New York Taxi Ride Data Analysis

1 Predicting Taxi Ride Duration Using Data Science and Machine Learning

The main aim of this project is to create a regression model that predicts the travel time of a taxi ride in New York.

This project focus on these data science elements:

- The data science lifecycle: data selection and cleaning, EDA, feature engineering, and model selection.
- Using `sklearn` to process data and fit linear regression models.
- Embedding linear regression as a component in a more complex model.
- Exploring other modeling methods to try to improve further from the results received from linear regression predictions

[1]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
```

1.1 The Data

Attributes of all yellow taxi trips in January 2016 are published by the NYC Taxi and Limosine Commission.

The full data set takes a long time to download directly, so a simple random sample of the data is collected in `taxi.db`, a SQLite database. The code used to generate this sample is in the `taxi_sample.ipynb` file.

Columns of the `taxi` table in `taxi.db` include: - `pickup_datetime`: date and time when the meter was engaged - `dropoff_datetime`: date and time when the meter was disengaged - `pickup_lon`: the longitude where the meter was engaged - `pickup_lat`: the latitude where the meter was engaged - `dropoff_lon`: the longitude where the meter was disengaged - `dropoff_lat`: the latitude where the meter was disengaged - `passengers`: the number of passengers in the vehicle (driver entered value) - `distance`: trip distance - `duration`: duration of the trip in seconds

The goal will be to use data science tools and python programming to predict `duration` from the pick-up time, pick-up and drop-off locations, and distance.

1.2 Part 1: Data Selection and Cleaning

In this part, I limit the data to only trips that began and ended on Manhattan Island (map).

The below cell uses a SQL query to load the `taxi` table from `taxi.db` into a Pandas DataFrame called `all_taxi`.

It only includes trips that have **both** pick-up and drop-off locations within the boundaries of New York City:

- Longitude is between -74.03 and -73.75 (inclusive of both boundaries)
- Latitude is between 40.6 and 40.88 (inclusive of both boundaries)

[2]:
```
import sqlite3

conn = sqlite3.connect('taxi.db')
lon_bounds = [-74.03, -73.75]
lat_bounds = [40.6, 40.88]

c = conn.cursor()

my_string = 'SELECT * FROM taxi WHERE'

for word in ['pickup_lat', 'AND dropoff_lat']:
    my_string += ' {} BETWEEN {} AND {}'.format(word, lat_bounds[0],␣
 ↪lat_bounds[1])

for word in ['AND pickup_lon', 'AND dropoff_lon']:
    my_string += ' {} BETWEEN {} AND {}'.format(word, lon_bounds[0],␣
 ↪lon_bounds[1])

c.execute(my_string)

results = c.fetchall()

row_res = conn.execute('select * from taxi')
names = list(map(lambda x: x[0], row_res.description))


all_taxi = pd.DataFrame(results)
all_taxi.columns = names
all_taxi.head()
```

[2]:

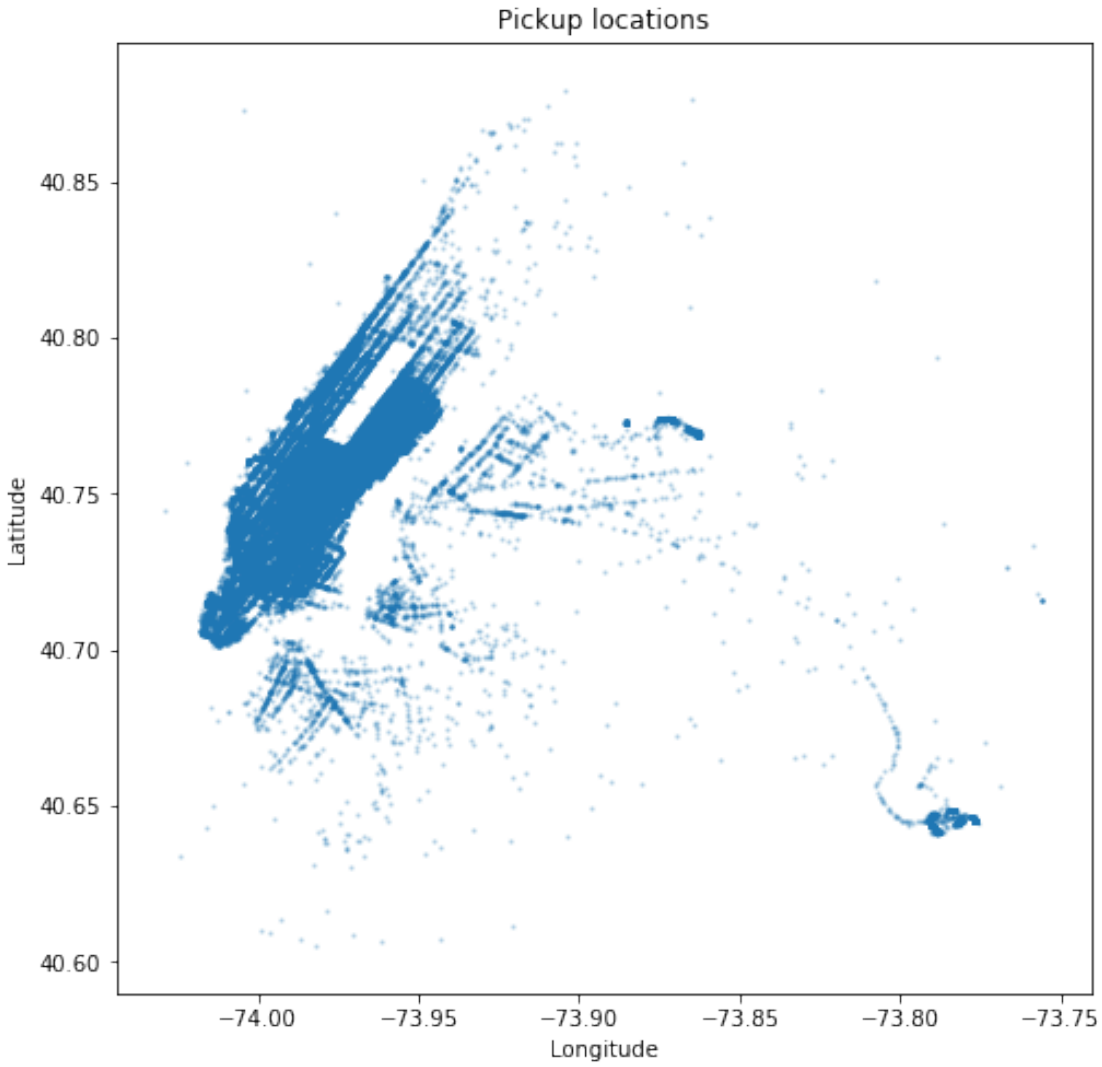
	pickup_datetime	dropoff_datetime	pickup_lon	pickup_lat	\
0	2016-01-30 22:47:32	2016-01-30 23:03:53	-73.988251	40.743542	
1	2016-01-04 04:30:48	2016-01-04 04:36:08	-73.995888	40.760010	
2	2016-01-07 21:52:24	2016-01-07 21:57:23	-73.990440	40.730469	
3	2016-01-01 04:13:41	2016-01-01 04:19:24	-73.944725	40.714539	
4	2016-01-08 18:46:10	2016-01-08 18:54:00	-74.004494	40.706989	

	dropoff_lon	dropoff_lat	passengers	distance	duration
0	-74.015251	40.709808	1	3.99	981

1	-73.975388	40.782200	1	2.03	320
2	-73.985542	40.738510	1	0.70	299
3	-73.955421	40.719173	1	0.80	343
4	-74.010155	40.716751	5	0.97	470

A scatter plot of pickup locations shows that most of them are on the island of Manhattan. The empty white rectangle is Central Park; cars are not allowed there.

[3]:
```
def pickup_scatter(t):
    plt.scatter(t['pickup_lon'], t['pickup_lat'], s=2, alpha=0.2)
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.title('Pickup locations')

