Contents

Introduction

This kernel will be an exploratory data analysis on the BigQuery-Geotab Intersection Congestion competition data.

Geotab provides a wide variety of aggregate datasets gathered from commercial vehicle telematics devices. Harnessing the insights from this data has the power to improve safety, optimize operations, and identify opportunities for infrastructure challenges.

We have a regression problem for which we must predict 6 target values: three statistics: the 20th, 50th, and 80th percentiles, for each of two metrics: the total time a vehicle stopped at an intersection, and the distance between the intersection and the first place a vehicle stopped while waiting,for each observation in the test set. The given data consists of aggregated trip logging metrics from commercial vehicles, such as semi-trucks. The data have been grouped by intersection, month, hour of day, direction driven through the intersection, and whether the day was on a weekend or not.

Image Source: *https://www.geotab.com/blog/traffic-congestion/*

Importing libraries..

In [1]:

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python docker image: https://github.com/kaggle/docker-python


import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.style as style
style.use('seaborn-bright')
import matplotlib.pyplot as plt
import seaborn as sns
import tqdm

import os
```

```
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

import matplotlib.gridspec as gridspec
```

```
/kaggle/input/bigquery-geotab-intersection-congestion/test.csv
/kaggle/input/bigquery-geotab-intersection-congestion/BigQuery-Dataset-Access.md
/kaggle/input/bigquery-geotab-intersection-congestion/train.csv
/kaggle/input/bigquery-geotab-intersection-congestion/sample_submission.csv
/kaggle/input/bigquery-geotab-intersection-congestion/submission_metric_map.json
```

DataLoading&Overview

Loading data..

In [2]:

```
train = pd.read_csv('../input/bigquery-geotab-intersection-congestion/train.csv')
test = pd.read_csv('../input/bigquery-geotab-intersection-congestion/test.csv')
mergedData = train.merge(test, 'outer')
```

In [3]:

```
train.head()
```

Out[3]:

	RowId	IntersectionId	Latitude	Longitude	EntryStreetName	ExitStreetName	EntryHeading	ExitHeading	Hour	Week
0	1920335	0	33.79166	-84.43003	Marietta Boulevard Northwest	Marietta Boulevard Northwest	NW	NW	0	0
1	1920336	0	33.79166	-84.43003	Marietta Boulevard Northwest	Marietta Boulevard Northwest	SE	SE	0	0
2	1920337	0	33.79166	-84.43003	Marietta Boulevard Northwest	Marietta Boulevard Northwest	NW	NW	1	0
3	1920338	0	33.79166	-84.43003	Marietta Boulevard Northwest	Marietta Boulevard Northwest	SE	SE	1	0
4	1920339	0	33.79166	-84.43003	Marietta Boulevard Northwest	Marietta Boulevard Northwest	NW	NW	2	0

5 rows × 28 columns

In [4]:

```
test.head()
```

Out[4]:

	RowId	IntersectionId	Latitude	Longitude	EntryStreetName	ExitStreetName	EntryHeading	ExitHeading	Hour	Weeker
0	0	1	33.75094	-84.39303	Peachtree Street Southwest	Mitchell Street Southwest	SW	SE	0	0
1	1	1	33.75094	-84.39303	Peachtree Street Southwest	Peachtree Street Southwest	SW	SW	0	0

	RowId	IntersectionId	Latitude	Longitude	EntryStreetName	ExitStreetName	EntryHeading	ExitHeading	Hour	Weeker
2	2	1	33.75094	-84.39303	Peachtree Street Southwest	Peachtree Street Southwest	NE	NE	1	0
3	3	1	33.75094	-84.39303	Peachtree Street Southwest	Peachtree Street Southwest	SW	SW	1	0
4	4	1	33.75094	-84.39303	Peachtree Street Southwest	Peachtree Street Southwest	NE	NE	2	0

In [5]:

```
mergedData.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2777744 entries, 0 to 2777743
Data columns (total 28 columns):
RowId                       int64
IntersectionId              int64
Latitude                    float64
Longitude                   float64
EntryStreetName             object
ExitStreetName              object
EntryHeading                object
ExitHeading                 object
Hour                        int64
Weekend                     int64
Month                       int64
Path                        object
TotalTimeStopped_p20        float64
TotalTimeStopped_p40        float64
TotalTimeStopped_p50        float64
TotalTimeStopped_p60        float64
TotalTimeStopped_p80        float64
TimeFromFirstStop_p20       float64
TimeFromFirstStop_p40       float64
TimeFromFirstStop_p50       float64
TimeFromFirstStop_p60       float64
TimeFromFirstStop_p80       float64
DistanceToFirstStop_p20     float64
DistanceToFirstStop_p40     float64
DistanceToFirstStop_p50     float64
DistanceToFirstStop_p60     float64
DistanceToFirstStop_p80     float64
City                        object
dtypes: float64(17), int64(5), object(6)
memory usage: 614.6+ MB
```

Data features:

- RowId: Unique identifier for each row/observation, each of which is an aggregate of some congestion data.
- IntersectionId: Identifier for specific intersection (2839 unique in train and test sets combined)
- Latitude
- Longitude
- EntryStreetName
- ExitStreetName
- EntryHeading: Direction cars entering intersection are moving in (N, W, E, S, NW, SW, SE, NE)
- ExitHeading: Direction cars exiting intersection are moving in (N, W, E, S, NW, SW, SE, NE)
- Hour: Hour of day (0-23)
- Weekend: No: 0, Yes: 1
- Month: Has unique values (1,5,6,7,8,9,10,11,12) in both train and test sets, which implies that the congestion data ranges from May of one year to January of the next year.
- Path: Concatenation of strings from the columns: EntryStreetName, EntryHeading, ExitStreetName, ExitHeading
- City: Atlanta, Boston, Chicago, or Philadelphia

In [6]:

```
print(f'Train dataset has {train.shape[0]} rows and {train.shape[1]} columns.')
```

```
print(f'Test dataset has {test.shape[0]} rows and {test.shape[1]} columns.')
```

```
Train dataset has 857409 rows and 28 columns.
Test dataset has 1920335 rows and 13 columns.
```

We have two moderately sized datasets, with the number of observations in the test set being notably higher than that of the train set. The following features are the difference between the two sets:

In [7]:

```
[col for col in train.columns if col not in test.columns]
```

Out[7]:

```
['TotalTimeStopped_p20',
 'TotalTimeStopped_p40',
 'TotalTimeStopped_p50',
 'TotalTimeStopped_p60',
 'TotalTimeStopped_p80',
 'TimeFromFirstStop_p20',
 'TimeFromFirstStop_p40',
 'TimeFromFirstStop_p50',
 'TimeFromFirstStop_p60',
 'TimeFromFirstStop_p80',
 'DistanceToFirstStop_p20',
 'DistanceToFirstStop_p40',
 'DistanceToFirstStop_p50',
 'DistanceToFirstStop_p60',
 'DistanceToFirstStop_p80']
```

The test set contains the same columns from the training set except for the above features. The task is to predict the 20th, 50th, and 80th percentiles of the variables TotalTimeStopped and DistanceToFirstStop. It seems that additional percentile statistics, as well as a TimeFromFirstStop metric have been provided to assist with model building.

Examine the columns that have missing data:

In [8]:

```
print('Print names of columns in train set with null values, along with number of null values:')
trainNull = [col for col in  train.columns if train[col].isnull().any()]
trainNullD = {}
for col in trainNull:
    trainNullD.update({col: ['Number missing: ' + str(train[col].isnull().sum()),
                             'Proportion missing: ' + str(round(train[col].isnull().sum() / train.s
ape[0],3))]})
for a,b in trainNullD.items():
    print(a)
    print(b)
```

```
Print names of columns in train set with null values, along with number of null values:
EntryStreetName
['Number missing: 8189', 'Proportion missing: 0.01']
ExitStreetName
['Number missing: 5534', 'Proportion missing: 0.006']
```

In [9]:

```
print('Print names of columns in test set with null values, along with number of null values:')
testNull = [col for col in  test.columns if test[col].isnull().any()]
testNullD = {}
for col in testNull:
    testNullD.update({col: ['Number missing: ' + str(test[col].isnull().sum()),
                             'Proportion missing: ' + str(round(test[col].isnull().sum() / test.sha
e[0],3))]})
for a,b in testNullD.items():
    print(a)
    print(b)
```

```
Print names of columns in test set with null values, along with number of null values:
EntryStreetName
```

```
EntryStreetName
['Number missing: 19157', 'Proportion missing: 0.01']
ExitStreetName
['Number missing: 16340', 'Proportion missing: 0.009']
```

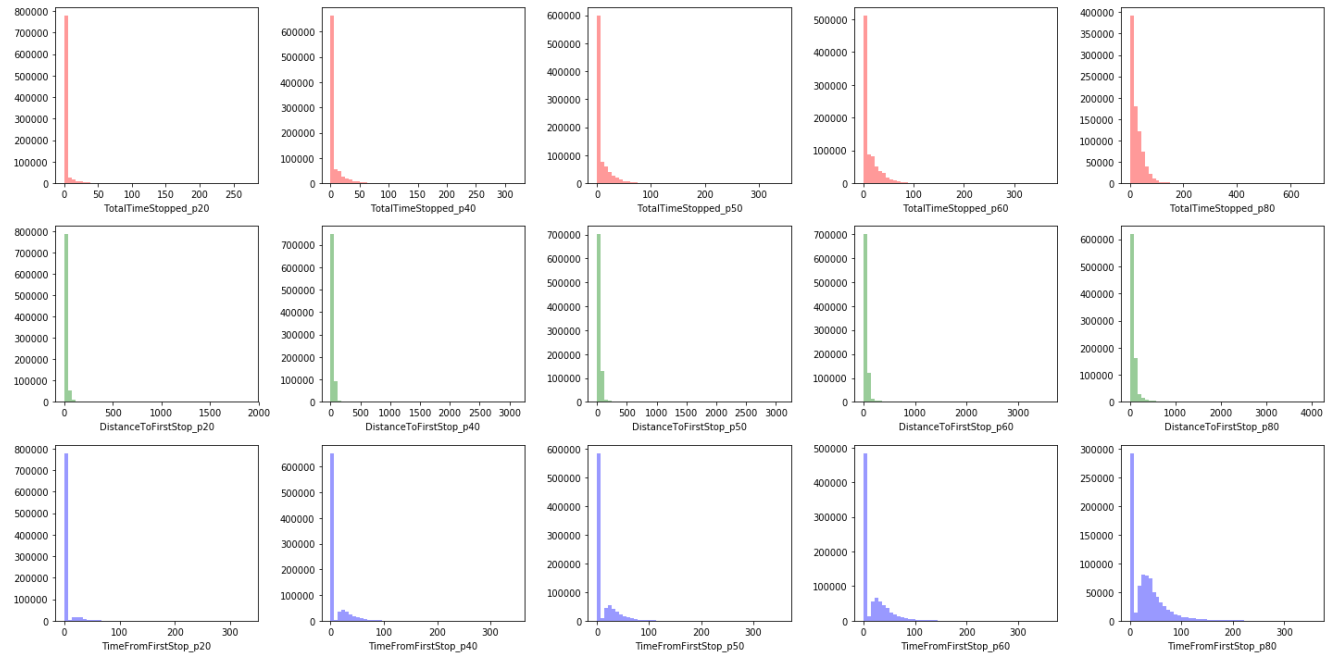
The columns with missing data are the same in both train and test sets: EntryStreetName and ExitStreetName. It seems that most columns are without any missing data. For the two that do have missing data, one strategy would be to fill them with their respective modes, or some other constant using the fillna method. But since these columns only have a very small proportion missing, simply removing those observations should also be a safe strategy.

DataExploration:TargetDistribution

First, let's see how each of the given statistics are distributed.

In [10]:

