proj3

March 2, 2020

[1]:
```
# Initialize autograder
# If you see an error message, you'll need to do
# pip3 install otter-grader
import otter
grader = otter.Notebook()
```

1 Project 3: Predicting Taxi Ride Duration

1.1 Due Date: Wednesday 3/4/20, 11:59PM

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the project, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** at the top of your notebook.

Collaborators: *list collaborators here*

1.2 Score Breakdown

Question	Points
1b	2
1c	3
1d	2
2a	1
2b	2
3a	2
3b	1
3c	2
3d	2
4a	2
4b	2
4c	2
4d	2
4e	2
4f	2
4g	4

Question	Points
5b	7
5c	3
Total	43

1.3 This Assignment

In this project, you will use what you've learned in class to create a regression model that predicts the travel time of a taxi ride in New York. Some questions in this project are more substantial than those of past projects.

After this project, you should feel comfortable with the following:

- 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.

First, let's import:

[2]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
```

1.4 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 we've placed a simple random sample of the data into `taxi.db`, a SQLite database. You can view the code used to generate this sample in the `taxi_sample.ipynb` file included with this project (not required).

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

Your goal will be to predict `duration` from the pick-up time, pick-up and drop-off locations, and distance.

1.5 Part 1: Data Selection and Cleaning

In this part, you will limit the data to 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)

You don't have to change anything, just run this cell.

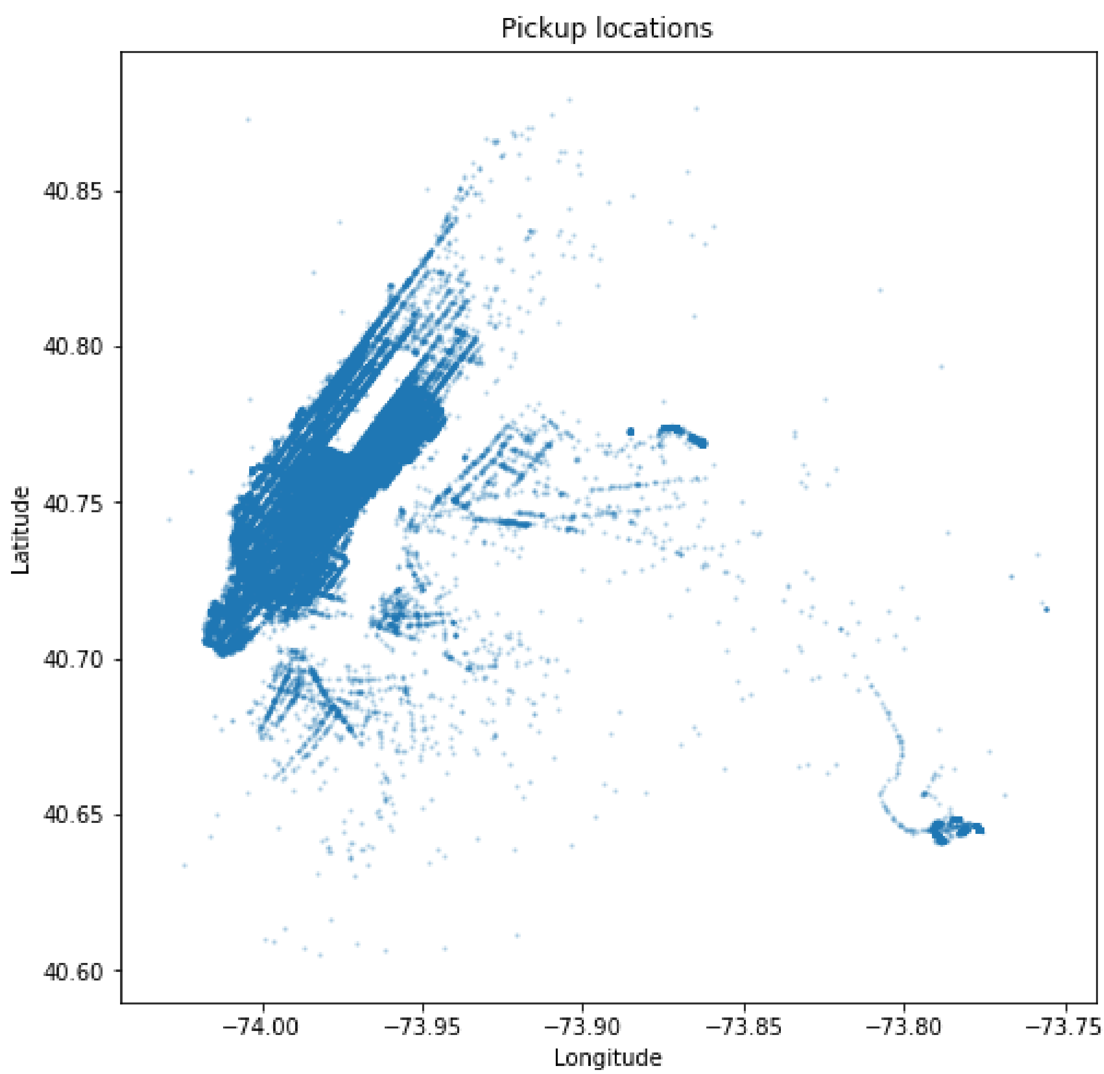
[3]:
```
import sqlite3