plt.figure(figsize=(8, 8))
pickup_scatter(all_taxi)
```



The two small blobs outside of Manhattan with very high concentrations of taxi pick-ups are airports.

In this portion, I create a DataFrame called `clean_taxi` that only includes trips with a positive passenger count, a positive distance, a duration of at least 1 minute and at most 1 hour, and an average speed of at most 100 miles per hour.

[4]:
```
all_taxi.head()
```

[4]:

	pickup_datetime	dropoff_datetime	pickup_lon	pickup_lat	dropoff_lon	dropoff_lat	passengers	distance	duration
0	2016-01-30 22:47:32	2016-01-30 23:03:53	-73.988251	40.743542	-74.015251	40.709808	1	3.99	981
1	2016-01-04 04:30:48	2016-01-04 04:36:08	-73.995888	40.760010	-73.975388	40.782200	1	2.03	320
2	2016-01-07 21:52:24	2016-01-07 21:57:23	-73.990440	40.730469	-73.985542	40.738510	1	0.70	299
3	2016-01-01 04:13:41	2016-01-01 04:19:24	-73.944725	40.714539	-73.955421	40.719173	1	0.80	343
4	2016-01-08 18:46:10	2016-01-08 18:54:00	-74.004494	40.706989	-74.010155	40.716751	5	0.97	470

[5]:
```
clean_taxi = all_taxi[(all_taxi['passengers'] > 0) & (all_taxi['distance'] > 0)␣
↪& (all_taxi['duration'] >= 60) &
                      (all_taxi['duration'] <= 3600) & ((all_taxi['distance'] /␣
↪all_taxi['duration'] * 60 * 60) <= 100)]
```

1.2.1 Extracting data based on location using geocoded boundaries

In this portion, I explore the ability to collect the data that I want in a geometric form using python implementations of defining a polygon and verifying the points with longitude and latitude values that exist within that polygon.

Steps taken and references:

- Create a DataFrame called `manhattan_taxi` that only includes trips from `clean_taxi` that start and end within a polygon that defines the boundaries of Manhattan Island.
- The vertices of this polygon are defined in `manhattan.csv` as (latitude, longitude) pairs, which are published here.
- This page was a great reference in having an efficient way to test if a point is contained within a polygon. There are even implementations on that page (though not in Python).

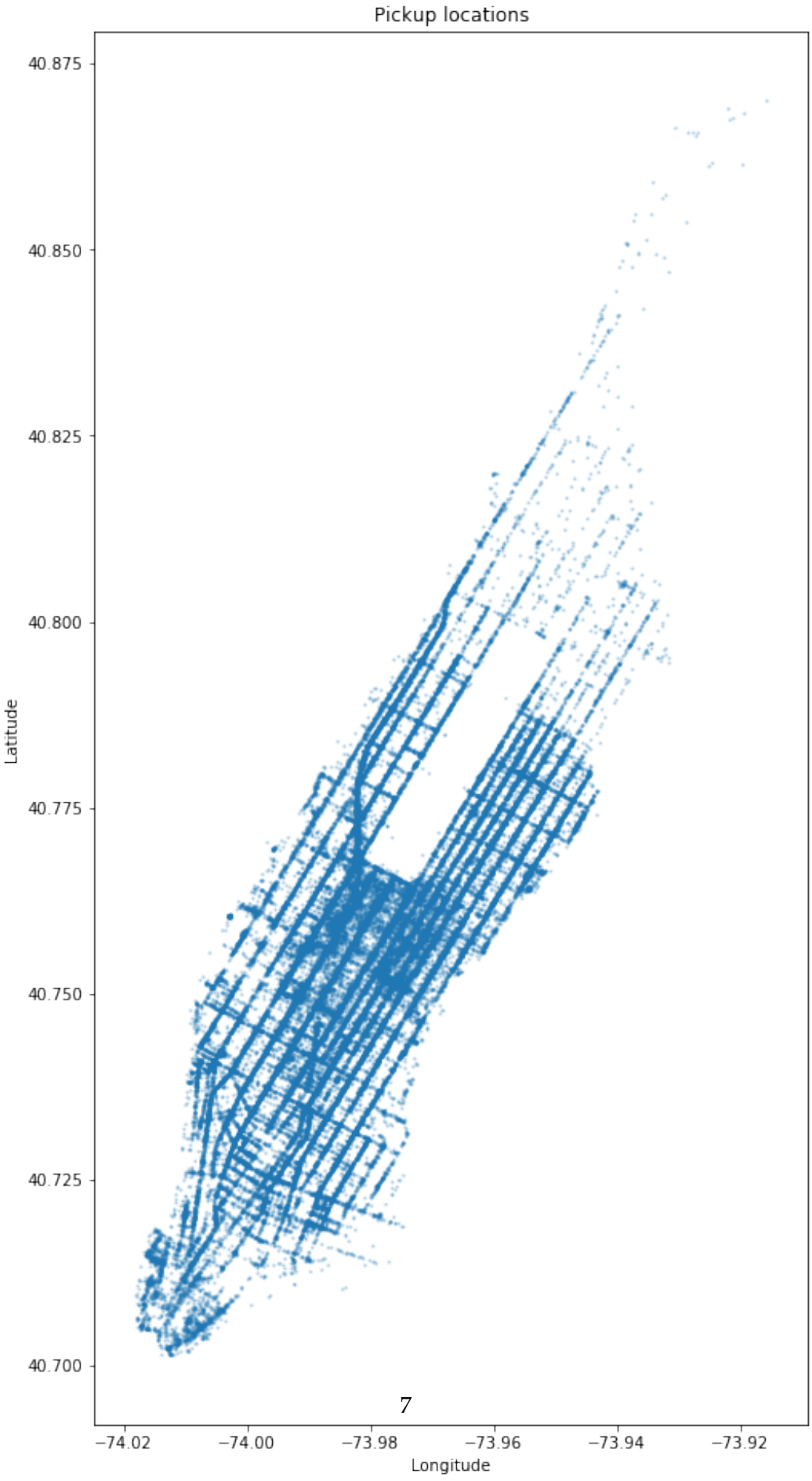
Because even with an efficient approach the process of checking each point can take several minutes, I first tested the code on a small sample to verify its accuracy. Then I proceeded to check on the full data.

[6]:
```
polygon = pd.read_csv('manhattan.csv')

# Recommended: First develop and test a function that takes a position
#              and returns whether it's in Manhattan.

polyCorners = 30 #how many corners the polygon has (no repeats)

polyX = polygon['lon']


polyY = polygon['lat']



#x, y         =  #point to be tested

#The following global arrays should be allocated before calling these functions:

constant = [None] * 30 #storage for precalculated constants (same size as polyX)
multiple = [None] * 30 #storage for precalculated multipliers (same size as␣
 ↪polyX)

def precalc_values():
    j = polyCorners-1

    for i in range(0, polyCorners):
        if(polyY[j] == polyY[i]):
            constant[i] = polyX[i]
            multiple[i] = 0
        else:
            constant[i]=polyX[i]-(polyY[i]*polyX[j])/
 ↪(polyY[j]-polyY[i])+(polyY[i]*polyX[i])/(polyY[j]-polyY[i])
            multiple[i]=(polyX[j]-polyX[i])/(polyY[j]-polyY[i])
        j = i

"""
def pointInPolygon(x, y):
    j = polyCorners-1
    oddNodes = False

    for i in range(0, polyCorners):
        if ((polyY[i]< y and polyY[j]>=y or polyY[j]< y and polyY[i]>=y)):
            oddNodes^=(y*multiple[i]+constant[i]<x)
        j=i

    return oddNodes
"""
```

```
precalc_values()

#test = pointInPolygon(-74.015251, 40.709808)

#print(test)

def in_manhattan(x, y):
    """Whether a longitude-latitude (x, y) pair is in the Manhattan polygon."""
    j = polyCorners-1
    oddNodes = False

    for i in range(0, polyCorners):
        if ((polyY[i]< y and polyY[j]>=y or polyY[j]< y and polyY[i]>=y)):
            oddNodes^=(y*multiple[i]+constant[i]<x)
        j=i

    return oddNodes

#print(type(clean_taxi['pickup_lon'].iloc[1]))

filtered_taxi = clean_taxi

#filtered_taxi.head(10)

# Recommended: Then, apply this function to every trip to filter clean_taxi.

#for i in range(0, clean_taxi.size):
for index, row in filtered_taxi.iterrows():
    if not (in_manhattan(clean_taxi['pickup_lon'].loc[index],␣
 ↪clean_taxi['pickup_lat'].loc[index])
            and in_manhattan(clean_taxi['dropoff_lon'].loc[index],␣
 ↪clean_taxi['dropoff_lat'].loc[index])):
        #print("dropped", index)
        filtered_taxi = filtered_taxi.drop(index)

manhattan_taxi = filtered_taxi
```

A scatter diagram of only Manhattan taxi rides has the familiar shape of Manhattan Island. This tests the correctness of our code above.

[7]:
```
plt.figure(figsize=(8, 16))
pickup_scatter(manhattan_taxi)
```


Pickup locations
40.875
40.850
40.825
40.800
40.775
40.750
40.725
40.700
Latitude
−74.02
−74.00
−73.98
−73.96
−73.94
−73.92
Longitude

1.2.2 Printing summary of data selection and cleaning

[8]:
```
total_trips = all_taxi.shape[0]
clean_diff = all_taxi.shape[0] - clean_taxi.shape[0]
percent_cleaned = clean_diff / all_taxi.shape[0]
clean_size = clean_taxi.shape[0]
manhattan_size = manhattan_taxi.shape[0]

print(f"Of the original {total_trips} trips, {clean_diff} anomalous trips␣
 ↪({percent_cleaned}) were removed through data cleaning \nleaving {clean_size}␣
 ↪clean trips. \n")
print(f"Then the {manhattan_size} trips within Manhattan were selected for␣
 ↪further analysis.")
```

```
Of the original 97692 trips, 1247 anomalous trips (0.01276460713262089) were
removed through data cleaning
leaving 96445 clean trips.