```
#Create lists of columns for each group of statistics
TotalTimeStoppedCols = list(train.loc[:,'TotalTimeStopped_p20':'TotalTimeStopped_p80'].columns)
DistanceToFirstStop =
list(train.loc[:,'DistanceToFirstStop_p20':'DistanceToFirstStop_p80'].columns)
TimeFromFirstStopCols = list(train.loc[:,'TimeFromFirstStop_p20':'TimeFromFirstStop_p80'].columns)


f1, axes = plt.subplots(3,5, figsize=(20, 10), sharex=False)

for i, col in enumerate(TotalTimeStoppedCols):
    sns.distplot(train[col], kde=False, ax=axes[0,i], label = col, color = 'r')

for i, col in enumerate(DistanceToFirstStop):
    sns.distplot(train[col], kde=False, ax=axes[1,i], label = col,  color = 'g')

for i, col in enumerate(TimeFromFirstStopCols):
    sns.distplot(train[col], kde=False, ax=axes[2,i], label = col, color = 'b')


plt.tight_layout()
```



For each of the three statistics, the number of samples contained in each percentile visibly increases as the histograms are read from left to right, as expected. Let's zoom in on the highest available percentile for each, which encompasses all available data.

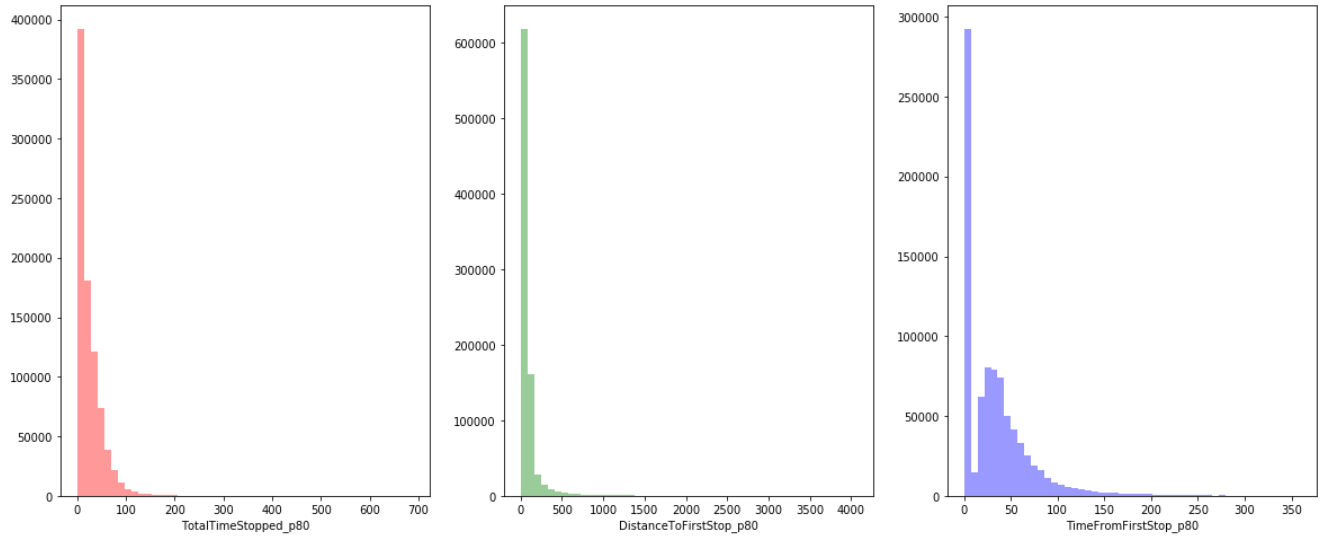
In [11]:

```
#lastPercentiles = ['TotalTimeStopped_p80', 'DistanceToFirstStop_p80', 'TimeFromFirstStop_p80']
f2, axes2 = plt.subplots(1,3, figsize=(20, 8), sharex=False)

sns.distplot(train['TotalTimeStopped_p80'], kde=False, ax=axes2[0], color = 'r')
```

```
sns.distplot(train['TotalTimeStopped_p80'], kde=False, ax=axes2[0], color = 'r' )
sns.distplot(train['DistanceToFirstStop_p80'], kde=False, ax=axes2[1], color = 'g' )
sns.distplot(train['TimeFromFirstStop_p80'], kde=False, ax=axes2[2], color = 'b' )
```

Out[11]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7d8b4b35f8>
```



The data is skewed towards zero for all three statistics. This implies that cars are most likely to pass through the intersection without stopping, out of all possible stopping times, and higher stopping times are increasingly unlikely, which makes sense. TotalTimeStopped by itself should be enough to provide a good idea of how heavy the congestion is. For each congestion data aggregate (each row), a low value of TotalTimeStopped_p80 should thus be representative of normal or good traffic, while a high value would represent slow traffic. As a simple test, the average 80th percentile of total stop time is 24.83 seconds on weekdays, and 18.05 seconds on weekends. This makes sense as most people are more likely to work on weekdays only and rest on weekends, leading to busier streets on weekdays due to commuting to/from work, etc.

In [12]:

```
mergedData.groupby('Weekend')['TotalTimeStopped_p80'].mean()
```

Out[12]:

```
Weekend
0    24.828532
1    18.051728
Name: TotalTimeStopped_p80, dtype: float64
```

Direction

From prior driving experience, one can recall that it would take on average more time to make a left/right turn than to drive straight through an intersection. This makes sense since oncoming cars in the opposite lane have the right of way, and one would have to wait for them to pass before turning, leading to blocking/slowing down the cars behind you as well. Let's see if the data agrees with us by creating an 'isSameDirection' variable, which is true if the entry direction and exit direction are the same and false otherwise:

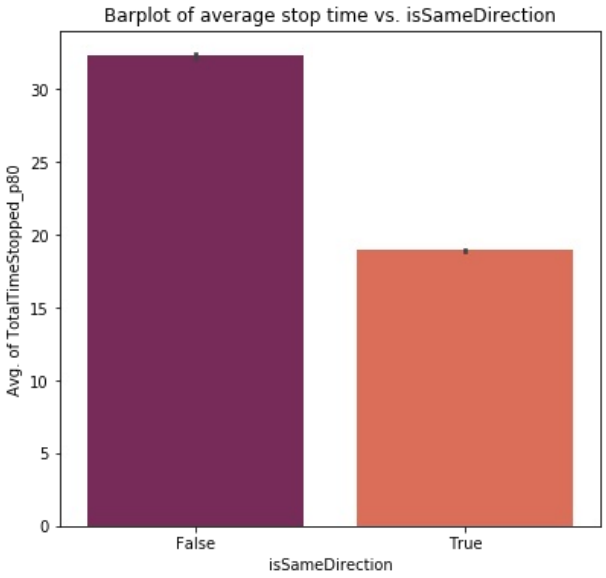
In [13]:

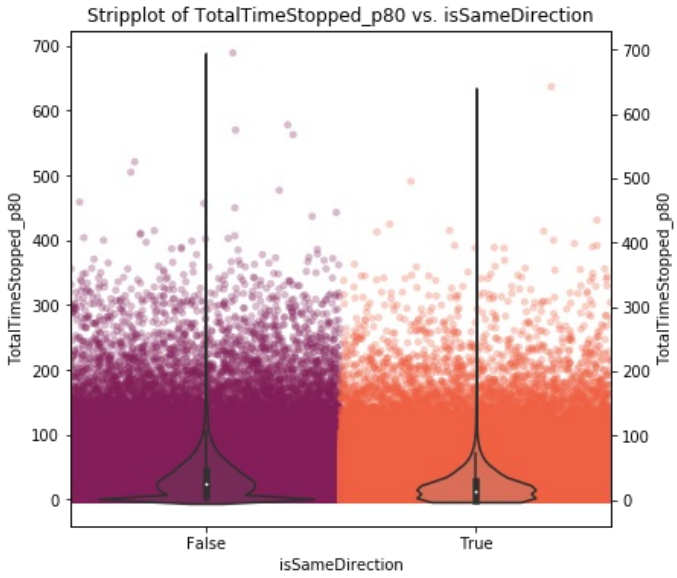
```
#%%time
train['isSameDirection'] = train['EntryHeading'] == train['ExitHeading']

f3, axes3 = plt.subplots(1,2, figsize=(14, 6))

bar = sns.barplot(x='isSameDirection', y='TotalTimeStopped_p80', data = train, palette = 'rocket',
                  ax = axes3[0] )
axes3[0].set_title('Barplot of average stop time vs. isSameDirection')
axes3[0].set_xlabel('isSameDirection')
axes3[0].set_ylabel('Avg. of TotalTimeStopped_p80')

strip = sns.stripplot(x='isSameDirection', y='TotalTimeStopped_p80', data = train, palette = 'rocke
t',
```

```
                      jitter = 0.5, alpha = 0.3, ax = axes3[1])
axt = axes3[1].twinx()
vio = sns.violinplot(x='isSameDirection', y='TotalTimeStopped_p80', data = train, palette = 'rocket
',
                     ax = axt)

axes3[1].set_title('Stripplot of TotalTimeStopped_p80 vs. isSameDirection')
axes3[1].set_xlabel('isSameDirection')
axes3[1].set_ylabel('TotalTimeStopped_p80')
```

Out[13]:

```
Text(0, 0.5, 'TotalTimeStopped_p80')
```





In [14]:

```
print(train.groupby('isSameDirection')['TotalTimeStopped_p80'].agg('mean'))
print(train['isSameDirection'].value_counts(normalize=True))
```

```
isSameDirection
False    32.296313
True     18.942388
Name: TotalTimeStopped_p80, dtype: float64
True     0.700037
False    0.299963
Name: isSameDirection, dtype: float64
```

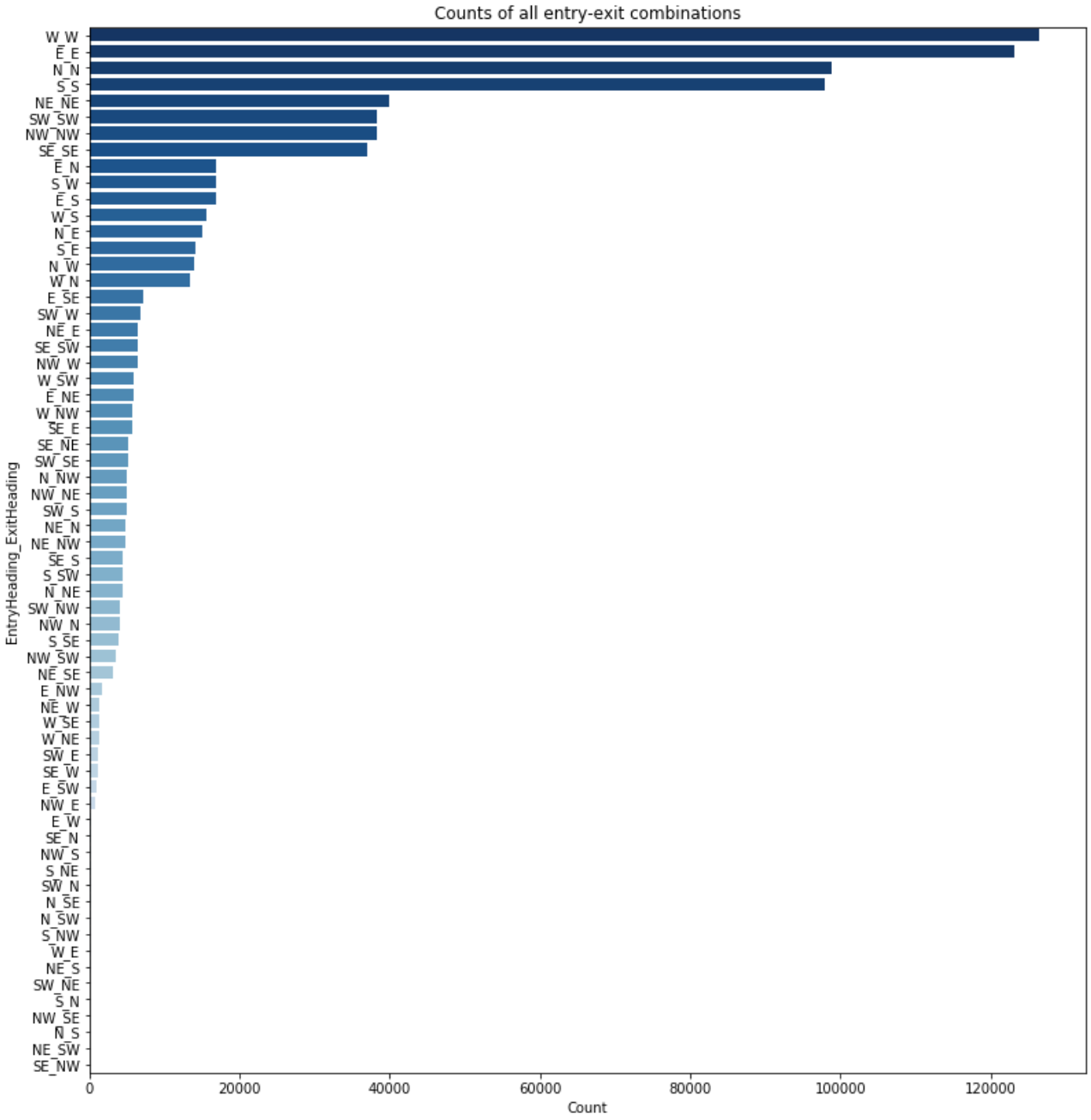
It seems that cars approaching intersections drive straight through with a probability of roughly 0.70, and turn in a different direction with a probability of 0.30 (This distribution is very nearly the same for the test set as well).
The barplot supports our earlier guess: that cars travelling through intersections without changing direction would, on average, have lower total stop time.
isSameDirection should be a feature worth keeping for model training. Taking this idea further, it might be worth investigating the same metric for all combinations of entry and exit direction from (N, W, E, S, NW, SW, SE, NE). Let's take a look at the counts of all the entry-exit combinations:

In [15]:

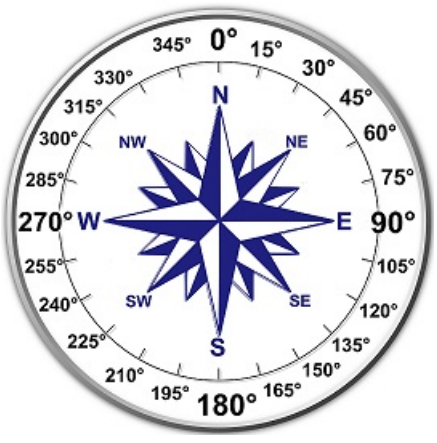
```
#column 'Path' has extraneous information (i.e. street names), we only want path direction
train['pathDirectionOnly'] = train['EntryHeading'] + '_' + train['ExitHeading']

pathd = pd.DataFrame({'EntryHeading_ExitHeading': train['pathDirectionOnly'].value_counts().index,

                      'Count': train['pathDirectionOnly'].value_counts()})

f4, axes4 = plt.subplots( figsize=(13, 14))
sns.barplot(x='Count', y='EntryHeading_ExitHeading', data = pathd, ax = axes4, palette='Blues_r')
axes4.set_title('Counts of all entry-exit combinations')
```

Out[15]:

```
Text(0.5, 1.0, 'Counts of all entry-exit combinations')
```



There are a total of 64 entry-exit combinations. The most frequent ones involve no turning (i.e. East -> East: 0 degrees), the next most frequent are simple right/left turns (i.e. East ->North: 90 degrees), followed by turns of abnormal angles (i.e. Northwest -> West: 45 degrees). It may be more informative to convert these 64 entry-exit combinations into the 5 possible angles associated with a turn (0, 45, 90, 135, 180), and check how this feature compares with stop time.



In [16]:

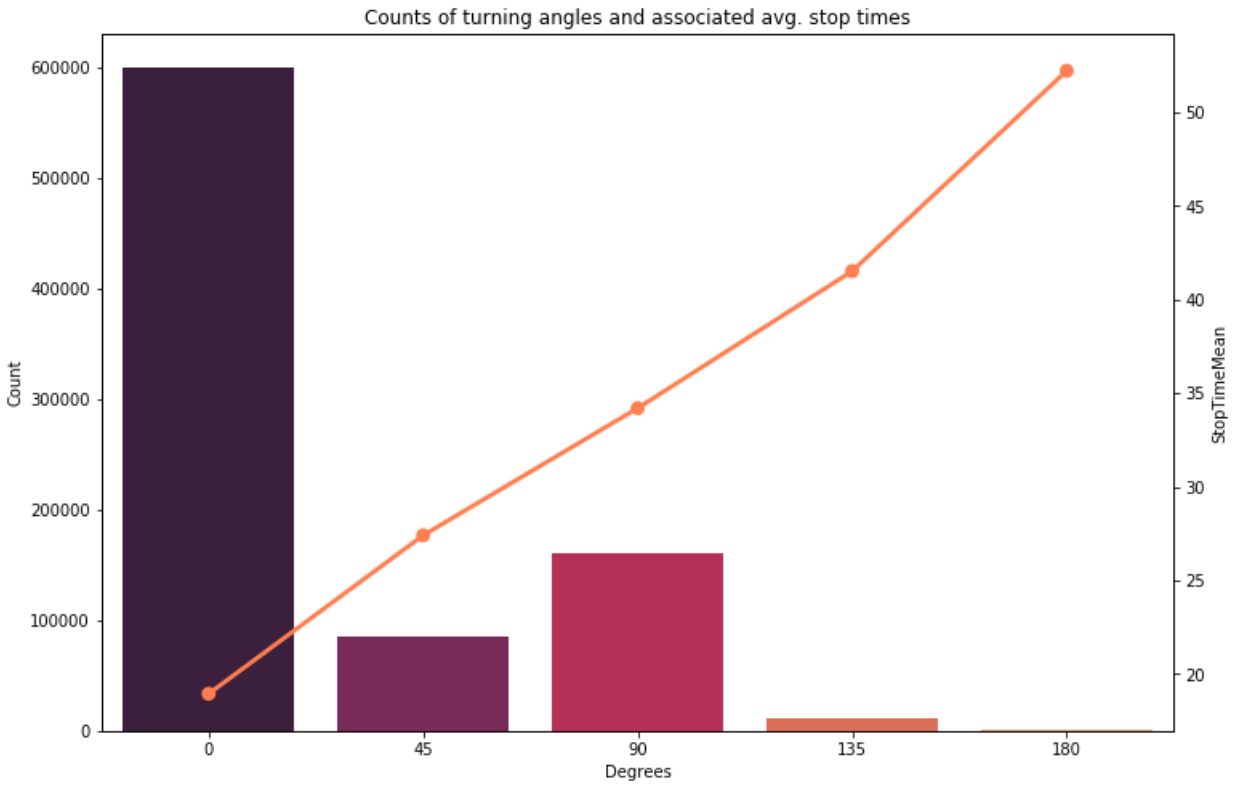
```
#Define function to get angle between 'EntryHeading' and 'ExitHeading'
def getAngleOfTurn(df):
        compassAngle = {'N':0, 'NE':45, 'E':90, 'SE':135, 'S':180,
                        'SW':225, 'W':270, 'NW':315 }
        a = df['EntryHeading']
        b = df['ExitHeading']
        angle = abs(compassAngle[a] - compassAngle[b])
        if angle < 225:
            return angle
        elif angle == 225:
            return 135
        elif angle == 270:
            return 90
        elif angle == 315:
            return 45
train['angleOfTurn'] = train.apply(getAngleOfTurn, axis=1)
```

In [17]:

```
anglesDf = pd.DataFrame({'angleOfTurn': train.groupby('angleOfTurn')
['TotalTimeStopped_p80'].agg('mean').index,
                         'Count': train.groupby('angleOfTurn')['angleOfTurn'].agg('count'),
                         'StopTimeMean' : train.groupby('angleOfTurn')['TotalTimeStopped_p80'].agg(
mean')})

f5, axes5 = plt.subplots( figsize=(12, 8))

dircount = sns.barplot(x='angleOfTurn', y='Count' , data = anglesDf, palette = 'rocket', ax = axes5
)
dircount.set_title('Counts of turning angles and associated avg. stop times')
dircount.set_ylabel('Count')
dircount.set_xlabel('Degrees')

a = axes5.twinx()
sns.pointplot(x = 'angleOfTurn', y = 'StopTimeMean', color = 'coral',data = anglesDf, ax = a)
```

Out[17]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7d886e82e8>
```



In [18]:

In [18]:

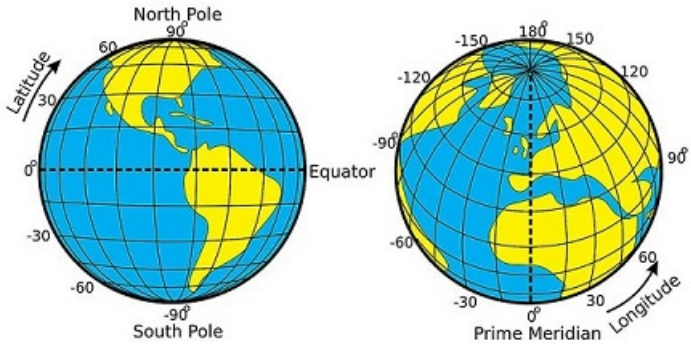
```
anglesDf.iloc[:,1:]
```

Out[18]:

	Count	StopTimeMean
angleOfTurn		
0	600218	18.942388
45	85578	27.403632
90	159939	34.192605
135	10941	41.511105
180	733	52.208731

The frequency of each turn angle is consistent with our previous investigation of 'isSameDirection' - turns of 0 degrees count for around 0.70 of all turns, and frequencies of all other turn angles (45, 90, 135, and 180) sum to roughly 0.30. Plotting the average 80th percentile of TotalTimeStopped (abbreviated as StopTimeMean) for each of the angle groups on the same axis reveals a linear relationship between the magnitude of the angle turned and the time stopped. Interestingly, there are even some drivers that made complete U-turns of 180 degrees at the intersection, going back the way they came. Unsurprisingly, these turned out to have the highest average stop time. 'angleOfTurn' should be a feature worth keeping.

Location&Time



Another space-relevant component of data given to us is latitude and longitude. Latitude provides information about the distance north or south of the equator and ranges from 0 to 90 (0 at the equator, 90 at the North or South Pole), while longitude provides information about the distance east or west of the Prime Meridian, an imaginary line drawn between the North and South Poles, passing through Greenwich, England, and ranges from 0 to 180.

The Earth's axis is tilted 23.5° to the perpendicular, meaning that the amount of sunlight that a particular latitude receives changes with the seasons. From April to September, the Northern Hemisphere is tilted toward the Sun, where it receives more energy; the Southern Hemisphere receives this additional energy between October and March, when it is tilted toward the Sun. *Source:* *https://enviroliteracy.org/air-climate-weather/climate/latitude-climate-zones/*

With this in mind, it should be worthwhile to investigate the effect of climate on traffic by plotting the relationship between latitude and average stop times, organized by seasons. For our purposes here, we'll treat values of latitude as a categorical variable and round them, since our data encompass only four cities, for each of which the values of latitude are sharply distributed.

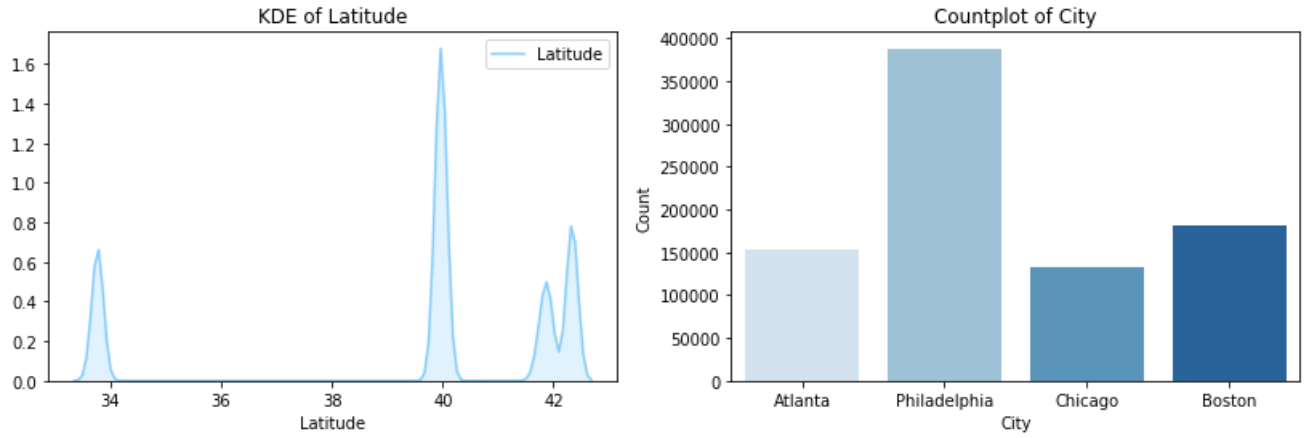
In [19]:

```
f7, axes7 = plt.subplots(1,2, figsize=(14, 4))
kde1 = sns.kdeplot(data = train['Latitude'], shade = True, color = 'lightskyblue', ax = axes7[0])
kde1.set_xlabel('Latitude')
kde1.set_title('KDE of Latitude')

kde2 = sns.countplot(x = 'City', data = train, ax = axes7[1],  palette = 'Blues',
```

```
                    order = ['Atlanta', 'Philadelphia', 'Chicago', 'Boston'] )
kde2.set_title('Countplot of City')
kde2.set_ylabel('Count')
```

Out[19]:

```
Text(0, 0.5, 'Count')
```



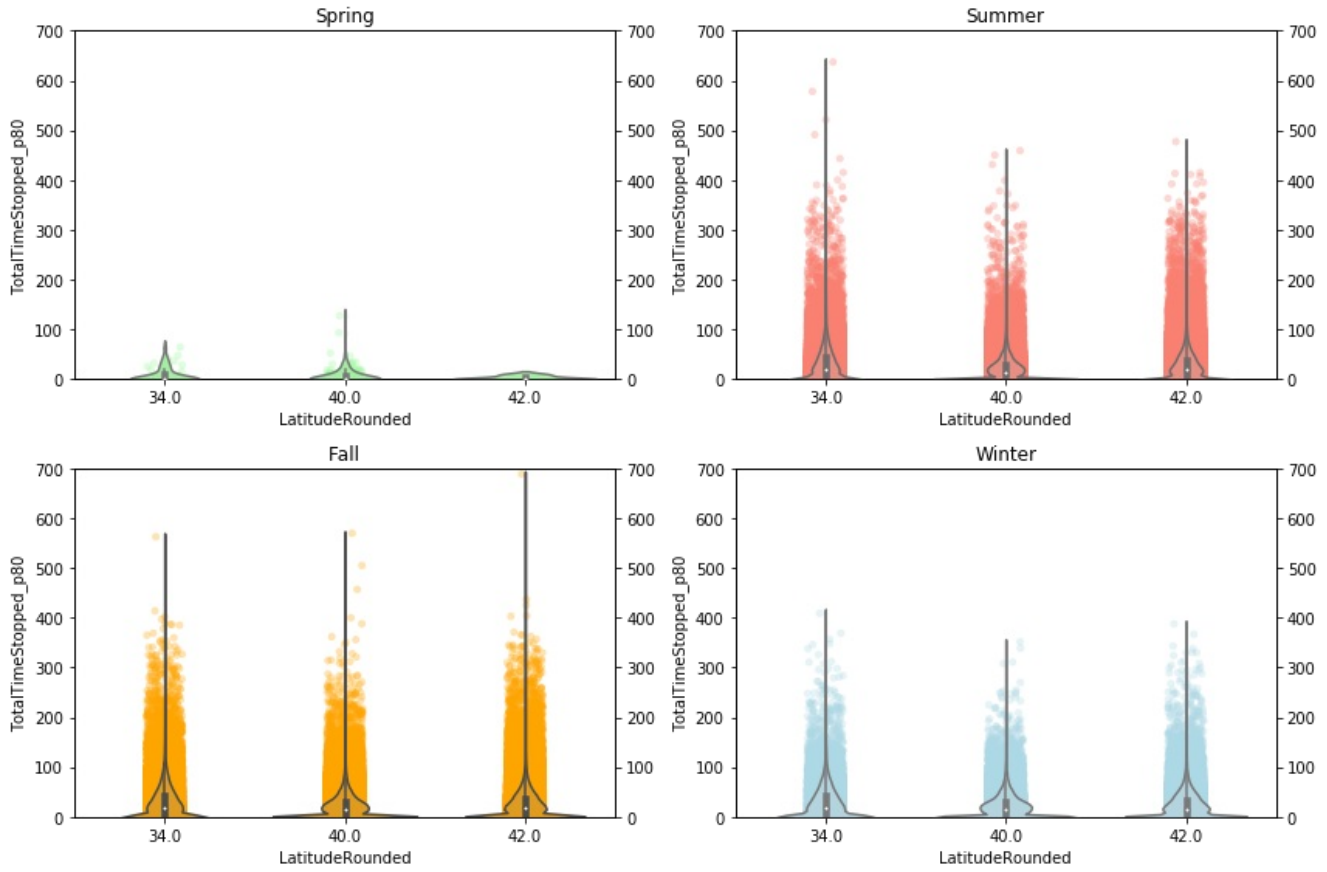
- Atlanta: Latitude ~34° N
- Philadelphia: Latitude ~40° N
- Chicago & Boston: Latitude ~ 42° N

Since Boston and Chicago have nearly the same latitude, and climate differences are unlikely to be noticeable over such a small range of latitude difference, we will categorize these two cities together.

In [20]:

```
#%%time
train['LatitudeRounded'] = round(train['Latitude'])

trainSpring = train.loc[train['Month'].isin([3,4,5])]
trainSummer = train.loc[train['Month'].isin([6,7,8])]
trainFall = train.loc[train['Month'].isin([9,10,11])]
trainWinter = train.loc[train['Month'].isin([12,1,2])]

f6, axes6 = plt.subplots(2, 2, figsize=(12, 8))
#########################################################################
a = sns.stripplot(x='LatitudeRounded', y='TotalTimeStopped_p80', data = trainSpring,
                  color = 'palegreen', alpha = 0.3, ax=axes6[0,0])
axt1 = axes6[0,0].twinx()
vio1 = sns.violinplot(x='LatitudeRounded', y='TotalTimeStopped_p80', data = trainSpring,
                      color = 'palegreen', alpha = 0.3, ax = axt1)
a.set_title('Spring')
a.set_ylim(0,700)
a.set_autoscaley_on(False)
axt1.set_ylabel('')
axt1.set_ylim(0,700)
axt1.set_autoscaley_on(False)
#########################################################################
b = sns.stripplot(x='LatitudeRounded', y='TotalTimeStopped_p80', data = trainSummer,
                  color = 'salmon', alpha = 0.3, ax=axes6[0,1])
axt2 = axes6[0,1].twinx()
vio2 = sns.violinplot(x='LatitudeRounded', y='TotalTimeStopped_p80', data = trainSummer,
                      color = 'salmon', alpha = 0.3, ax = axt2)
b.set_title('Summer')
b.set_ylim(0,700)
b.set_autoscaley_on(False)
axt2.set_ylabel('')
axt2.set_ylim(0,700)
axt2.set_autoscaley_on(False)
#########################################################################
c = sns.stripplot(x='LatitudeRounded', y='TotalTimeStopped_p80', data = trainFall,
                  color = 'orange', alpha = 0.3, ax=axes6[1,0])
axt3 = axes6[1,0].twinx()
vio3 = sns.violinplot(x='LatitudeRounded', y='TotalTimeStopped_p80', data = trainFall,
                      color = 'orange', alpha = 0.3, ax = axt3)
```

```
c.set_title('Fall')
c.set_ylim(0,700)
c.set_autoscaley_on(False)
axt3.set_ylabel('')
axt3.set_ylim(0,700)
axt3.set_autoscaley_on(False)
###############################################################################
d = sns.stripplot(x='LatitudeRounded', y='TotalTimeStopped_p80', data = trainWinter,
                  color = 'lightblue', alpha = 0.3, ax=axes6[1,1])
axt4 = axes6[1,1].twinx()
vio4 = sns.violinplot(x='LatitudeRounded', y='TotalTimeStopped_p80', data = trainWinter,
                      color = 'lightblue', alpha = 0.3, ax = axt4)
d.set_title('Winter')
d.set_ylim(0,700)
d.set_autoscaley_on(False)
axt4.set_ylabel('')
axt4.set_ylim(0,700)
axt4.set_autoscaley_on(False)

plt.tight_layout()
```



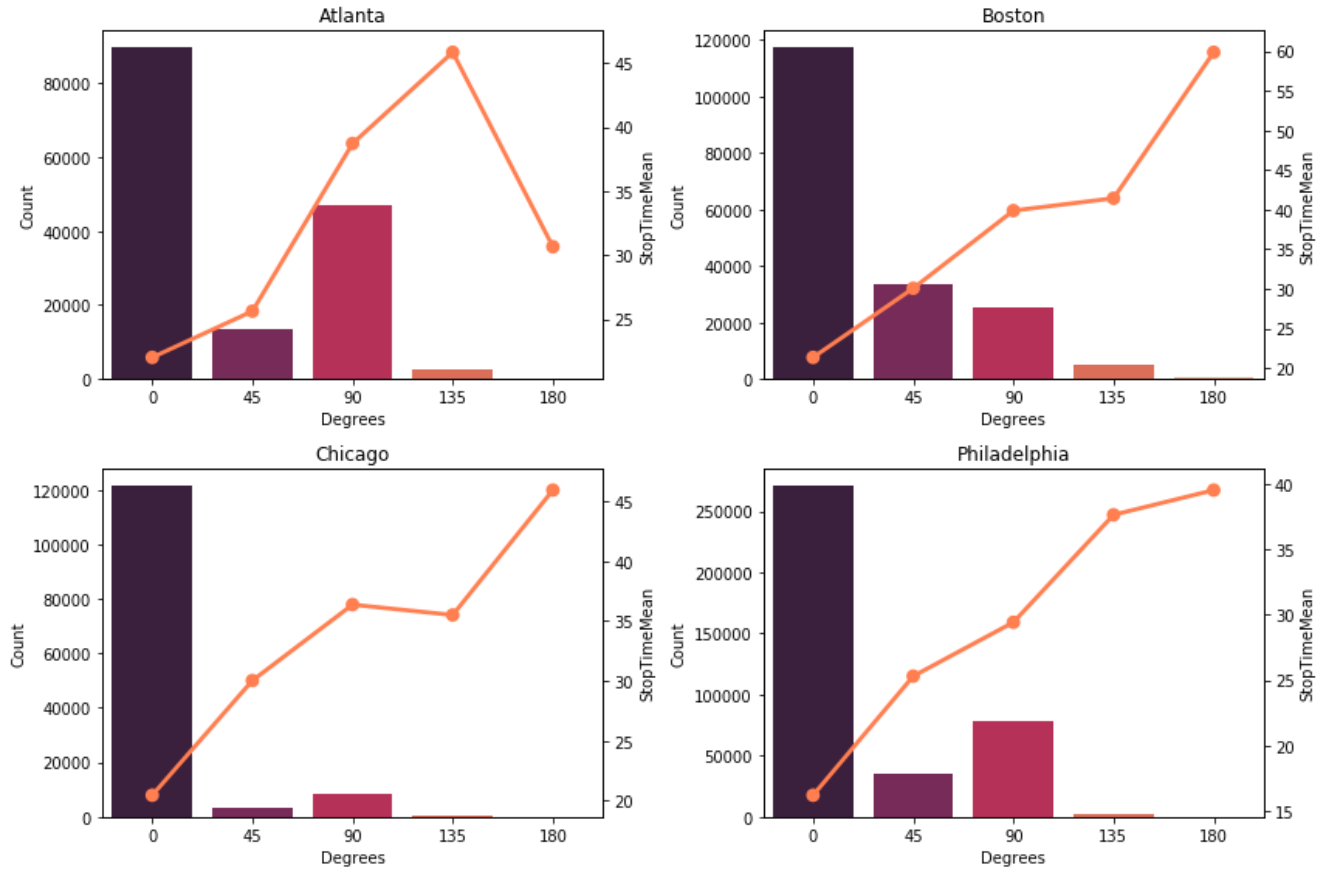
Unfortunately for our "Spring" dataset, there contains data only from the month of May, with data from March and April missing, and our "Winter" dataset is also missing a month, February (Since data was taken for all months except February, March, and April). Observing each of the remaining seasons, Summer and Fall, it seems that there are differences in stop time among the different locations. However, this cannot simply be attributed to climate differences since there may have been differences in infrastructure etc. among the cities that were the cause instead. In Summer, Latitude 34 corresponds to the highest stop times, while in Fall, Latitude 42 corresponds to the highest stop times. However, this too may be due to indirect effects of season on people's behaviors and is not necessarily evidence of a direct effect of climate on traffic.
It seems that we are unable to distinguish between effects of infrastructure and climate. Let's instead try to investigate the differences in infrastructure among the four cities.

We will revisit our earlier plot of 'Counts of turning angles and associated avg. stop times', but now we will create four different subplots, grouping data by each of the cities.

In [21]:

```
trainAtlanta = train.loc[train['City'] == 'Atlanta']
trainBoston = train.loc[train['City'] == 'Boston']
trainChicago = train.loc[train['City'] == 'Chicago']
```

```
trainPhiladelphia = train.loc[train['City'] == 'Philadelphia']
t = [trainAtlanta, trainBoston, trainChicago, trainPhiladelphia]
n = ['Atlanta', 'Boston', 'Chicago', 'Philadelphia']
I = 0
f8, axes8 = plt.subplots(2,2, figsize=(12, 8))
splots = [axes8[0,0], axes8[0,1], axes8[1,0], axes8[1,1]]

for item in splots:
    anglesDf = pd.DataFrame({'angleOfTurn': t[I].groupby('angleOfTurn')['TotalTimeStopped_p80'].agg
('mean').index,
                         'Count': t[I].groupby('angleOfTurn')['angleOfTurn'].agg('count'),
                         'StopTimeMean' : t[I].groupby('angleOfTurn')['TotalTimeStopped_p80'].agg('
ean')})

    dircount = sns.barplot(x='angleOfTurn', y='Count' , data = anglesDf, palette = 'rocket', ax = i
tem)
    dircount.set_title(n[I])
    dircount.set_ylabel('Count')
    dircount.set_xlabel('Degrees')

    a = item.twinx()
    sns.pointplot(x = 'angleOfTurn', y = 'StopTimeMean', color = 'coral',data = anglesDf, ax = a)

    I = I+1
plt.tight_layout()
```



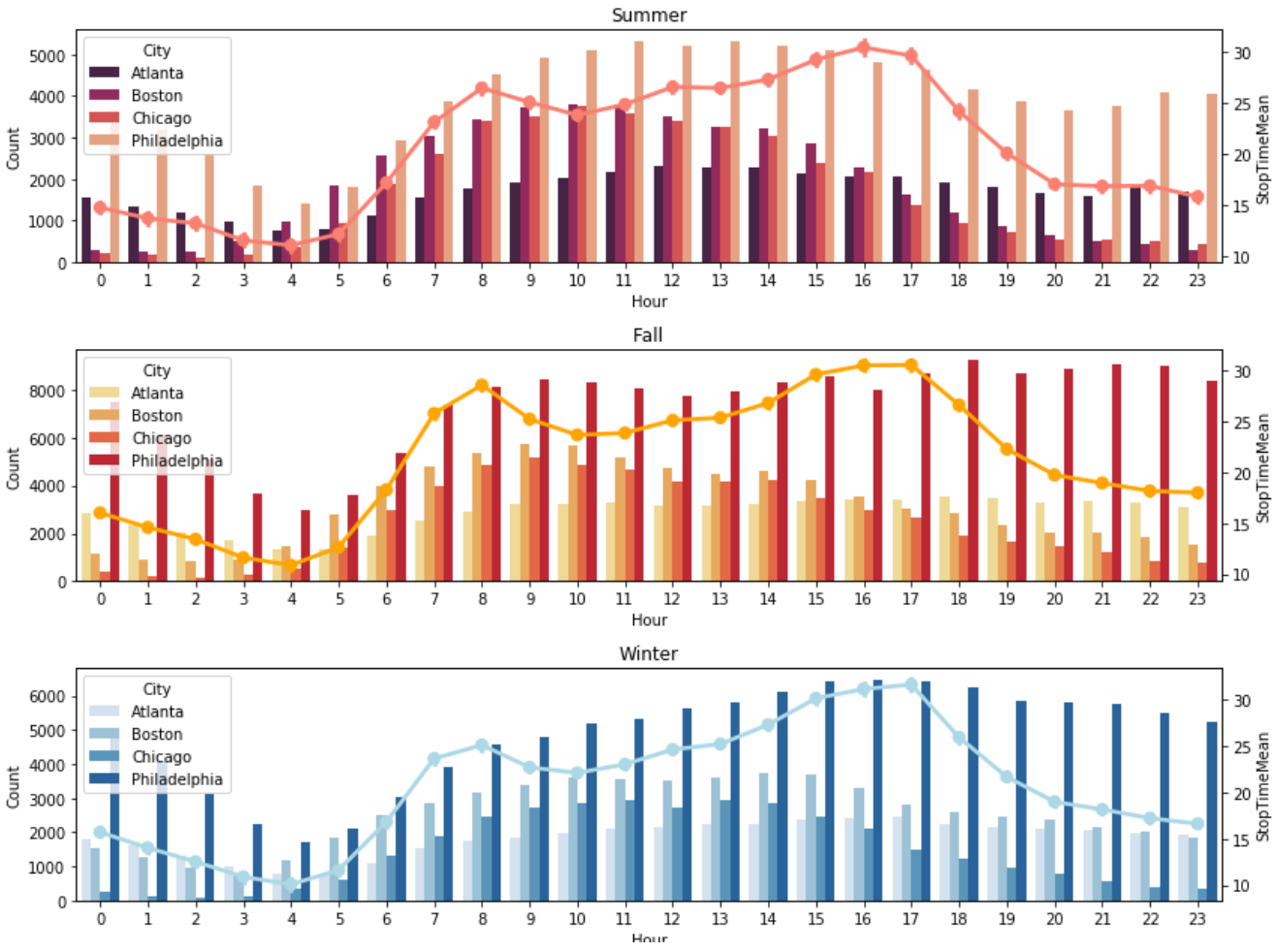
Although quite abstract, information on the frequency of turn angles should still provide some insight on differences in infrastructure among the four cities. For all cities except Boston, the trend remains that turn frequencies have the following decreasing order: 0, 90, 45, 135, and 180 degrees. But there are very clear differences in the proportions of each when comparisons are made among the different cities. The exact reasons for these differences may be un-extractable from this abstract view of the data. It may be due to genuine structural differences among the four cities, or it may simply be a matter of where/when Geotab decided to collect their data, or even a combination of the two.
Also, it seems to be generally true that increasing turn angle means increasing stop time, even when we group the data by city. The few exceptions that can be seen above are likely attributed to these large-angle turns being made at non-busy intersections and/or times, removing the need for the driver to stop and wait due to blockage from other cars.

Now let us explore the times of day and see which times are busiest. At the same time, we'll also take a look at the relationship between season and traffic from a different perspective. Since data for the months of Spring are missing except for its last, May, we will consider May to be part of the season that immediately follows: Summer. And for the purpose of maintaining (approximately)

equal amounts of data in each season set, we will shift the last month of Summer "down" into the next, and do the same for Fall. After this process, our new seasons contain the following months... Summer: 5,6,7; Fall: 8,9,10; Winter: 11,12,1.

In [22]:

```
trainSummer2 = train.loc[train['Month'].isin([5,6,7])]
trainFall2 = train.loc[train['Month'].isin([8,9,10])]
trainWinter2 = train.loc[train['Month'].isin([11,12,1])]


f9, axes9 = plt.subplots(3,1, figsize=(12, 9))
c = sns.countplot(x = 'Hour', data = trainSummer2, hue = 'City', palette = 'rocket', ax = axes9[0])
c.set_title('Summer')
c.set_ylabel('Count')

axt = axes9[0].twinx()
p = sns.pointplot(x='Hour', y='TotalTimeStopped_p80', data = trainSummer2, color = 'salmon',ax = ax
t )
p.set_ylabel('StopTimeMean')
#####
c = sns.countplot(x = 'Hour', data = trainFall2, hue = 'City', palette = 'YlOrRd', ax = axes9[1])
c.set_title('Fall')
c.set_ylabel('Count')

axt = axes9[1].twinx()
p = sns.pointplot(x='Hour', y='TotalTimeStopped_p80', data = trainFall2, color = 'orange', ax = axt
)
p.set_ylabel('StopTimeMean')
#####
c = sns.countplot(x = 'Hour', data = trainWinter2, hue = 'City', palette = 'Blues', ax = axes9[2])
c.set_title('Winter')
c.set_ylabel('Count')

axt = axes9[2].twinx()
p = sns.pointplot(x='Hour', y='TotalTimeStopped_p80', data = trainWinter2, color = 'lightblue', ax
= axt )
p.set_ylabel('StopTimeMean')

plt.tight_layout()
```



For all three seasons, there are two notable peaks in average stop time. The first of these occurs at around 8 AM and the second occurs at around 5 PM. Seem familiar? This pattern suggests the commute to-and-from your typical 9-to-5 day job. It makes sense to observe peaks in traffic activity at these two times. There also seems to be a minimum in the amount of data available, centered at around 4 AM. This is also unsurprising, as most people are asleep at this time and so one wouldn't expect high levels of traffic. There doesn't seem to be a significant difference in stop times among the different seasons.

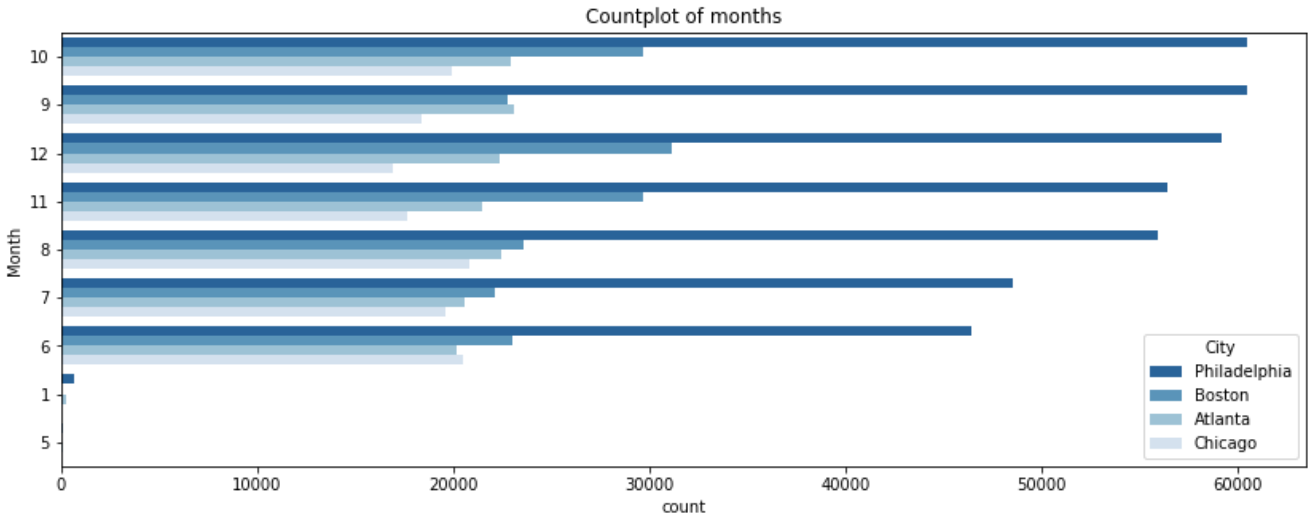
Let's take a closer look at the data contained in different months.

In [23]:

```
f11, axes11 = plt.subplots(figsize=(14, 5))
m = sns.countplot(y= 'Month', hue = 'City',data = train, palette = 'Blues_r', ax = axes11,
                  order = [10,9,12,11,8,7,6,1,5], hue_order=['Philadelphia','Boston','Atlanta', 'Chi
cago'])
#a = axes11.twinx()
#p = sns.pointplot(x='Month', y='TotalTimeStopped_p80', data =
train[train['Month'].isin([6,7,8,9,10,11,12])], color = 'lightblue', ax = a )
m.set_title('Countplot of months')
```

Out[23]:

```
Text(0.5, 1.0, 'Countplot of months')
```

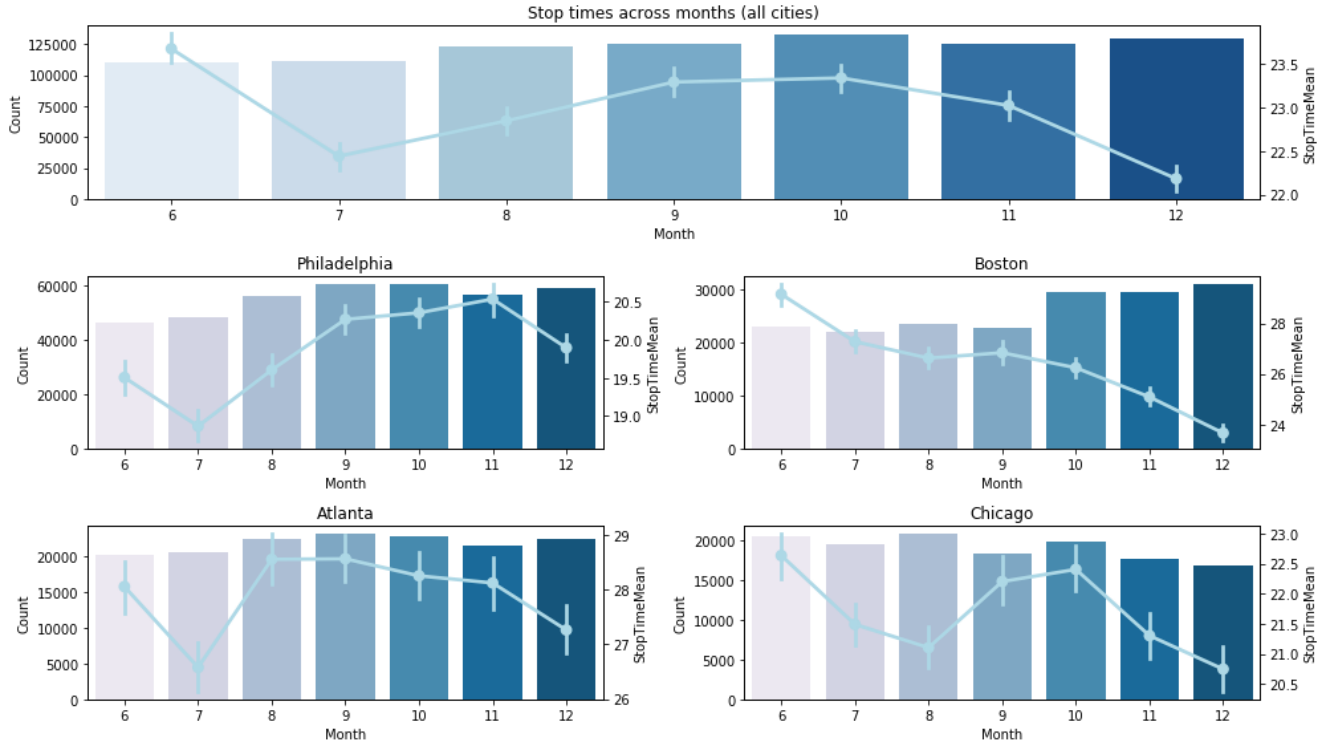


Upon closer inspection, it seems that there are disproportionately few data for the months of January and May compared to the other months. Before, we observed stop times over the time frame of hours and found some sensible trends. Now let's observe stop times over the time frame of months and see if we can find patterns on this larger time scale. We will exclude January and May since they don't contain much data.

In [24]:

```
trainP = train.loc[train['City'] == 'Philadelphia']
trainB = train.loc[train['City'] == 'Boston']
trainA = train.loc[train['City'] == 'Atlanta']
trainC = train.loc[train['City'] == 'Chicago']

f = plt.figure(figsize = (14,8))
s = plt.subplot2grid((3,2),(0,0), 1, 3,f)
p = plt.subplot2grid((3,2),(1,0), 1, 1,f)
b = plt.subplot2grid((3,2),(1,1), 1, 1,f)
a = plt.subplot2grid((3,2),(2,0), 1, 1,f)
c = plt.subplot2grid((3,2),(2,1), 1, 1,f)

m=sns.countplot(x = 'Month', data = train[train['Month'].isin([6,7,8,9,10,11,12])],
                palette = 'Blues', ax = s)
at = s.twinx()
point = sns.pointplot(x='Month', y='TotalTimeStopped_p80', data = train[train['Month'].isin([6,7,8,
9,10,11,12])],
                      color = 'lightblue', ax = at )
m.set_title('Stop times across months (all cities)')
```

```
m.set_ylabel('Count')
point.set_ylabel('StopTimeMean')
#####
temp = [[trainP,trainB,trainA,trainC],
        [p,b,a,c],
        ['Philadelphia', 'Boston', 'Atlanta', 'Chicago']]
for i in range(0,4):

    count = sns.countplot(x = 'Month', data = temp[0][i][temp[0][i]['Month'].isin([6,7,8,9,10,11,12]
)],
                  palette = 'PuBu', ax = temp[1][i])
    count.set_title(temp[2][i])
    count.set_ylabel('Count')
    at = temp[1][i].twinx()
    point = sns.pointplot(x='Month', y='TotalTimeStopped_p80', data = temp[0][i][temp[0][i]['Month'
].isin([6,7,8,9,10,11,12])],
                      color = 'lightblue', ax = at )

    point.set_ylabel('StopTimeMean')


plt.tight_layout()
```



Viewing the relationship between month and stop times for all cities, a minimum for the stop time can be seen in the data for July, after which the average stop time climbs upwards reaching a maximum at around October and starts to decrease once again towards the months of November and December. My guess is that as summer time approaches, a significant portion of people take breaks from work and maybe go on vacation, which would explain the minimum at July. And similar reasoning should also apply as we approach the holiday months near the end of the year, as evidenced by the second dip in stop times. The same general trend can be seen if we observe each of the cities individually, with some variance in where the minimum and maximum are for these specific cases. An indirect effect of season/monthly times on average stop time is thus observed. Direct effects of climate on traffic seem impossible to isolate from the given data alone.

Conclusions

- The data has only a small proportion of null values, located in EntryStreetName and ExitStreetName.
- There is no traffic data recorded for months February, March, and April, and a very low amount of data is recorded for the months of January and May compared to the other months with data.
- TotalTimeStopped turns out to be a useful metric for estimating traffic activity.
- Drivers drive straight through intersections without turning 0.70 of the time, and turn 0.30 of the time.
- Average stop times depend on whether or not a driver has made a turn at an intersection, with turning resulting in higher average stop times than not turning.
- Taking the above idea further, we found that stop time has an approximately linear relationship with the magnitude of the

angle of the turn.

- From our observation of differing distributions of turn-angle frequencies for each city, it seems that there are nontrivial infrastructural differences among the 4 cities.
- Due to the above, it would be quite difficult to distinguish the effects of these infrastructural differences from the effects of climate differences at different latitudes (if any) from the given data alone.

Modeling

In [25]:

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.multioutput import MultiOutputRegressor
import eli5
import math
import statistics
```

```
/opt/conda/lib/python3.6/site-packages/sklearn/externals/six.py:31: DeprecationWarning: The module
is deprecated in version 0.21 and will be removed in version 0.23 since we've dropped support for
Python 2.7. Please rely on the official version of six (https://pypi.org/project/six/).
  "(https://pypi.org/project/six/).", DeprecationWarning)
/opt/conda/lib/python3.6/site-packages/sklearn/externals/joblib/__init__.py:15:
DeprecationWarning: sklearn.externals.joblib is deprecated in 0.21 and will be removed in 0.23.
Please import this functionality directly from joblib, which can be installed with: pip install jo
blib. If this warning is raised when loading pickled models, you may need to re-serialize those mo
dels with scikit-learn 0.21+.
  warnings.warn(msg, category=DeprecationWarning)
Using TensorFlow backend.
```

Now we will try to do some modelling with simple linear regression. Reload data to undo any feature engineering/processing done in previous sections. We want to establish a baseline model with just the raw features so that we can compare the model's performance after feature engineering. Task: To predict the 20th, 50th, and 80th percentiles of the variables TotalTimeStopped and DistanceToFirstStop.

In [26]:

```
train = pd.read_csv('../input/bigquery-geotab-intersection-congestion/train.csv')
test = pd.read_csv('../input/bigquery-geotab-intersection-congestion/test.csv')
mergedData = train.merge(test, 'outer')
```

Summary of Data features:

- RowId: Unique identifier for each row/observation, each of which is an aggregate of some congestion data.
- IntersectionId: Identifier for specific intersection (2839 unique in train and test sets combined)
- Latitude
- Longitude
- EntryStreetName
- ExitStreetName
- EntryHeading: Direction cars entering intersection are moving in (N, W, E, S, NW, SW, SE, NE)
- ExitHeading: Direction cars exiting intersection are moving in (N, W, E, S, NW, SW, SE, NE)
- Hour: Hour of day (0-23)
- Weekend: No: 0, Yes: 1
- Month: Has unique values (1,5,6,7,8,9,10,11,12) in both train and test sets, which implies that the congestion data ranges from May of one year to January of the next year.
- Path: Concatenation of strings from the columns: EntryStreetName, EntryHeading, ExitStreetName, ExitHeading
- City: Atlanta, Boston, Chicago, or Philadelphia

LinearRegression can only take numerical inputs, so we'll have to do some basic label encoding for the categorical features.

In [28]:

```
#Select columns with non-numeric datatypes
objectCols = [col for col in train.columns if train[col].dtype == 'object']
objectCols
```

Out[28]:

```
['EntryStreetName',
 'ExitStreetName',
 'EntryHeading',
 'ExitHeading',
 'Path',
 'City']
```

In [29]:

```
for col in objectCols:
    le = LabelEncoder()
    mergedData[col] = mergedData[col].astype(str)
    le.fit(mergedData[col])
    traincopy[col] = le.transform(train[col])
    testcopy[col] = le.transform(test[col])
```

In [30]:

```
featureCols = [col for col in train.columns if col in test.columns]
targetCols = ['TotalTimeStopped_p20', 'TotalTimeStopped_p50', 'TotalTimeStopped_p80',
              'DistanceToFirstStop_p20', 'DistanceToFirstStop_p50', 'DistanceToFirstStop_p80']
```

Let's see how the linear regression model performs, using root mean squared error as the performance metric.