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()
```

[3]:
```
       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.

[4]:
```
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.

1.5.1 Question 1b

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. Inequalities should not be strict (e.g., <= instead of <) unless comparing to 0.

The provided tests check that you have constructed `clean_taxi` *correctly.*

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

[5]:
```
       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
```

[6]:
```
grader.check("q1b")
```

[6]:
```
All tests passed!
```

1.5.2 Question 1c (challenging)

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.

An efficient way to test if a point is contained within a polygon is described on this page. There are even implementations on that page (though not in Python). Even with an efficient approach, the process of checking each point can take several minutes. It's best to test your work on a small sample of `clean_taxi` before processing the whole thing. (To check if your code is working, draw a scatter diagram of the (lon, lat) pairs of the result; the scatter diagram should have the shape of Manhattan.)

The provided tests check that you have constructed `manhattan_taxi` correctly. It's not required that you implement the `in_manhattan` helper function, but that's recommended. If you cannot solve this problem, you can still continue with the project; see the instructions below the answer cell.

[7]:
```
polygon = pd.read_csv('manhattan.csv')



# //  Globals which should be set before calling these functions:
# //
# //  int    polyCorners  =  how many corners the polygon has (no repeats)
# //  float  polyX[]      =  horizontal coordinates of corners
# //  float  polyY[]      =  vertical coordinates of corners
# //  float  x, y         =  point to be tested
# //
# //  The following global arrays should be allocated before calling these␣
 ↪functions:
# //
# //  float  constant[] = storage for precalculated constants (same size as␣
 ↪polyX)
# //  float  multiple[] = storage for precalculated multipliers (same size as␣
 ↪polyX)
# //
# //  (Globals are used in this example for purposes of speed.  Change as
# //  desired.)
# //
# //  USAGE:
# //  Call precalc_values() to initialize the constant[] and multiple[] arrays,
# //  then call pointInPolygon(x, y) to determine if the point is in the polygon.
# //
# //  The function will return YES if the point x,y is inside the polygon, or
# //  NO if it is not.  If the point is exactly on the edge of the polygon,
# //  then the function may return YES or NO.
# //
# //  Note that division by zero is avoided because the division is protected
# //  by the "if" clause which surrounds it.
polyCorners = len(polygon)
polyX = polygon['lon']
polyY = polygon['lat']
```

```
constant = np.zeros(polyCorners)
multiple = np.zeros(polyCorners)