Then the 82800 trips within Manhattan were selected for further analysis.
```

1.3 Part 2: Exploratory Data Analysis

In this part, I choose which days to include as training data in your regression model.

My goal is to develop a general model that could potentially be used for future taxi rides. Though there is no guarantee that future distributions will resemble observed distributions, some effort to limit training data to typical examples can help ensure that the training data we have are representative of future observations.

January 2016 had some atypical days. New Year's Day (January 1) fell on a Friday. MLK Day was on Monday, January 18. A historic blizzard passed through New York that month. Using this dataset to train a general regression model for taxi trip times must account for these unusual phenomena, and one way to account for them is to remove atypical days from the training data.

1.3.1 Adding a "date" column

Here, I add a column labeled `date` to `manhattan_taxi` that contains the date (but not the time) of pickup, formatted as a `datetime.date` value (docs).

[9]:
```
from datetime import date


manhattan_taxi['date'] = manhattan_taxi['pickup_datetime'].str[0:10].apply(pd.
 ↪to_datetime).dt.date
#manhattan_taxi['date'] = manhattan_taxi['pickup_datetime'].str[0:10].apply(date.
 ↪fromisoformat)
```

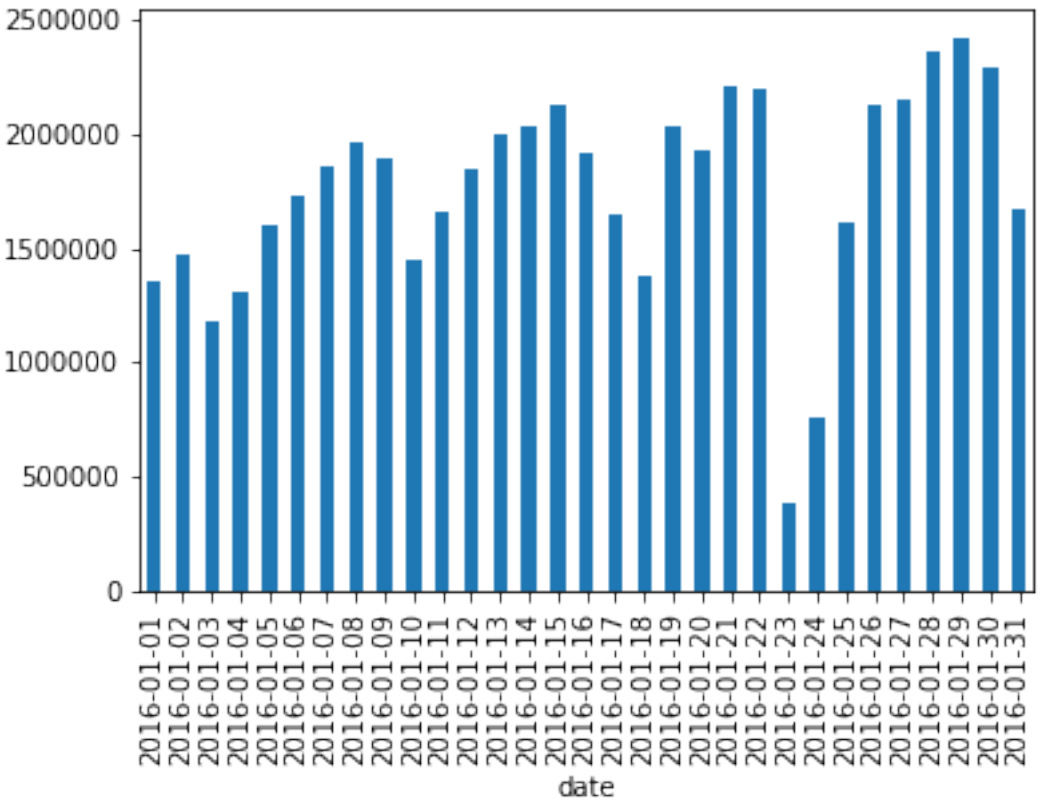
1.3.2 Data Visualization: Blizzard dates

Here I create a data visualization to identify which dates were affected by the historic blizzard of January 2016.

A blizzard would prevent a lot of people from going outside, which would reduce taxi usage. Therefore, I think trying to see a sum of duration would help visualize a blizzard because the days with a very significantly lower duration total could be affected by the blizzard.

[10]:
```
manhattan_grouped = manhattan_taxi.groupby(['date'])['duration'].sum()
manhattan_grouped.plot.bar()
```

[10]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x197d5489088>
```



The code below generates a list of dates that should have a fairly typical distribution of taxi rides, which excludes holidays and blizzards. The cell below assigns `final_taxi` to the subset of `manhattan_taxi` that is on these days.

For visual sake, the typical dates are printed in calendar form.

[11]:
```
import calendar
import re
```

```
from datetime import date

atypical = [1, 2, 3, 18, 23, 24, 25, 26]
typical_dates = [date(2016, 1, n) for n in range(1, 32) if n not in atypical]
typical_dates

print('Typical dates:\n')
pat = '  [1-3]|18 | 23| 24|25 |26 '
print(re.sub(pat, '    ', calendar.month(2016, 1)))

final_taxi = manhattan_taxi[manhattan_taxi['date'].isin(typical_dates)]
```

```
Typical dates:

    January 2016
Mo Tu We Th Fr Sa Su

 4  5  6  7  8  9 10
11 12 13 14 15 16 17
   19 20 21 22
      27 28 29 30 31
```

This is space for more exploratory data analysis, if I come back to explore other features or factors that might enhance the training data.

[12]:
```
# Maybe more EDA
```

1.4 Part 3: Feature Engineering

In this part, I create a design matrix (i.e., feature matrix) for a linear regression model. Similar to building pipelines, I will be adding features, removing labels, and scaling among other things.

I trip duration will be predicted from the following inputs: start location, end location, trip distance, time of day, and day of the week (*Monday, Tuesday, etc.*).

The process of transforming observations into a design matrix is expressed as a Python function called `design_matrix`, so that it's easy to make predictions for different samples in later parts of the project.

Since I have to look at the data in detail in order to define features, it seems best to split the data into training and test sets now, then only inspect the training set.

Below, I split the data into training and testing immediately.

[13]:
```
import sklearn.model_selection

train, test = sklearn.model_selection.train_test_split(
    final_taxi, train_size=0.8, test_size=0.2, random_state=42)
print('Train:', train.shape, 'Test:', test.shape)
```

```
Train: (53680, 10) Test: (13421, 10)
```

1.4.1 Creating boxplot to compare distributions of taxi trip durations

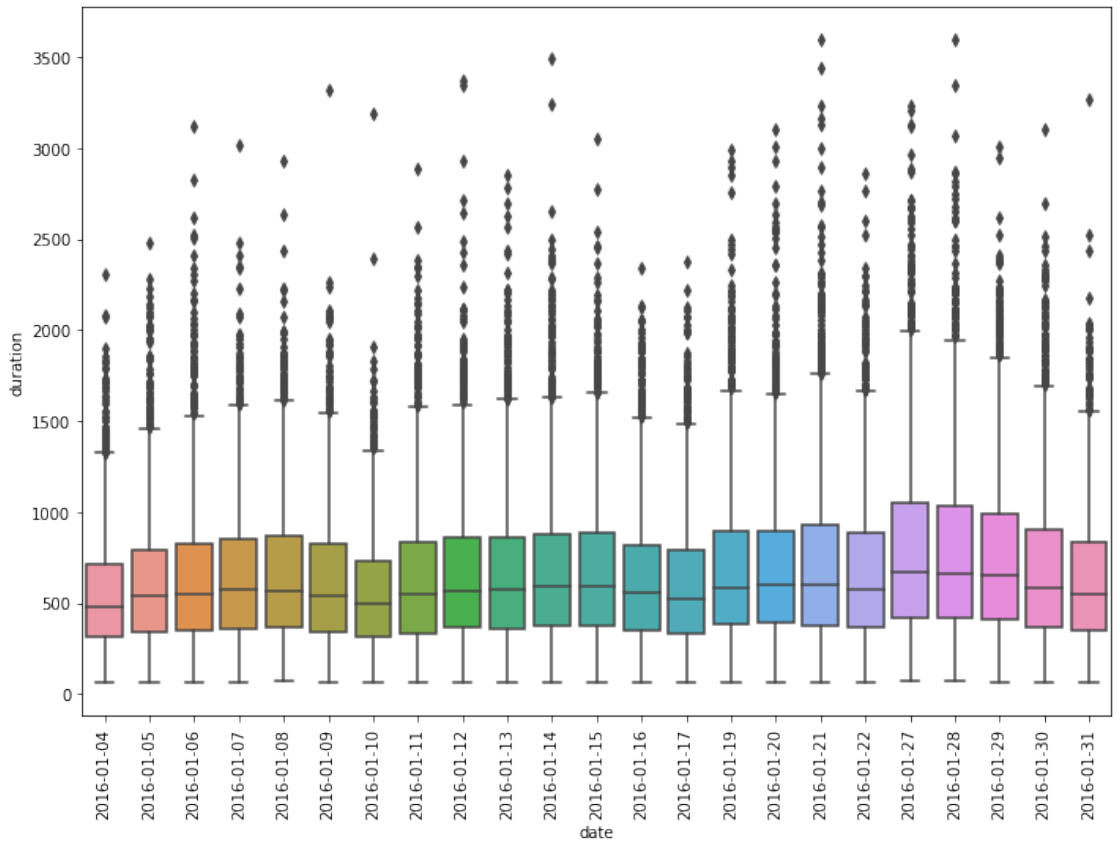
Here I create a boxplot that compares the distributions of taxi trip durations for each day **using the training data only**.

