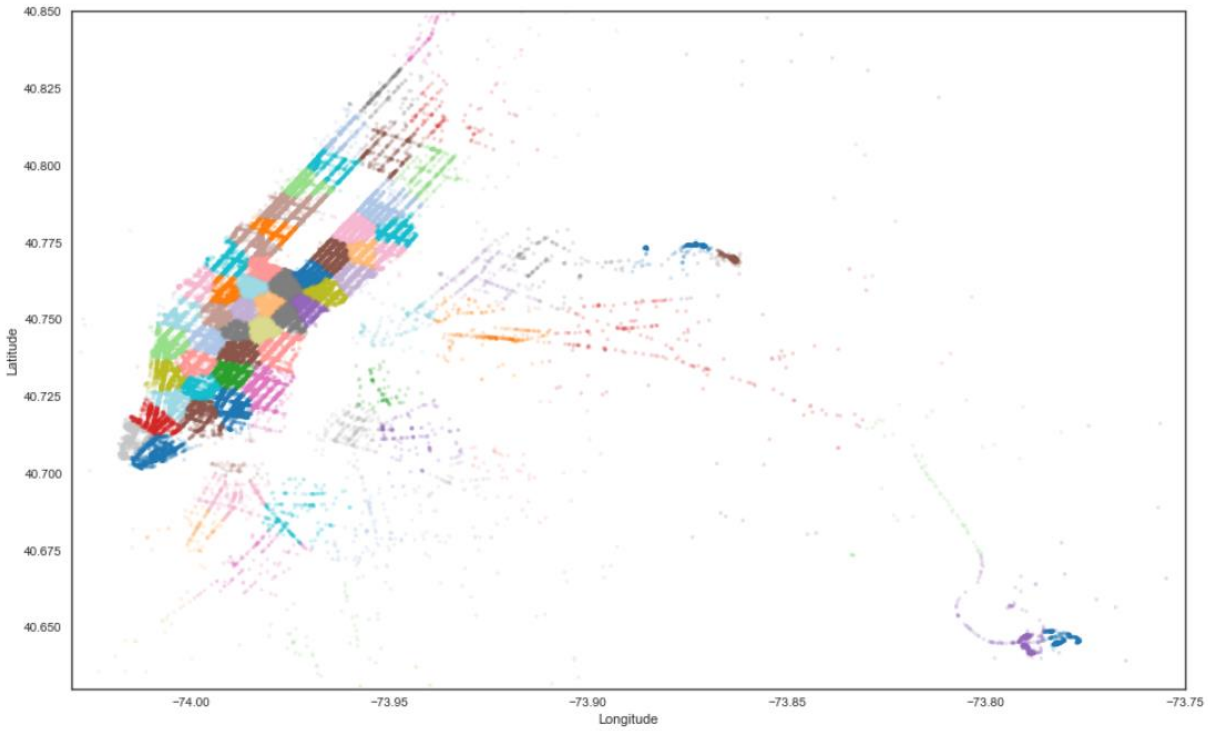
Out[44]: <matplotlib.text.Text at 0x274bc3e8668>



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.