# void precalc_values() {

#   int   i, j=polyCorners-1 ;

#   for(i=0; i<polyCorners; i++) {
#     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 precalc_values():
    j = polyCorners - 1
    for i in range(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

precalc_values()

# bool pointInPolygon() {

#   bool oddNodes=NO, current=polY[polyCorners-1]>y, previous;
#   for (int i=0; i<polyCorners; i++) {
#     previous=current; current=polyY[i]>y; if (current!=previous)␣
 ↪oddNodes^=y*multiple[i]+constant[i]<x; }
#   return oddNodes; }

# Recommended: First develop and test a function that takes a position
#              and returns whether it's in Manhattan.
def in_manhattan(x, y):
    """Whether a longitude-latitude (x, y) pair is in the Manhattan polygon."""
    oddNodes = False
    current = polyY[polyCorners - 1] > y

    for i in range(polyCorners):
```

```
        previous = current
        current = polyY[i] > y
        if current != previous:
            oddNodes ^= y * multiple[i] + constant[i] < x

    return oddNodes

# Recommended: Then, apply this function to every trip to filter clean_taxi.
manhattan_taxi = clean_taxi[
    clean_taxi.apply(lambda row: in_manhattan(row['pickup_lon'],␣
 ↪row['pickup_lat']), axis=1) &
    clean_taxi.apply(lambda row: in_manhattan(row['dropoff_lon'],␣
 ↪row['dropoff_lat']), axis=1)
]
```

[8]:
```
grader.check("q1c")
```

[8]:

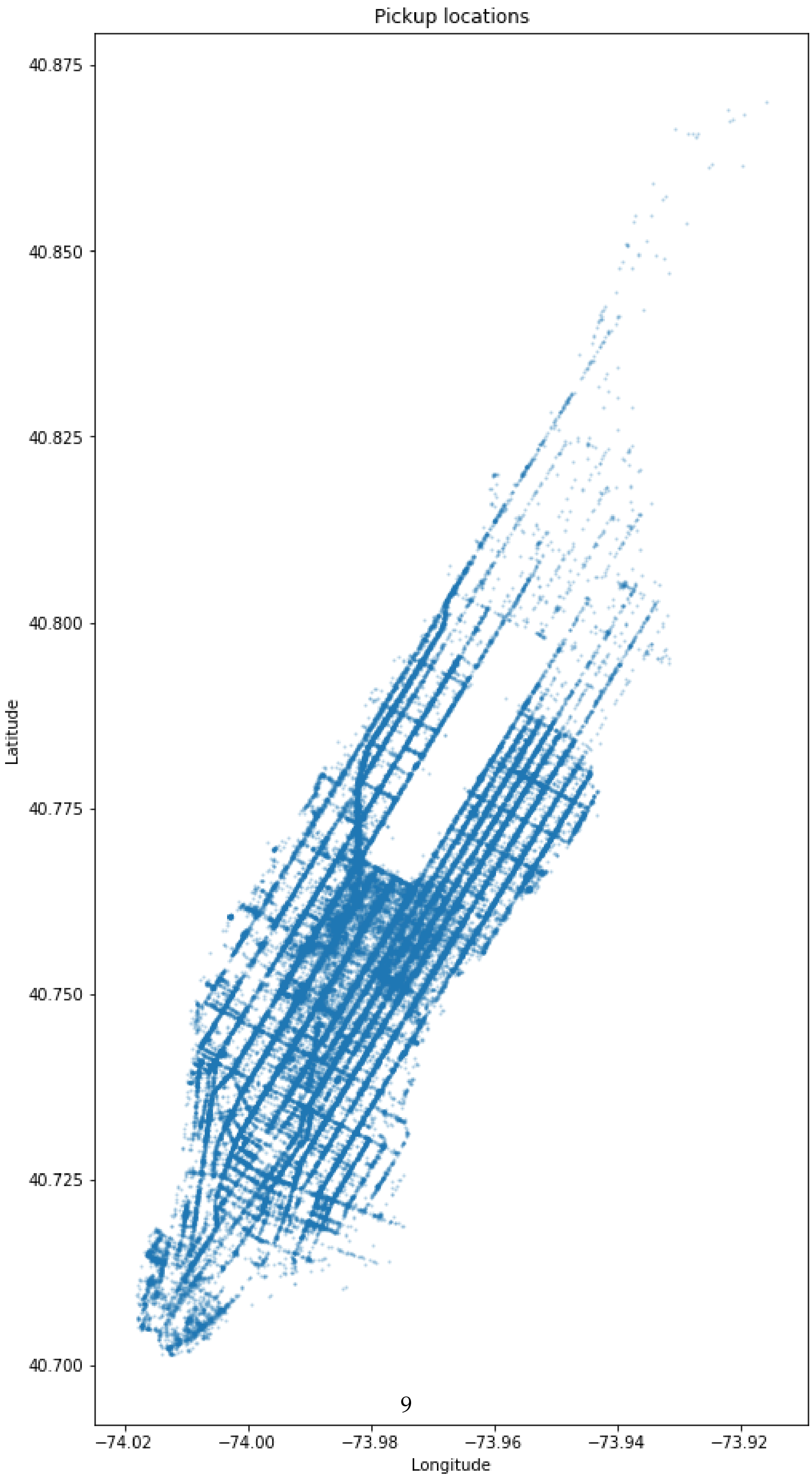
```
All tests passed!
```

If you are unable to solve the problem above, have trouble with the tests, or want to work on the rest of the project before solving it, run the following cell to load the cleaned Manhattan data directly. (Note that you may not solve the previous problem just by loading this data file; you have to actually write the code.)

[9]:
```
manhattan_taxi = pd.read_csv('manhattan_taxi.csv')
```

A scatter diagram of only Manhattan taxi rides has the familiar shape of Manhattan Island.

[10]:
```
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