For the boxplot format, the individual dates appear on the horizontal axis, and duration values appear on the vertical axis.

The plot is generated using `sns.boxplot`

[14]:
```
axis_order = train['date'].unique()
axis_order.sort()
#print(axis_order)

fig, ax = plt.subplots(figsize=(11.7, 8.27))
plt.setp(ax.get_xticklabels(), rotation=90)
sns.boxplot(ax=ax, x="date", y="duration", data=train, order=axis_order)
```

[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x197d6aef888>
```



Using the boxplots and the calendar of typical dates that was generated above, it looks like generally on the weekends, the duration of the taxi trips were shorter than ones during the weekdays. A lot of the weekdays seem to have a fairly even trend with no specific day of the weekdays having a much higher/lower duration on average.

Below, the `augment` function adds these columns to a taxi ride dataframe:

- `hour`: The integer hour of the pickup time. E.g., a 3:45pm taxi ride would have `15` as the hour. A 12:20am ride would have `0`.
- `day`: The day of the week with Monday=0, Sunday=6.
- `weekend`: 1 if and only if the `day` is Saturday or Sunday.
- `period`: 1 for early morning (12am-6am), 2 for daytime (6am-6pm), and 3 for night (6pm-12pm).
- `speed`: Average speed in miles per hour.

[15]:
```
def speed(t):
    # Returning a column of speeds in miles per hour.
    return t['distance'] / t['duration'] * 60 * 60

def augment(t):
    # Augmenting a dataframe t with additional columns.
    u = t.copy()
    pickup_time = pd.to_datetime(t['pickup_datetime'])
    u.loc[:, 'hour'] = pickup_time.dt.hour
    u.loc[:, 'day'] = pickup_time.dt.weekday
    u.loc[:, 'weekend'] = (pickup_time.dt.weekday >= 5).astype(int)
    u.loc[:, 'period'] = np.digitize(pickup_time.dt.hour, [0, 6, 18])
    u.loc[:, 'speed'] = speed(t)
    return u

train = augment(train)
test = augment(test)
train.iloc[0,:] # printing an example row
```

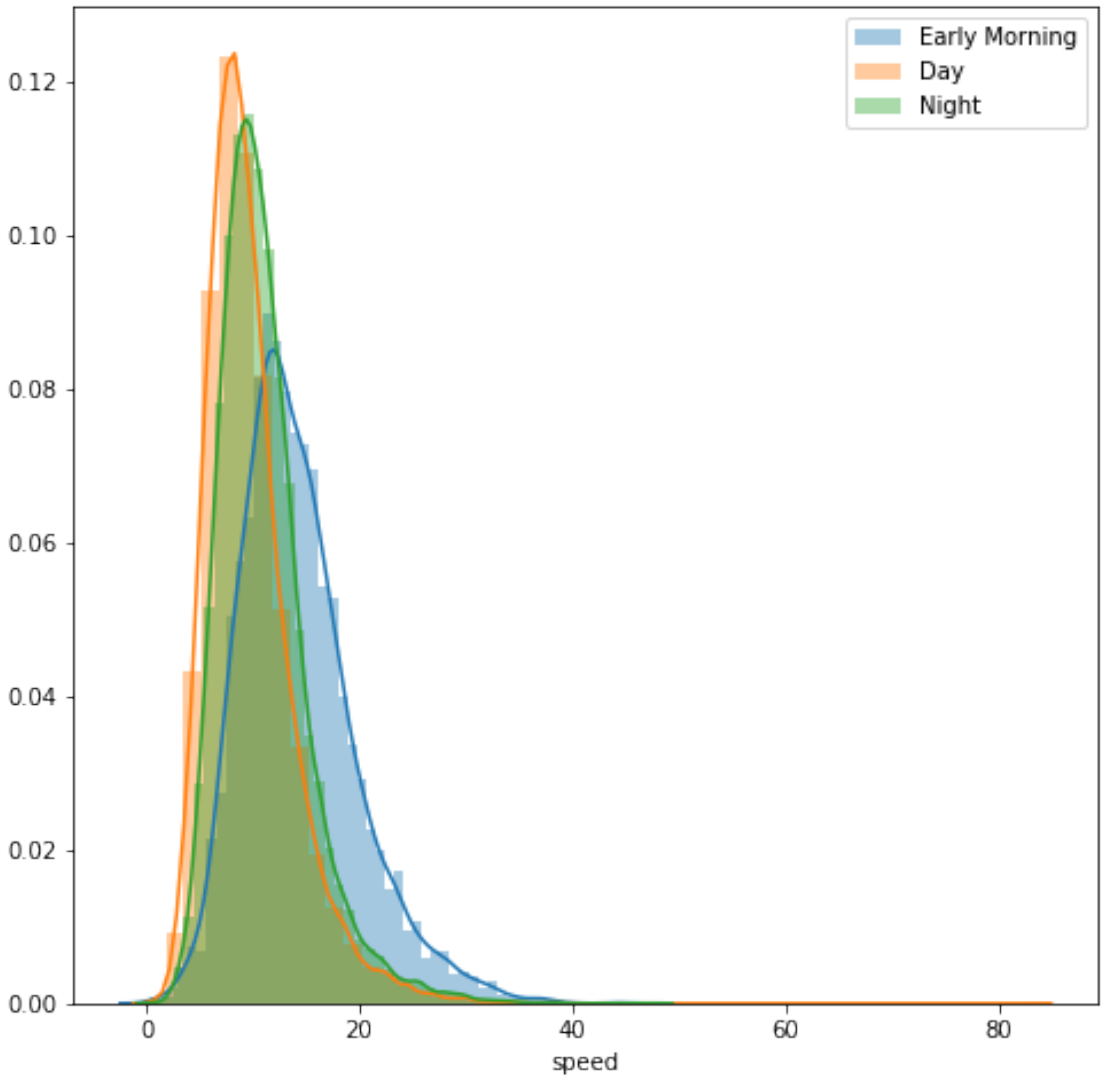
[15]:
```
pickup_datetime     2016-01-21 18:02:20
dropoff_datetime    2016-01-21 18:27:54
pickup_lon                     -73.9942
pickup_lat                       40.751
dropoff_lon                    -73.9637
dropoff_lat                     40.7711
passengers                            1
distance                           2.77
duration                           1534
date                         2016-01-21
hour                                 18
day                                   3
weekend                               0
period                                3
```

```
speed                            6.50065
Name: 16548, dtype: object
```

1.4.2 Comparing distribution of average speeds among different time periods

Using `sns.distplot`, I create an overlaid histogram comparing the distribution of average speeds for taxi rides that start in the early morning (12am-6am), day (6am-6pm; 12 hours), and night (6pm-12am; 6 hours).

[16]:
```
fig = plt.figure(figsize=(8,8))

x = train[train['period'] == 1]['speed']
x2 = train[train['period'] == 2]['speed']
x3 = train[train['period'] == 3]['speed']
sns.distplot(x, label="Early Morning")
sns.distplot(x2, label="Day")
sns.distplot(x3, label="Night")