In [31]:

```
#Take list of feature names, a single target column, and fit LinearRegression model to the target
column
#10 times (10 fold cross validation), then return list of the 10 rmse scores

def get_scores(features, target, n_folds=10):
    scores = []
    kf = KFold(n_splits = n_folds, random_state=42)
    for trainindices, CVindices in kf.split(traincopy):
        lr = LinearRegression()
        X_train = traincopy.iloc[trainindices][features]
        y_train = traincopy.iloc[trainindices][target]
        X_cv = traincopy.iloc[CVindices][features]
        y_cv = traincopy.iloc[CVindices][target]

        lr.fit(X_train, y_train)
        pred = lr.predict(X_cv)
        mse = mean_squared_error(y_cv, pred)
        rmse = math.sqrt(mse)
        scores.append(rmse)

    return scores
```

Get the mean of (the list of 10 rmse scores returned, from above function), for each of the 6 target columns:

In [32]:

```
#Get mean of each run of get_scores function (6 target columns -> 6 runs total)
def getMeanScores(features):
    meanScores = []
    for i in tqdm.tqdm_notebook(targetCols):

        scorelist = get_scores(features = features, target = i, n_folds = 10)
        mean = statistics.mean(scorelist)
        meanScores.append(mean)
        print(str(i) + ': ' + str(mean) )
    return meanScores
```

In [33]:

```
meanScores = getMeanScores(features=featureCols)
print('Total mean: ' + str(statistics.mean(meanScores)))
```

```
TotalTimeStopped_p20: 6.849838407033481
TotalTimeStopped_p50: 15.065730377881964
TotalTimeStopped_p80: 27.102272546938988
DistanceToFirstStop_p20: 27.369379706571042
DistanceToFirstStop_p50: 69.32924379562886
DistanceToFirstStop_p80: 144.76595443158737

Total mean: 48.41373654427362
```

Now let's see the effect of adding two new features: isSameDirection and angleOfTurn

In [34]:

```
traincopy['isSameDirection'] = train['EntryHeading'] == train['ExitHeading']
traincopy['isSameDirection'] = traincopy['isSameDirection'].astype(int)

traincopy['angleOfTurn'] = train.apply(getAngleOfTurn, axis=1)
```

In [35]:

```
newFeatureCols = featureCols + ['isSameDirection', 'angleOfTurn']
meanScores2 = getMeanScores(features=newFeatureCols)
print('Total mean: ' + str(statistics.mean(meanScores2)))
```

```
TotalTimeStopped_p20: 6.799761046798703
TotalTimeStopped_p50: 14.836313912696005
TotalTimeStopped_p80: 26.517867349564668
DistanceToFirstStop_p20: 27.270302634366434
DistanceToFirstStop_p50: 69.15505252335282
DistanceToFirstStop_p80: 144.61287879948847

Total mean: 48.19869604437785
```

There is a slight decrease in the average rmse for each of the target columns, which indicates that our new features have had a helpful effect, albeit small.

Linear regression is a classic and is great for exploiting linear relationships between the features and targets. Most of our data here, however, has non-linear relationships with the targets. Additionally, although the simple label encoding done earlier served the purpose of converting categorical to numerical inputs for our model, there is a downside to this approach: It subtly conveys an ordinal aspect of our features to the model, when in reality the features may not be ordinal at all.

Now for a more serious modelling attempt, we will use the more modern lightGBM library which is better suited for dealing with nonlinear relationships and has also become famous in the Kaggle community for its efficiency at handling tabular data with high accuracy, even with large datasets containing both numerical and categorical inputs/outputs.

In [36]:

```
import lightgbm as lgb
from bayes_opt import BayesianOptimization
```

In [37]:

```
#Create 5 train sets (5-fold cv)
kf = KFold(n_splits = 5, random_state=42)
train_set = []
cv_set = []

for trainindices, CVindices in kf.split(traincopy):
    ##### train sets
    this_train_split=[]
    for i in range(0,6):

        traindf = traincopy.iloc[trainindices][newFeatureCols + [targetCols[i]]]
        traindf_lgb = lgb.Dataset(traindf, label = traindf[targetCols[i]])
        this_train_split.append(traindf_lgb)
```

```
    ##### CV sets
    this_cv_split=[]
    for i in range(0,6):

        CVdf = traincopy.iloc[CVindices][newFeatureCols + [targetCols[i]]]
        CVdf_lgb = lgb.Dataset(CVdf, label = CVdf[targetCols[i]])
        this_cv_split.append(CVdf_lgb)

    train_set.append(this_train_split)
    cv_set.append(this_cv_split)
```

In [38]:

```
%%time
params = {'application':'regression', 'num_iterations': 1000, 'metric':'rmse', 'DUSE_GPU':1}
results = {'TotalTimeStopped_p20':[],'TotalTimeStopped_p50':[],'TotalTimeStopped_p80':[],
           'DistanceToFirstStop_p20':[],'DistanceToFirstStop_p50':[],'DistanceToFirstStop_p80':[]}
for i in tqdm.tqdm_notebook(range(0,5)):
    for j in tqdm.tqdm_notebook(range(0,6)):
        a=lgb.train(params, train_set[i][j],valid_sets = cv_set[i][j],
                    verbose_eval = False, early_stopping_rounds=100)
        results[targetCols[j]].append(a.best_score['valid_0']['rmse'])

results
```

```
/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations`
in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

```
CPU times: user 11min 40s, sys: 16.3 s, total: 11min 57s
Wall time: 6min 16s
```

Out[38]:

```
{'TotalTimeStopped_p20': [0.37925844084565996,
  0.19909474868044838,
  0.0600561727145273,
  0.09022421668657822,
  0.13127036402350944],
 'TotalTimeStopped_p50': [0.4036533150051009,
  0.091883732297029,
  0.08078771657183798,
  0.015015869476685968,
  0.01162842371345911],
 'TotalTimeStopped_p80': [0.8020527543851551,
  0.6146327355378055,
  0.04002622142005416,
  0.15262198779550198,
  0.42777593091299904],
 'DistanceToFirstStop_p20': [6.5387694193005235,
  0.237126711996507,
  2.254565033113491,
  5.618012734232699,
  2.1312840589349737],
 'DistanceToFirstStop_p50': [18.476679810870394,
  2.046339005051225,
  6.066355094342038,
  8.874897356089775,
  7.6304478180141141],
 'DistanceToFirstStop_p80': [42.48157004994555,
  2.6910679590400393,
  8.771775604702334,
  11.053315569784436,
  16.986115511400406]}
```

In [47]:

```
resultsMean =  {'TotalTimeStopped_p20':-1,'TotalTimeStopped_p50':-1,'TotalTimeStopped_p80':-1,
           'DistanceToFirstStop_p20':-1,'DistanceToFirstStop_p50':-1,'DistanceToFirstStop_p80':-1}
for i in targetCols:
    resultsMean[i] = (statistics.mean(results[i]))

for i in resultsMean:
    print(i + ': ' +str(resultsMean[i]))

print('Total mean: ' + str(statistics.mean(resultsMean.values())))
# 4.845
```

```
TotalTimeStopped_p20: 0.17198078859014465
TotalTimeStopped_p50: 0.12059381141282259
TotalTimeStopped_p80: 0.40742192601030314
DistanceToFirstStop_p20: 3.355951591515639
DistanceToFirstStop_p50: 8.618943816873509
DistanceToFirstStop_p80: 16.396768938974553
Total mean: 4.845276812229495
```

LightGBM is substantially more suited for this task than simple linear regression, as shown in the results above. Decent performance was achieved even without any hyper-parameter tuning. Now let's see if we can improve the performance by adjusting a few of the hyperparameters using Bayesian optimization.

In [48]:

```
def lgb_eval(num_leaves, bagging_fraction, feature_fraction, lambda_l1, lambda_l2):

    params = {}
    params['application'] = 'regression'
    params['num_iterations'] = 1000
    params['num_leaves'] = int(round(num_leaves))
    #params['max_depth'] = int(round(max_depth))
    params['bagging_fraction'] = max(min(bagging_fraction, 1), 0)
    params['feature_fraction']= max(min(feature_fraction, 1), 0)
    params['lambda_l1'] = max(lambda_l1, 0)
    params['lambda_l2'] = max(lambda_l2, 0)
    params['metric'] = 'rmse'
    params['DUSE_GPU'] = 1

    results2 = {'TotalTimeStopped_p20':[],'TotalTimeStopped_p50':[],'TotalTimeStopped_p80':[],
           'DistanceToFirstStop_p20':[],'DistanceToFirstStop_p50':[],'DistanceToFirstStop_p80':[]}

    for i in tqdm.tqdm_notebook(range(0,5)):
        for j in range(0,6):

            b=lgb.train(params, train_set=train_set[i][j] , valid_sets = cv_set[i][j],verbose_eval =
False,
                        early_stopping_rounds=100)
            results2[targetCols[j]].append(b.best_score['valid_0']['rmse'])

    resultsMean2 =  {'TotalTimeStopped_p20':-1,'TotalTimeStopped_p50':-1,'TotalTimeStopped_p80':-1,
           'DistanceToFirstStop_p20':-1,'DistanceToFirstStop_p50':-1,'DistanceToFirstStop_p80':-1}
    for i in targetCols:
        resultsMean2[i] = (statistics.mean(results2[i]))
    for i in resultsMean2:
        print(i + ': ' +str(resultsMean2[i]))

    totalMean2 = statistics.mean(resultsMean2.values())

    print("Total mean: " + str(totalMean2))
    return (1/totalMean2)
    # Bayesian optimization will maximize (1/rmse) -> equivalent of minimizing rmse
```

In [49]:

```
lgbBO = BayesianOptimization(lgb_eval, {'num_leaves': (24, 45),

                                        'bagging_fraction': (0.8, 1),
                                        'feature_fraction': (0.1, 1),
                                        'lambda_l1': (0, 5),
                                        'lambda_l2': (0, 3),}, random_state=0)
```

In [50]:

```
%%time
lgbBO.maximize(n_iter=10)
```

```
|   iter    |  target   | baggin... | featur... | lambda_l1 | lambda_l2 | num_le... |
-------------------------------------------------------------------------------------


TotalTimeStopped_p20: 0.18272340050673727
TotalTimeStopped_p50: 0.16357621831247515
TotalTimeStopped_p80: 0.4575144725666447
DistanceToFirstStop_p20: 3.480626637796504
DistanceToFirstStop_p50: 8.597974946427787
DistanceToFirstStop_p80: 16.16168681478575
Total mean: 4.840683748399316
|  1        |  0.2066   |  0.9098   |  0.7437   |  3.014    |  1.635    |  32.9     |


TotalTimeStopped_p20: 0.2985338353226629
TotalTimeStopped_p50: 0.5067274045507314
TotalTimeStopped_p80: 1.0571939944872812
DistanceToFirstStop_p20: 3.872466959645213
DistanceToFirstStop_p50: 9.865173131584722
DistanceToFirstStop_p80: 19.34075420638062
Total mean: 5.823474921995205
|  2        |  0.1717   |  0.9292   |  0.4938   |  4.459    |  2.891    |  32.05    |


TotalTimeStopped_p20: 0.23131331754730688
TotalTimeStopped_p50: 0.38478143718909086
TotalTimeStopped_p80: 0.7715039121593154
DistanceToFirstStop_p20: 3.7320881005198476
DistanceToFirstStop_p50: 9.55959352273216
DistanceToFirstStop_p80: 18.595633257676226
Total mean: 5.545818924637325
|  3        |  0.1803   |  0.9583   |  0.576    |  2.84     |  2.777    |  25.49    |