1.5.3 Question 1d

Print a summary of the data selection and cleaning you performed. **Your Python code should not include any number literals, but instead should refer to the shape of `all_taxi`, `clean_taxi`, and `manhattan_taxi`.**

E.g., you should print something like: "Of the original 1000 trips, 21 anomalous trips (2.1%) were removed through data cleaning, and then the 600 trips within Manhattan were selected for further analysis."

(Note that the numbers in the example above are not accurate.)

One way to do this is with Python's f-strings. For instance,

```
name = "Joshua"
print(f"Hi {name}, how are you?")
```

prints out `Hi Joshua, how are you?`.

Please ensure that your Python code does not contain any very long lines, or we can't grade it.

Your response will be scored based on whether you generate an accurate description and do not include any number literals in your Python expression, but instead refer to the dataframes you have created.

[11]:
```
print(f'all_taxi shape: {all_taxi.shape}')
print(f'clean_taxi shape: {clean_taxi.shape}')
print(f'manhattan_taxi shape: {manhattan_taxi.shape}')
print('\n')

num_original_trips = all_taxi.shape[0]
num_anomalous_trips = num_original_trips - clean_taxi.shape[0]
anomalous_trips_percentage = round(num_anomalous_trips*100/num_original_trips, 2)
num_manhattan_trips = manhattan_taxi.shape[0]

print(f'Of the original {num_original_trips} trips, \
{num_anomalous_trips} anamalous trips ({anomalous_trips_percentage}%) were \
removed through data cleaning, and then {num_manhattan_trips} within \
Manhattan were selected for further analysis.'
)
```

```
all_taxi shape: (97692, 9)
clean_taxi shape: (96445, 9)
manhattan_taxi shape: (82800, 9)


Of the original 97692 trips, 1247 anamalous trips (1.28%) were removed through
data cleaning, and then 82800 within Manhattan were selected for further
analysis.
```

1.6 Part 2: Exploratory Data Analysis

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

Your goal is to develop a general model that could potentially be used for future taxi rides. There is no guarantee that future distributions will resemble observed distributions, but some effort to limit training data to typical examples can help ensure that the training data 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.