#fig.legend(labels=['Early Morning','Day', 'Night'])
plt.legend()
plt.show()
```



The plot seems to suggest that the time of day **is** associated with the average speed of a taxi ride.

1.4.3 Using Principal Component Analysis to divide Manhattan

Manhattan can roughly be divided into Lower, Midtown, and Upper regions. To represent this division in our dataset, we can approximate by finding the first principal component of the pick-up location (latitude and longitude).

Principal component analysis (PCA) is a technique that finds new axes as linear combinations of your current axes. These axes are found such that the first returned axis (the first principal component) explains the most variation in values, the 2nd the second most, etc.

Here, I add a `region` column to `train` that categorizes each pick-up location as 0, 1, or 2 based on the value of each point's first principal component, such that an equal number of points fall into each region.

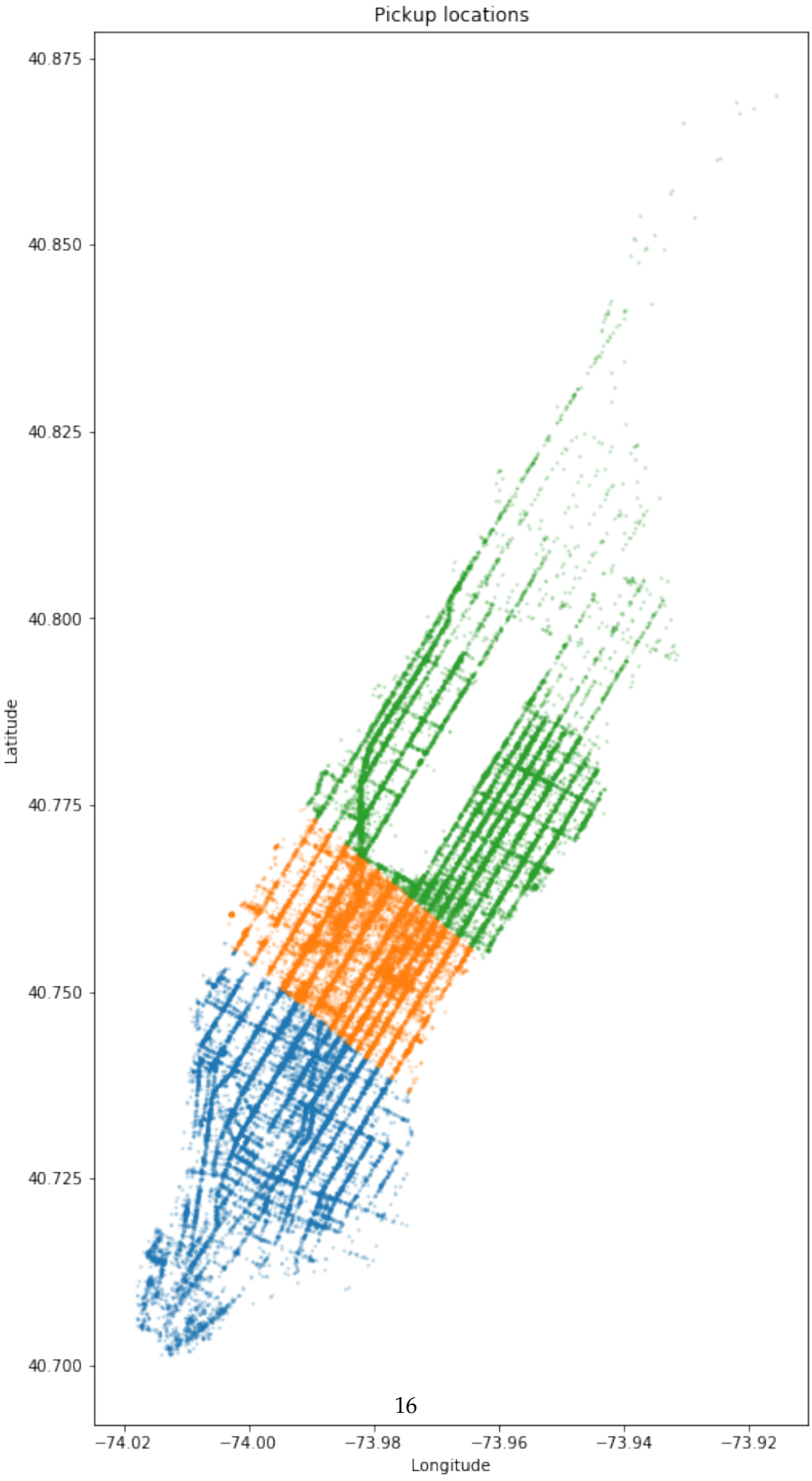
The documentation of `pd.qcut` was helpful in this portion, since it categorizes points in a distribution into equal-frequency bins.

Before implementing PCA, it is important to scale and shift the values. Therefore, the line with `np.linalg.svd` will return a transformation matrix, among other things. Then I use this matrix to convert points in (lat, lon) space into (PC1, PC2) space.

[17]:
```
# Find the first principle component
D = train[['pickup_lon', 'pickup_lat']]
pca_n = D.shape[0]
pca_means = D.mean(axis=0)
X = (D - pca_means) / np.sqrt(pca_n)
u, s, vt = np.linalg.svd(X, full_matrices=False)

v = np.linalg.inv(vt)

def add_region(t):
    # Adding a region column to t based on vt above.
    D = t[['pickup_lon', 'pickup_lat']]
    assert D.shape[0] == t.shape[0], 'set D using the incorrect table'
    # Always use the same data transformation used to compute vt
    X = (D - pca_means) / np.sqrt(pca_n)
    first_pc = (X.dot(v))[0]

    t.loc[:,'region'] = pd.qcut(first_pc, 3, labels=[0, 1, 2])

add_region(train)

add_region(test)
```

By drawing a plot to see how PCA divided the trips into three groups, we can see that these regions do roughly correspond to Lower Manhattan (below 14th street), Midtown Manhattan (between 14th and the park), and Upper Manhattan (bordering Central Park). This is interesting because we were able to create this division without referencing any information of the geography of New York.

[18]:
```
plt.figure(figsize=(8, 16))
for i in [0, 1, 2]:
    pickup_scatter(train[train['region'] == i])
```

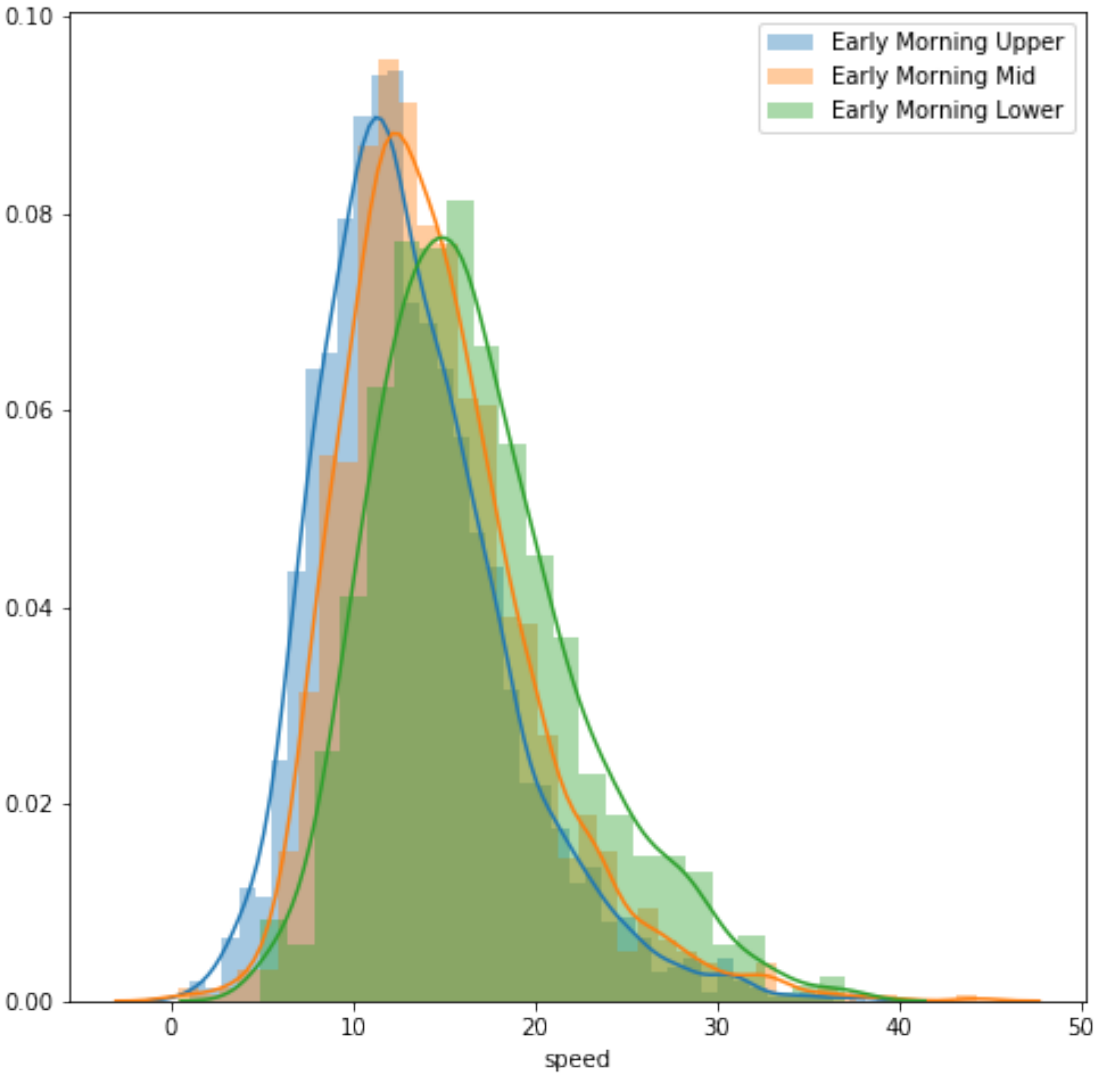

Pickup locations
40.875
40.850
40.825
40.800
40.775
40.750
40.725
40.700
Latitude
−74.02
−74.00
−73.98
−73.96
−73.94
−73.92
Longitude

1.4.4 Comparing speed of taxi rides with respect to times and regions

Using `sns.distplot`, I created three overlaid histogram comparing the distribution of speeds of the taxi rides for the three time period divisions from an earlier part (Early Morning, Day, Night), comparing the three different regions defined from the PCA portion.

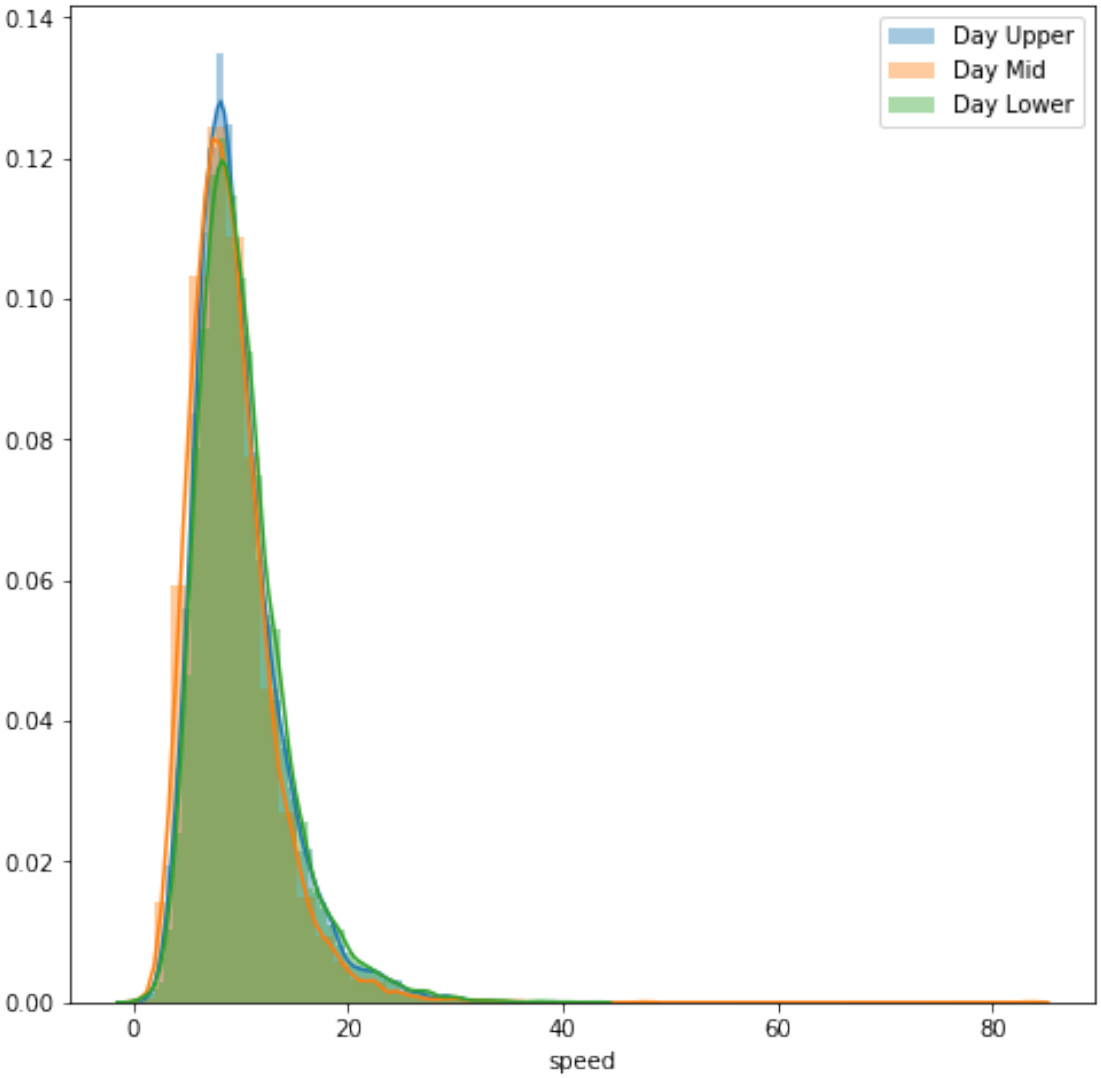
Interestingly, there seems to be the most variation between the speeds in the early morning, very little in the night, and basically the exact same among all three regions in daytime.

[19]:
```
fig = plt.figure(figsize=(8,8))