TotalTimeStopped_p20: 0.2317890127914461
TotalTimeStopped_p50: 0.5688169790908889
TotalTimeStopped_p80: 1.1803780559093862
DistanceToFirstStop_p20: 3.3086365745527213
DistanceToFirstStop_p50: 9.469281909384856
DistanceToFirstStop_p80: 19.36477519122665
Total mean: 5.6872796204926575
|  4        |  0.1758   |  0.8174   |  0.1182   |  4.163    |  2.334    |  42.27    |


TotalTimeStopped_p20: 0.1732552279733954
TotalTimeStopped_p50: 0.14592125205302375
TotalTimeStopped_p80: 0.4260570530428972
DistanceToFirstStop_p20: 3.4073439760868407
DistanceToFirstStop_p50: 8.571486538924647
DistanceToFirstStop_p80: 16.246167268332957
Total mean: 4.82837188606896
|  5        |  0.2071   |  0.9957   |  0.8192   |  2.307    |  2.342    |  26.48    |

/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations` in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))

TotalTimeStopped_p20: 0.16827280217587104
TotalTimeStopped_p50: 0.12018238260589693
TotalTimeStopped_p80: 0.4058988925872991
DistanceToFirstStop_p20: 3.331916077546259
DistanceToFirstStop_p50: 8.52290936566614
DistanceToFirstStop_p80: 15.335960511462734
Total mean: 4.647523338674033
|  6        |  0.2152   |  1.0      |  1.0      |  0.0      |  3.0      |  44.66    |

/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations`
```

```
/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations`
in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

```
TotalTimeStopped_p20: 0.24702989980713816
TotalTimeStopped_p50: 0.5499157015122537
TotalTimeStopped_p80: 1.272152950387795
DistanceToFirstStop_p20: 3.355167653455503
DistanceToFirstStop_p50: 10.267175911163559
DistanceToFirstStop_p80: 19.306763532458053
Total mean: 5.833034274797384
|  7        |  0.1714   |  0.9697   |  0.1766   |  0.03354  |  2.946    |  32.04    |
```

```
/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations`
in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

```
TotalTimeStopped_p20: 0.1728452027603036
TotalTimeStopped_p50: 0.12463196528647315
TotalTimeStopped_p80: 0.4109269598270406
DistanceToFirstStop_p20: 3.393502086048626
DistanceToFirstStop_p50: 8.73266884200038
DistanceToFirstStop_p80: 15.771798697874253
Total mean: 4.767728958966179
|  8        |  0.2097   |  0.8071   |  0.9773   |  0.2169   |  0.00216  |  24.78    |
```

```
/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations`
in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

```
TotalTimeStopped_p20: 0.17359750925911638
TotalTimeStopped_p50: 0.12027781785514653
TotalTimeStopped_p80: 0.40653422301141845
DistanceToFirstStop_p20: 3.372296886136474
DistanceToFirstStop_p50: 8.672754941413384
DistanceToFirstStop_p80: 15.896477463959963
Total mean: 4.773656473605917
|  9        |  0.2095   |  1.0      |  1.0      |  5.0      |  0.0      |  45.0     |
```

```
/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations`
in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

```
TotalTimeStopped_p20: 0.22739916236591462
TotalTimeStopped_p50: 0.570882659162504
TotalTimeStopped_p80: 1.185727490122295
DistanceToFirstStop_p20: 3.306576403939611
DistanceToFirstStop_p50: 9.464275756985035
DistanceToFirstStop_p80: 19.434984403575232
Total mean: 5.698307646025099
|  10       |  0.1755   |  1.0      |  0.1      |  0.0      |  0.0      |  45.0     |
```

```
/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations`
in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

```
TotalTimeStopped_p20: 0.1752083703658822
TotalTimeStopped_p50: 0.11713557801405149
TotalTimeStopped_p80: 0.4072498159142112
DistanceToFirstStop_p20: 3.337333263929051
DistanceToFirstStop_p50: 8.708143775207521
DistanceToFirstStop_p80: 15.896924700037152
Total mean: 4.773665917244645
|  11       |  0.2095   |  1.0      |  1.0      |  4.638    |  0.0      |  28.36    |
```

```
/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations`
in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

```
TotalTimeStopped_p20: 0.174419674346285
TotalTimeStopped_p50: 0.125147274657081
TotalTimeStopped_p80: 0.4169625347206779
DistanceToFirstStop_p20: 3.394699942850605
DistanceToFirstStop_p50: 8.722413964361989
DistanceToFirstStop_p80: 16.04168609247288
Total mean: 4.812554913901586
|  12       |  0.2078   |  0.9066   |  0.9905   |  0.5271   |  0.2179   |  39.59    |
```

```
/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations`
in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

```
TotalTimeStopped_p20: 0.17219739684569743
TotalTimeStopped_p50: 0.12249183146970621
TotalTimeStopped_p80: 0.411733603893385
DistanceToFirstStop_p20: 3.3701502272484403
DistanceToFirstStop_p50: 8.68289720437846
DistanceToFirstStop_p80: 15.802871729913457
Total mean: 4.760390332291524
|  13       |  0.2101   |  0.8092   |  0.9835   |  0.7178   |  0.1998   |  30.1     |
```

```
/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations`
in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

```
TotalTimeStopped_p20: 0.1749084382524503
TotalTimeStopped_p50: 0.11929835442223041
TotalTimeStopped_p80: 0.4092931132303694
DistanceToFirstStop_p20: 3.3552342131114923
DistanceToFirstStop_p50: 8.618305987723089
DistanceToFirstStop_p80: 15.766383305736781
Total mean: 4.740570568746069
|  14       |  0.2109   |  1.0      |  1.0      |  5.0      |  0.0      |  35.23    |
```

```
/opt/conda/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `num_iterations`
in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

```
TotalTimeStopped_p20: 0.1715129405447633
TotalTimeStopped_p50: 0.12974462709494874
TotalTimeStopped_p80: 0.41159439850193213
DistanceToFirstStop_p20: 3.385558558714938
DistanceToFirstStop_p50: 8.620080106166492
DistanceToFirstStop_p80: 15.857019492667638
Total mean: 4.762585020615119
|  15       |  0.21     |  0.9364   |  0.9904   |  0.9031   |  2.974    |  35.99    |
=========================================================================================
CPU times: user 3h 56min 57s, sys: 4min 38s, total: 4h 1min 36s
Wall time: 2h 4min 27s
```

In [51]:

```
for i, res in enumerate(lgbBO.res):
    print("Iteration {}: \n\t{}".format(i, res))
```

```
Iteration 0:
 {'target': 0.20658238628596076, 'params': {'bagging_fraction': 0.909762700785465,
'feature_fraction': 0.7436704297351775, 'lambda_l1': 3.0138168803582195, 'lambda_l2':
1.6346495489906907, 'num_leaves': 32.896750786116996}}
Iteration 1:
 {'target': 0.17171877846043612, 'params': {'bagging_fraction': 0.9291788226133313,
'feature_fraction': 0.49382849013642327, 'lambda_l1': 4.4588650039103985, 'lambda_l2':
2.8909882815030876, 'num_leaves': 32.052271895341335}}
Iteration 2:
 {'target': 0.18031602069759178, 'params': {'bagging_fraction': 0.958345007616533,
'feature_fraction': 0.5760054277776141, 'lambda_l1': 2.8402228054696614, 'lambda_l2':
2.776789914877983, 'num_leaves': 25.491757222155627}}
Iteration 3:
 {'target': 0.17583098893129076, 'params': {'bagging_fraction': 0.8174258599403081,
```

```
'feature_fraction': 0.11819655769629316, 'lambda_l1': 4.16309922773969, 'lambda_l2':
2.3344702528495516, 'num_leaves': 42.270255113183204}}
Iteration 4:
 {'target': 0.20710915057832346, 'params': {'bagging_fraction': 0.9957236684465528,
'feature_fraction': 0.8192427077950513, 'lambda_l1': 2.3073968112646592, 'lambda_l2':
2.3415875288593666, 'num_leaves': 26.483762943247598}}
Iteration 5:
 {'target': 0.21516836541272885, 'params': {'bagging_fraction': 0.999999999999945,
'feature_fraction': 1.0, 'lambda_l1': 0.0, 'lambda_l2': 3.0, 'num_leaves': 44.66229796585494}}
Iteration 6:
 {'target': 0.17143736053817993, 'params': {'bagging_fraction': 0.9697084678655592,
'feature_fraction': 0.17661423952799987, 'lambda_l1': 0.0335364470853744, 'lambda_l2':
2.9463334557032965, 'num_leaves': 32.03751206659804}}
Iteration 7:
 {'target': 0.20974346667073063, 'params': {'bagging_fraction': 0.8070860435107429,
'feature_fraction': 0.9773056306490376, 'lambda_l1': 0.21685305526186804, 'lambda_l2':
0.002159870770371 83, 'num_leaves': 24.779825416896447}}
Iteration 8:
 {'target': 0.20948302533479574, 'params': {'bagging_fraction': 1.0, 'feature_fraction': 1.0,
'lambda_l1': 5.0, 'lambda_l2': 0.0, 'num_leaves': 45.0}}
Iteration 9:
 {'target': 0.17549070041831774, 'params': {'bagging_fraction': 1.0, 'feature_fraction': 0.1,
'lambda_l1': 0.0, 'lambda_l2': 0.0, 'num_leaves': 45.0}}
Iteration 10:
 {'target': 0.20948261091911496, 'params': {'bagging_fraction': 1.0, 'feature_fraction': 1.0,
'lambda_l1': 4.637762973457646, 'lambda_l2': 0.0, 'num_leaves': 28.35664258656271}}
Iteration 11:
 {'target': 0.20778983676869675, 'params': {'bagging_fraction': 0.9065912080957528,
'feature_fraction': 0.9905454105890101, 'lambda_l1': 0.5270630247242536, 'lambda_l2':
0.21785467868687924, 'num_leaves': 39.59464230537135}}
Iteration 12:
 {'target': 0.21006680759277713, 'params': {'bagging_fraction': 0.8091521046994323,
'feature_fraction': 0.9834717619554454, 'lambda_l1': 0.7177859690018651, 'lambda_l2':
0.19982410013076202, 'num_leaves': 30.100113082594916}}
Iteration 13:
 {'target': 0.2109450720115555, 'params': {'bagging_fraction': 1.0, 'feature_fraction': 1.0,
'lambda_l1': 4.999999985318554, 'lambda_l2': 0.0, 'num_leaves': 35.228078949544 54}}
Iteration 14:
 {'target': 0.2099700048758066, 'params': {'bagging_fraction': 0.9364350416968088,
'feature_fraction': 0.9903612710270754, 'lambda_l1': 0.9030546540520634, 'lambda_l2':
2.974102136659592, 'num_leaves': 35.98978666656229}}
```

In [52]:

```
lgbBO.max
```

Out[52]:

```
{'target': 0.21516836541272885,
 'params': {'bagging_fraction': 0.999999999999945,
  'feature_fraction': 1.0,
  'lambda_l1': 0.0,
  'lambda_l2': 3.0,
  'num_leaves': 44.66229796585494}}
```

After 15 iterations, Bayesian optimization has found the above parameters to be associated with the best performance, improving the base mean rmse of 4.845 to 4.648 (= 1/'target'). LightGBM supports implementation of ensembling through bootstrap aggregation of a fraction of the data ('bagging_fraction'), and/or a fraction of the feature columns ('feature_fraction') for each iteration. 'lambda_l1' and 'lambda_l2' are regularization parameters to prevent overfitting, and 'num_leaves' is the number of leaves in the gradient boosting tree, which also helps to control under/over fitting. Further improvements could still be made by exploring other hyperparameters and additional feature engineering, and even possibly incorporating external weather data (i.e. monthly rainfall in cities, etc.).