1.6.1 Question 2a

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).

The provided tests check that you have extended `manhattan_taxi` correctly.

[12]:
```
manhattan_taxi.head()
```

[12]:
```
        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  2016-01-08 18:46:10  2016-01-08 18:54:00  -74.004494   40.706989
4  2016-01-02 12:39:57  2016-01-02 12:53:29  -73.958214   40.760525

   dropoff_lon  dropoff_lat  passengers  distance  duration
0   -74.015251    40.709808           2      3.99       981
1   -73.975388    40.782200           1      2.03       320
2   -73.985542    40.738510           1      0.70       299
3   -74.010155    40.716751           5      0.97       470
4   -73.983360    40.760406           1      1.70       812
```

[13]:
```
from datetime import datetime

# https://www.tutorialspoint.com/python/time_strptime.htm
# print(datetime.strptime('2016-01-30 22:47:32', '%Y-%m-%d %H:%M:%S').date())
manhattan_taxi['date'] = manhattan_taxi.apply(lambda row: datetime.
 ↪strptime(row['pickup_datetime'], '%Y-%m-%d %H:%M:%S').date(), axis=1)
manhattan_taxi.head()
```

[13]:
```
        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  2016-01-08 18:46:10  2016-01-08 18:54:00  -74.004494   40.706989
```

```
4  2016-01-02 12:39:57  2016-01-02 12:53:29  -73.958214   40.760525

   dropoff_lon  dropoff_lat  passengers  distance  duration        date
0   -74.015251    40.709808           2      3.99       981  2016-01-30
1   -73.975388    40.782200           1      2.03       320  2016-01-04
2   -73.985542    40.738510           1      0.70       299  2016-01-07
3   -74.010155    40.716751           5      0.97       470  2016-01-08
4   -73.983360    40.760406           1      1.70       812  2016-01-02
```