x1 = train[(train['period'] == 1) & (train['region'] == 0)]['speed']

x2 = train[(train['period'] == 1) & (train['region'] == 1)]['speed']

x3 = train[(train['period'] == 1) & (train['region'] == 2)]['speed']


sns.distplot(x1, label="Early Morning Upper")

sns.distplot(x2, label="Early Morning Mid")

sns.distplot(x3, label="Early Morning Lower")


plt.legend()
plt.show()
```



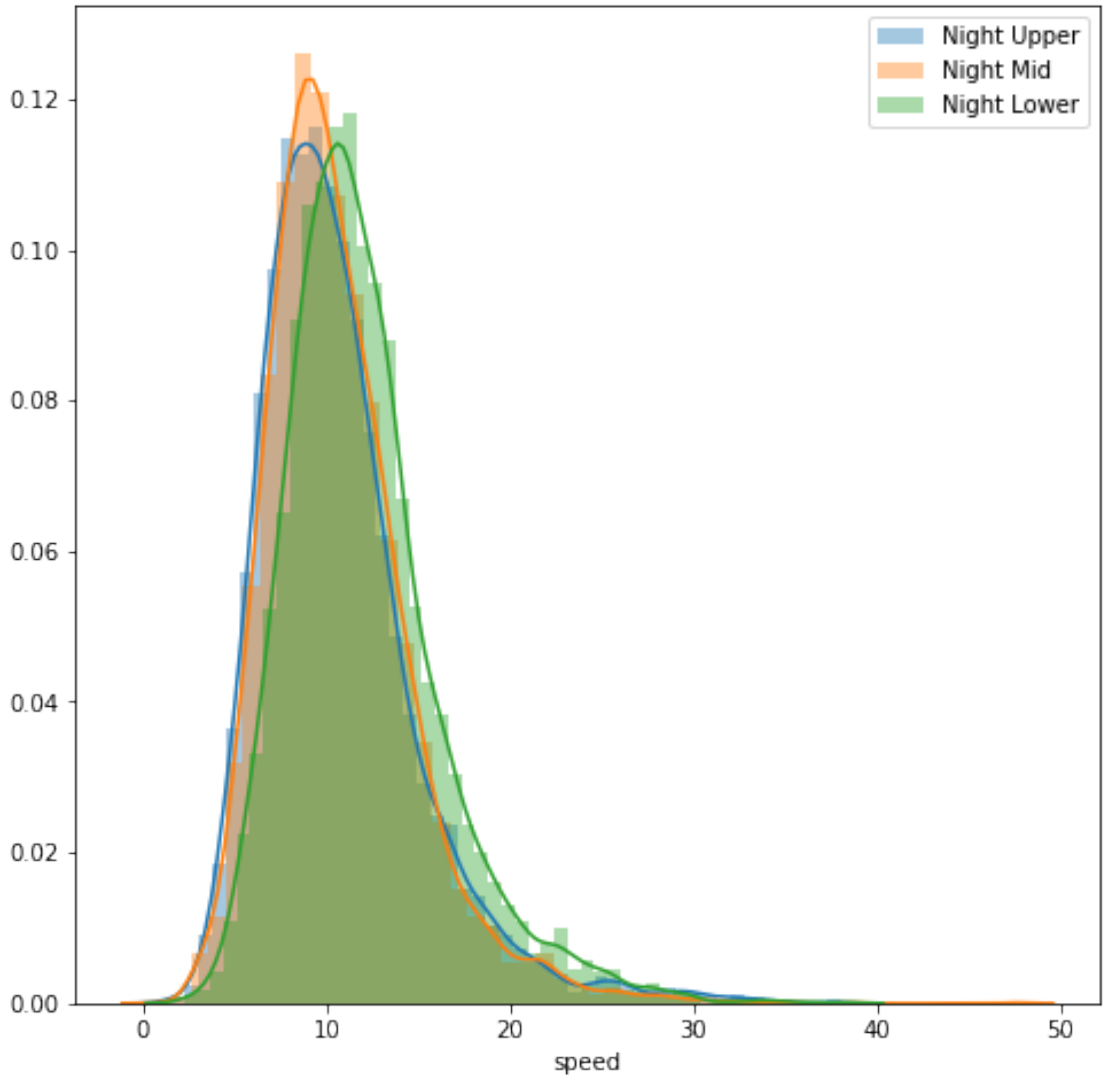
[20]:
```
fig = plt.figure(figsize=(8,8))



x1 = train[(train['period'] == 2) & (train['region'] == 0)]['speed']

x2 = train[(train['period'] == 2) & (train['region'] == 1)]['speed']

x3 = train[(train['period'] == 2) & (train['region'] == 2)]['speed']


sns.distplot(x1, label="Day Upper")

sns.distplot(x2, label="Day Mid")
```

```
sns.distplot(x3, label="Day Lower")


plt.legend()
plt.show()
```



[21]:
```
fig = plt.figure(figsize=(8,8))