[14]:
```
grader.check("q2a")
```

[14]:

```
All tests passed!
```

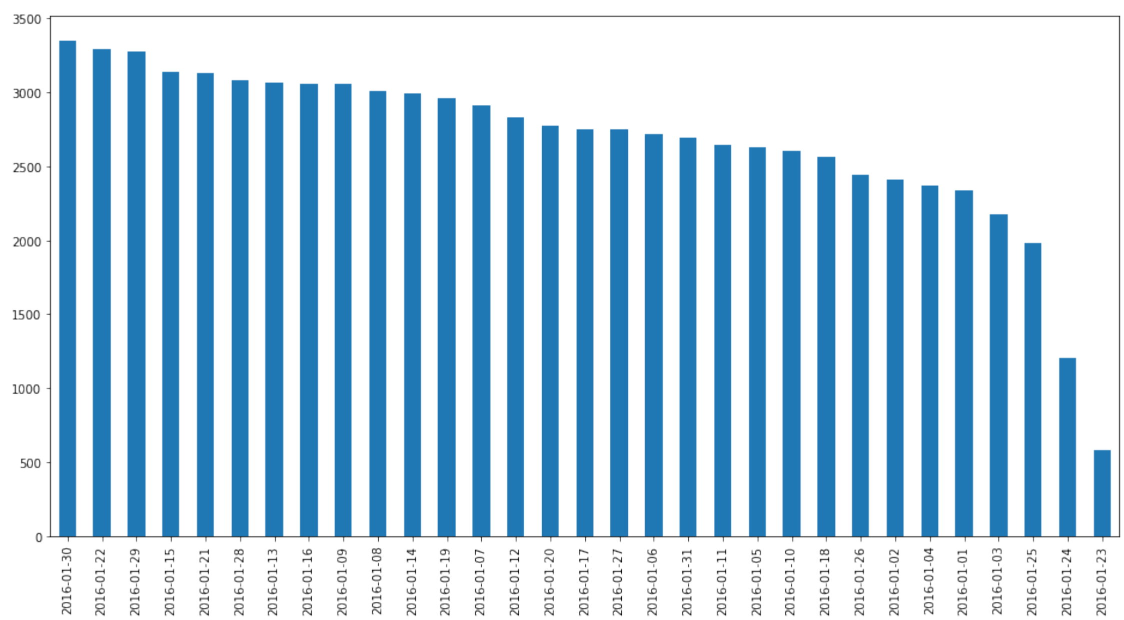
1.6.2 Question 2b

Create a data visualization that allows you to identify which dates were affected by the historic blizzard of January 2016. Make sure that the visualization type is appropriate for the visualized data.

As a hint, consider how taxi usage might change on a day with a blizzard. How could you visualize/plot this?

[15]:
```
plt.figure(figsize=(16, 8))
manhattan_taxi['date'].value_counts().plot.bar()
```

[15]: `<matplotlib.axes._subplots.AxesSubplot at 0x1a249d5650>`



Finally, we have generated 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. (No changes are needed; just run this cell.)

[16]:
```
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
```

You are welcome to perform more exploratory data analysis, but your work will not be scored. Here's a blank cell to use if you wish. In practice, further exploration would be warranted at this point, but the project is already pretty long.

[17]:
```
final_taxi.head()
```

[17]:
```
       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  2016-01-08 18:46:10  2016-01-08 18:54:00  -74.004494   40.706989
5  2016-01-17 19:21:16  2016-01-17 19:28:24  -73.978401   40.764992

   dropoff_lon  dropoff_lat  passengers  distance  duration        date
0   -74.015251    40.709808           2      3.99       981  2016-01-30
1   -73.975388    40.782200           1      2.03       320  2016-01-04
2   -73.985542    40.738510           1      0.70       299  2016-01-07
```

3	-74.010155	40.716751	5	0.97	470	2016-01-08
5	-73.981018	40.762100	1	0.95	428	2016-01-17

1.7 Part 3: Feature Engineering

In this part, you'll create a design matrix (i.e., feature matrix) for your linear regression model. This is analagous to the pipelines you've built already in class: you'll be adding features, removing labels, and scaling among other things.

You decide to predict trip duration from the following inputs: start location, end location, trip distance, time of day, and day of the week (*Monday, Tuesday, etc.*).

You will ensure that 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.

Because you are going to look at the data in detail in order to define features, it's best to split the data into training and test sets now, then only inspect the training set.

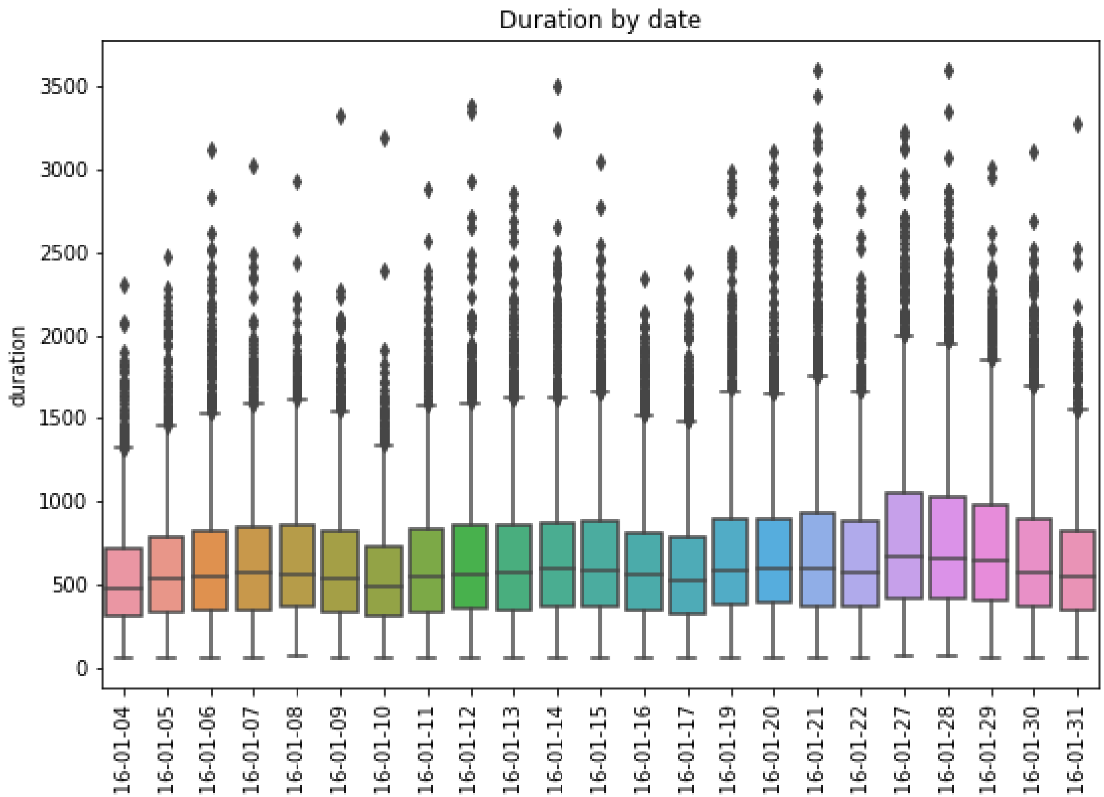
[18]:
```
import sklearn.model_selection

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)
```

1.7.1 Question 3a

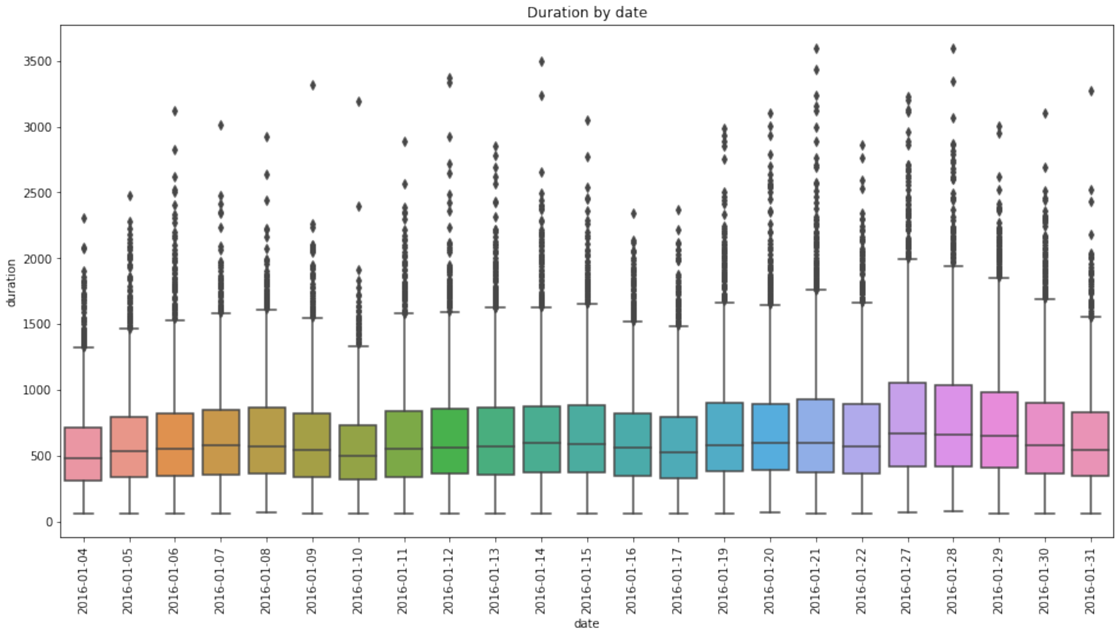
Create a box plot that compares the distributions of taxi trip durations for each day **using** `train` **only**. Individual dates shoud appear on the horizontal axis, and duration values should appear on the vertical axis. Your plot should look like the one below.