x1 = train[(train['period'] == 3) & (train['region'] == 0)]['speed']
```

```
x2 = train[(train['period'] == 3) & (train['region'] == 1)]['speed']

x3 = train[(train['period'] == 3) & (train['region'] == 2)]['speed']


sns.distplot(x1, label="Night Upper")

sns.distplot(x2, label="Night Mid")

sns.distplot(x3, label="Night Lower")


plt.legend()
plt.show()
```



Finally, I create a design matrix that includes many of these features.

Quantitative features are converted to standard units, while categorical features are converted to dummy variables using one-hot encoding.

- The `period` is not included because it is a linear combination of the `hour`.
- The `weekend` variable is not included because it is a linear combination of the `day`.
- The `speed` is not included because it was computed from the `duration`; it's impossible to know the speed without knowing the duration, given that you know the distance.

[22]:
```
from sklearn.preprocessing import StandardScaler

num_vars = ['pickup_lon', 'pickup_lat', 'dropoff_lon', 'dropoff_lat', 'distance']
cat_vars = ['hour', 'day', 'region']

scaler = StandardScaler()
scaler.fit(train[num_vars])

def design_matrix(t):
    # Creating a design matrix from taxi ride dataframe t.
    scaled = t[num_vars].copy()
    scaled.iloc[:,:] = scaler.transform(scaled) # Convert to standard units
    categoricals = [pd.get_dummies(t[s], prefix=s, drop_first=True) for s in␣
 ↪cat_vars]
    return pd.concat([scaled] + categoricals, axis=1)

# This processes the full train set, then gives us the first item
# Use this function to get a processed copy of the dataframe passed in
# for training / evaluation
design_matrix(train).iloc[0,:]
```

[22]:
```
pickup_lon    -0.805821
pickup_lat    -0.171761
dropoff_lon    0.954062
dropoff_lat    0.624203
distance       0.626326
hour_1         0.000000
hour_2         0.000000
hour_3         0.000000
hour_4         0.000000
hour_5         0.000000
hour_6         0.000000
hour_7         0.000000
hour_8         0.000000
hour_9         0.000000
hour_10        0.000000
hour_11        0.000000
```

```
hour_12        0.000000
hour_13        0.000000
hour_14        0.000000
hour_15        0.000000
hour_16        0.000000
hour_17        0.000000
hour_18        1.000000
hour_19        0.000000
hour_20        0.000000
hour_21        0.000000
hour_22        0.000000
hour_23        0.000000
day_1          0.000000
day_2          0.000000
day_3          1.000000
day_4          0.000000
day_5          0.000000
day_6          0.000000
region_1       1.000000
region_2       0.000000
Name: 16548, dtype: float64
```

[23]:
```
processed_train = design_matrix(train)
processed_test = design_matrix(test)
```

1.5 Part 4: Model Selection

In this part, I select a regression model to predict the duration of a taxi ride.

The cells below are records of the different models tried and mean squared errors calculated using the different models in order to compare the results.

1.5.1 Constant RMSE

I assign `constant_rmse` to the root mean squared error on the **test** set for a constant model that always predicts the mean duration of all **training set** taxi rides.

[24]:
```
def rmse(errors):
    # Returning the root mean squared error.
    return np.sqrt(np.mean(errors ** 2))

constant_rmse = rmse(test['duration'] - train['duration'].mean())
constant_rmse
```

[24]:
```
399.1437572352677
```

1.5.2 Simple RSME

I assign `simple_rmse` to the root mean squared error on the test set for a simple linear regression model that uses only the distance of the taxi ride as a feature (and includes an intercept).

I use the `LinearRegression` model from `sklearn` to fit the parameters to data.

[25]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

model = LinearRegression()

model.fit(train[['distance']], train[['duration']])

preds = model.predict(test[['distance']])
mse = mean_squared_error(test[['duration']], preds)

simple_rmse = np.sqrt(mse)
simple_rmse
```

[25]:
```
276.7841105000342
```

1.5.3 Linear RMSE

I assign `linear_rmse` to the root mean squared error on the test set for a linear regression model fitted to the training set without regularization, using the design matrix defined by the `design_matrix` function from Part 3.

[26]:
```
model = LinearRegression()

model.fit(processed_train, train[['duration']])

preds = model.predict(processed_test)
mse = mean_squared_error(test[['duration']], preds)

linear_rmse = np.sqrt(mse)
linear_rmse
```

[26]:
```
255.19146631882757
```

1.5.4 Fitting an unregularized linear regression model for each period value

For each possible value of `period`, I fit an unregularized linear regression model to the subset of the training set in that `period`.

I assign `period_rmse` to the root mean squared error on the test set for a model that first chooses linear regression parameters based on the observed period of the taxi ride then predicts the duration using those parameters. Again, I fit to the training set and use the `design_matrix` function for features.

[27]:
```
model = LinearRegression()
errors = []

for v in np.unique(train['period']):
    model.fit(design_matrix(train[train['period'] == v]), train[train['period']
 ↪== v]['duration'])
    preds = model.predict(design_matrix(test[test['period'] == v]))
    curr_error = test[test['period'] == v]['duration'] - preds
    errors.extend(curr_error)

period_rmse = rmse(np.array(errors))
period_rmse
```

[27]:
```
246.62868831165176
```

This approach is a simple form of decision tree regression, where a different regression function is estimated for each possible choice among a collection of choices. In this case, the depth of the tree is only 1.

We see that the our `period_rmse` has performed better than `linear_rmse` depsite the design matrix for linear regression already including one feature for each possible hour. While it felt strange at first, after some thought, what I got was definitely a possible outcome. The period regression model could outperform linear regression because in the most basic sense, we have a tree regression which can do a better job at capturing some non-linear parts of the data by dividing the space into smaller sub-spaces. In our case, we are now training 3 models rather than just one which can help reduce some errors caused by the non-linearity.

1.5.5 Predicting with speed

Instead of predicting duration directly, an alternative is to predict the average *speed* of the taxi ride using linear regression, then compute an estimate of the duration from the predicted speed and observed distance for each ride.

I assign `speed_rmse` to the root mean squared error in the **duration** predicted by a model that first predicts speed as a linear combination of features from the `design_matrix` function, fitted on the training set, then predicts duration from the predicted speed and observed distance.

Speed is in miles per hour in the data, but duration is measured in seconds. Because of this, I perform basic operations to match the two units.

[28]:
```
model = LinearRegression()

model.fit(processed_train, train[['speed']])

preds_speed = model.predict(processed_test)

preds_dur = test[['distance']] / preds_speed * 3600

mse = mean_squared_error(test[['duration']], preds_dur)
```

```
speed_rmse = np.sqrt(mse)
speed_rmse
```

[28]: 243.0179836851495

Here, we can see that by predicting the speed and converting to duration, we reduced our rmse. Most likely, this suggests that our linear regression model is more suited to predicting the average speed data, and that our conversion from the average speed to duration is correct.

1.5.6 Combining the ideas of the two previous models

The function `tree_regression_errors` combines the ideas from the two previous models and generalizes to multiple categorical variables.

The `tree_regression_errors` function does the following: - Find a different linear regression model for each possible combination of the variables in `choices`; - Fit to the specified `outcome` (on train) and predict that `outcome` (on test) for each combination - Use the specified `error_fn` to compute the error in predicted duration using the predicted outcome; - Aggregate those errors over the whole test set and return them.

I found that including each of `period`, `region`, and `weekend` improves prediction accuracy, and that predicting speed rather than duration leads to more accurate duration predictions.

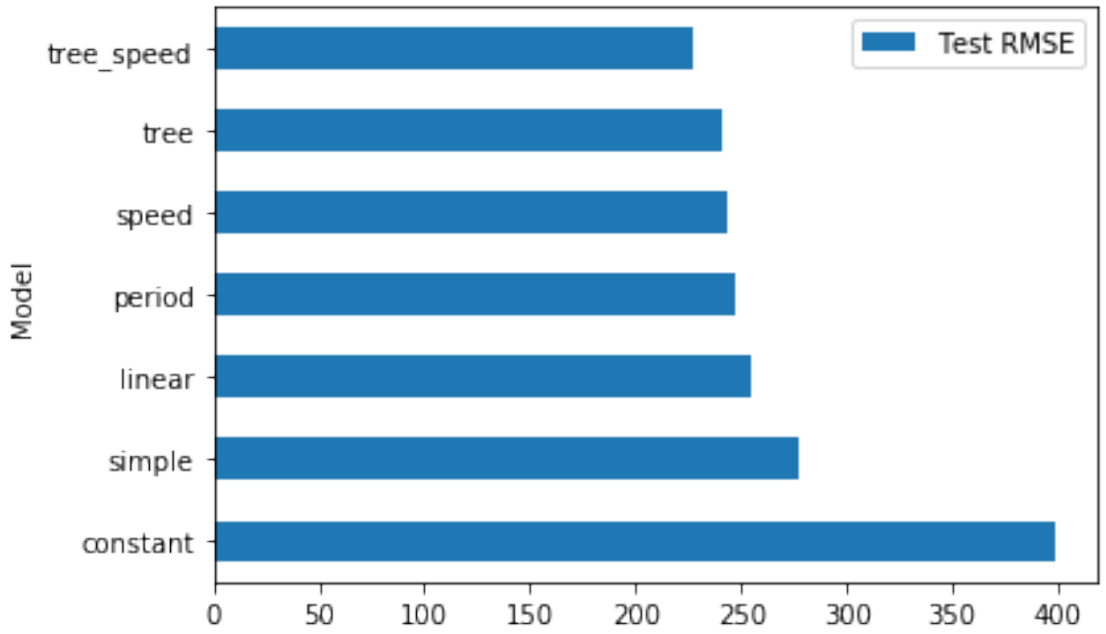
[29]:
```
model = LinearRegression()
choices = ['period', 'region', 'weekend']

def duration_error(predictions, observations):
    # Error between duration predictions (array) and observations (data frame)
    return predictions - observations['duration']

def speed_error(predictions, observations):
    # Duration error between speed predictions and duration observations
    preds_dur = observations['distance'] / predictions * 3600
    return preds_dur - observations['duration']

def tree_regression_errors(outcome='duration', error_fn=duration_error):
    # Return errors for all examples in test using a tree regression model.
    errors = []
    for vs in train.groupby(choices).size().index:
        v_train, v_test = train, test
        #print(vs)
        for v, c in zip(vs, choices):
            #print(v,c)
            v_train = v_train[v_train[c] == v]
            v_test = v_test[v_test[c] == v]
            #print(v_train.shape)
        model.fit(design_matrix(v_train), v_train[outcome])
```

```
        preds = model.predict(design_matrix(v_test))
        curr_error = error_fn(preds, v_test)
        errors.extend(curr_error)
    return errors

errors = tree_regression_errors()
errors_via_speed = tree_regression_errors('speed', speed_error)
tree_rmse = rmse(np.array(errors))
tree_speed_rmse = rmse(np.array(errors_via_speed))
print('Duration:', tree_rmse, '\nSpeed:', tree_speed_rmse)
```

```
Duration: 240.3395219270353
Speed: 226.90793945018308
```

Here's a summary of the results given in a bar graph form for easy comparison:

[30]:
```
models = ['constant', 'simple', 'linear', 'period', 'speed', 'tree',␣
 ↪'tree_speed']
pd.DataFrame.from_dict({
    'Model': models,
    'Test RMSE': [eval(m + '_rmse') for m in models]
}).set_index('Model').plot(kind='barh');
```



1.6 Part 5: Going even further beyond!!

In this part, I try to explore more regression model options to try to design one with the goal of achieving even higher performance than what I have gotten so far. The lower the RMSE the better

it would be.

1.6.1 Using tensorflow for a neural net model to use for training

In this part, I explore the tensorflow library. I chose the neural net model of tensorflow because I felt that it's complexity and recent popularity would provide much better results than the simple linear regressions of the previous attempts.

There are several Python code cells that only contain comments after I found them not necessary, but I have kept them in as a recordkeeping method of my learning and experimentation.

[31]:
```
#!pip install -q git+https://github.com/tensorflow/docs
```

[32]:
```
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers

#print(tf.__version__)
```

[33]:
```
import tensorflow_docs as tfdocs
import tensorflow_docs.plots
import tensorflow_docs.modeling
```

[34]:
```
def build_model():
  model = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=[len(processed_train.
 ↪keys())]),
    layers.Dense(64, activation='relu'),
    layers.Dense(1)
  ])

  optimizer = tf.keras.optimizers.RMSprop(0.001)

  model.compile(loss='mse',
                optimizer=optimizer,
                metrics=['mae', 'mse'])
  return model
```

[35]:
```
model = build_model()
```

[36]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                2368
```

```
_________________________________________________________________
dense_1 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
=================================================================
Total params: 6,593
Trainable params: 6,593
Non-trainable params: 0
_________________________________________________________________
```

[37]:
```
EPOCHS = 150
"""
history = model.fit(
  design_matrix(train), train['duration'],
  epochs=EPOCHS, validation_split = 0.2, verbose=0,
  callbacks=[tfdocs.modeling.EpochDots()])"""
```

[37]:
```
"\nhistory = model.fit(\n  design_matrix(train), train['duration'],\n
epochs=EPOCHS, validation_split = 0.2, verbose=0,\n
callbacks=[tfdocs.modeling.EpochDots()])"
```

[38]:
```
"""hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.tail()"""
```

[38]:
```
"hist = pd.DataFrame(history.history)\nhist['epoch'] =
history.epoch\nhist.tail()"
```

[39]:
```
"""loss, mae, mse = model.evaluate(design_matrix(test), test['duration'],␣
 ↪verbose=2)

print("Testing set Mean Abs Error: {:5.2f} MPG".format(mae))"""
```

[39]:
```
'loss, mae, mse = model.evaluate(design_matrix(test), test[\'duration\'],
verbose=2)\n\nprint("Testing set Mean Abs Error: {:5.2f} MPG".format(mae))'
```

[40]:
```
model = build_model()

# The patience parameter is the amount of epochs to check for improvement
early_stop = keras.callbacks.EarlyStopping(monitor='mse', patience=10)

early_history = model.fit(processed_train, train['duration'],
                    epochs=EPOCHS, validation_split = 0.2, verbose=0,
                    callbacks=[early_stop, tfdocs.modeling.EpochDots()])
```

```
Epoch: 0, loss:167078.2716,  mae:281.1115,  mse:167078.2031,
val_loss:67416.4786,  val_mae:186.0680,  val_mse:67416.4766,
```

```
...
...
Epoch: 100, loss:36027.8189,  mae:131.4822,  mse:36027.8203,
val_loss:36942.4717,  val_mae:132.0766,  val_mse:36942.4688,
...
```

[41]:
```
e_hist = pd.DataFrame(early_history.history)
e_hist['epoch'] = early_history.epoch
e_hist.tail()
```

[41]:

	loss	mae	mse	val_loss	val_mae	val_mse	epoch
145	35460.825619	130.493088	35460.863281	36597.503246	132.729965	36597.500000	145
146	35439.567072	130.463272	35439.550781	36865.913798	135.337418	36865.921875	146
147	35350.983111	130.405106	35350.976562	36794.892887	132.445007	36794.882812	147
148	35392.331269	130.337402	35392.328125	36730.580800	133.719086	36730.585938	148
149	35356.750490	130.181824	35356.761719	36069.251490	133.940933	36069.277344	149

1.6.2 Evaluating Performance

I printed a summary of the model's performance, including the RMSE on the train and test sets.

[42]:
```
loss, mae, mse = model.evaluate(design_matrix(test), test['duration'], verbose=2)

rmse_5 = np.sqrt(mse)

print("RMSE: ", rmse_5)
```

```
13421/13421 - 0s - loss: 37699.9570 - mae: 133.8874 - mse: 37699.9531
RMSE:  194.16476
```

1.6.3 Why I chose this model and brief summary of what I did

As you may see in the code, I chose to apply a neural network model in this to improve performance over the models in section 4. I chose neural net in attempt to improve the performance because we learned that neural nets usually perform better than most forms of linear regression because they can deal with non-linearities automatically. In section 4, we tried trees and switching the predicted metric (to 'speed') to try to improve the linear regression model as much as possible. Therefore, I decided that to go even further in performance in comparison to those improvements on the linear regression models, I needed to apply a different machine learning model, which I ended up choosing as neural net.

Within the neural net model, I played around with several parameters, such as the epoch and number of hidden neurons, but most of the time, the RMSE turned out better than the linear models regardless. Therefore, it was more of a matter to try to make the model train in a reasonable time frame while trying to maintain peak performance.

1.7 Notable Overview, Conclusion, and Final Thoughts

In Part 1 on data selection, I solved a domain-specific programming problem relevant to the analysis when choosing only those taxi rides that started and ended in Manhattan.

In Part 2 on EDA, I used the data to assess the impact of a historical event—the 2016 blizzard—and filtered the data accordingly.

In Part 3 on feature engineering, I used PCA to divide up the map of Manhattan into regions that roughly corresponded to the standard geographic description of the island.

In Part 4 on model selection, I found that using linear regression in practice can involve more than just choosing a design matrix. Tree regression made better use of categorical variables than linear regression. The domain knowledge that duration is a simple function of distance and speed allowed me to predict duration more accurately by first predicting speed.

In Part 5, I went further than simple linear regressions and experimented with neural nets using the tensorflow library with the goal to improve significantly on the previous linear regression models. It turns out, I was successful in reducing the RMSE by almost 30!

Overall, I felt that I was able to play around and explore a lot of data science tools and procedures while conducting this project. Starting from extracting relevant data to figuring out anomalies to finally performing the regression predictions, each step of this project felt meaningful and fun. Even seemingly small things such as predicting speed then converting to duration to increase accuracy truly proved the intricate world of data science, and I am excited on working on more projects to come.