You can generate this type of plot using `sns.boxplot`



[19]:
```
plt.figure(figsize=(16, 8))
ax = sns.boxplot(x='date', y='duration', data=train.sort_values(['date']))
ax.set_title("Duration by date")
ax.set_xticklabels(ax.get_xticklabels(), rotation=90)
ax
```

[19]: <matplotlib.axes._subplots.AxesSubplot at 0x1a245c19d0>



1.7.2 Question 3b

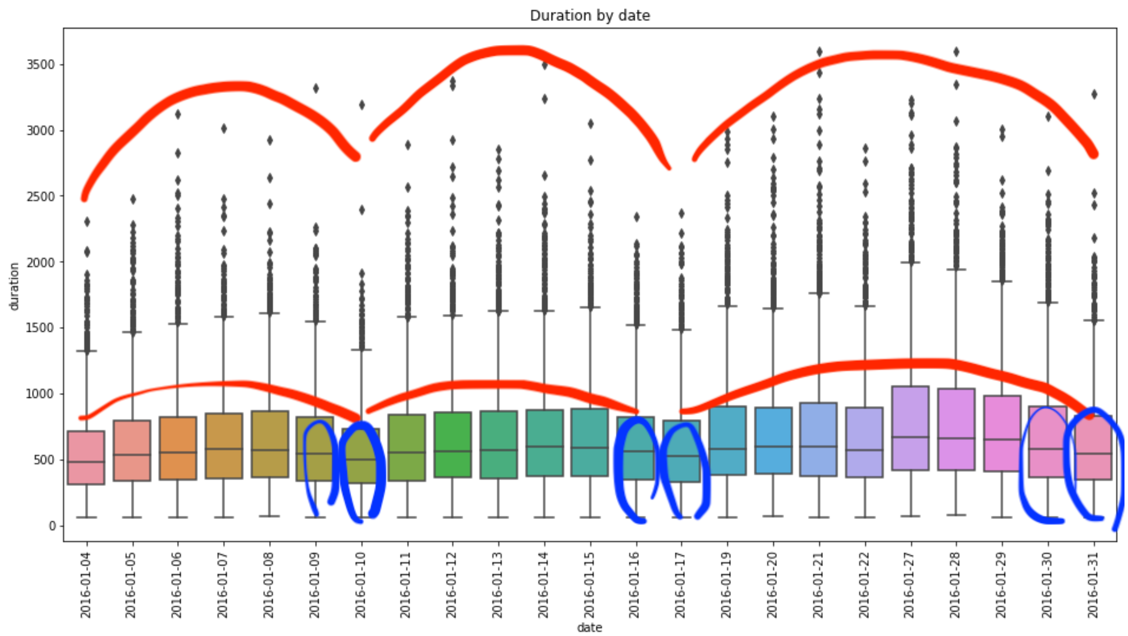
In one or two sentences, describe the assocation between the day of the week and the duration of a taxi trip. Your answer should be supported by your boxplot above.

Note: The end of Part 2 showed a calendar for these dates and their corresponding days of the week.

[20]:
```
print(re.sub(pat, '   ', calendar.month(2016, 1)))
```

```
    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
```

It looks like there are more taxi trips during the weekdays than on the weekends. Circled boxes are on the weekends. We can see that there are red "bumps" during the weekdays.



Below, the provided `augment` function adds various 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.

No changes are required; just run this cell.

[21]:
```
def speed(t):
    """Return a column of speeds in miles per hour."""
    return t['distance'] / t['duration'] * 60 * 60

def augment(t):
    """Augment 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,:] # An example row
```

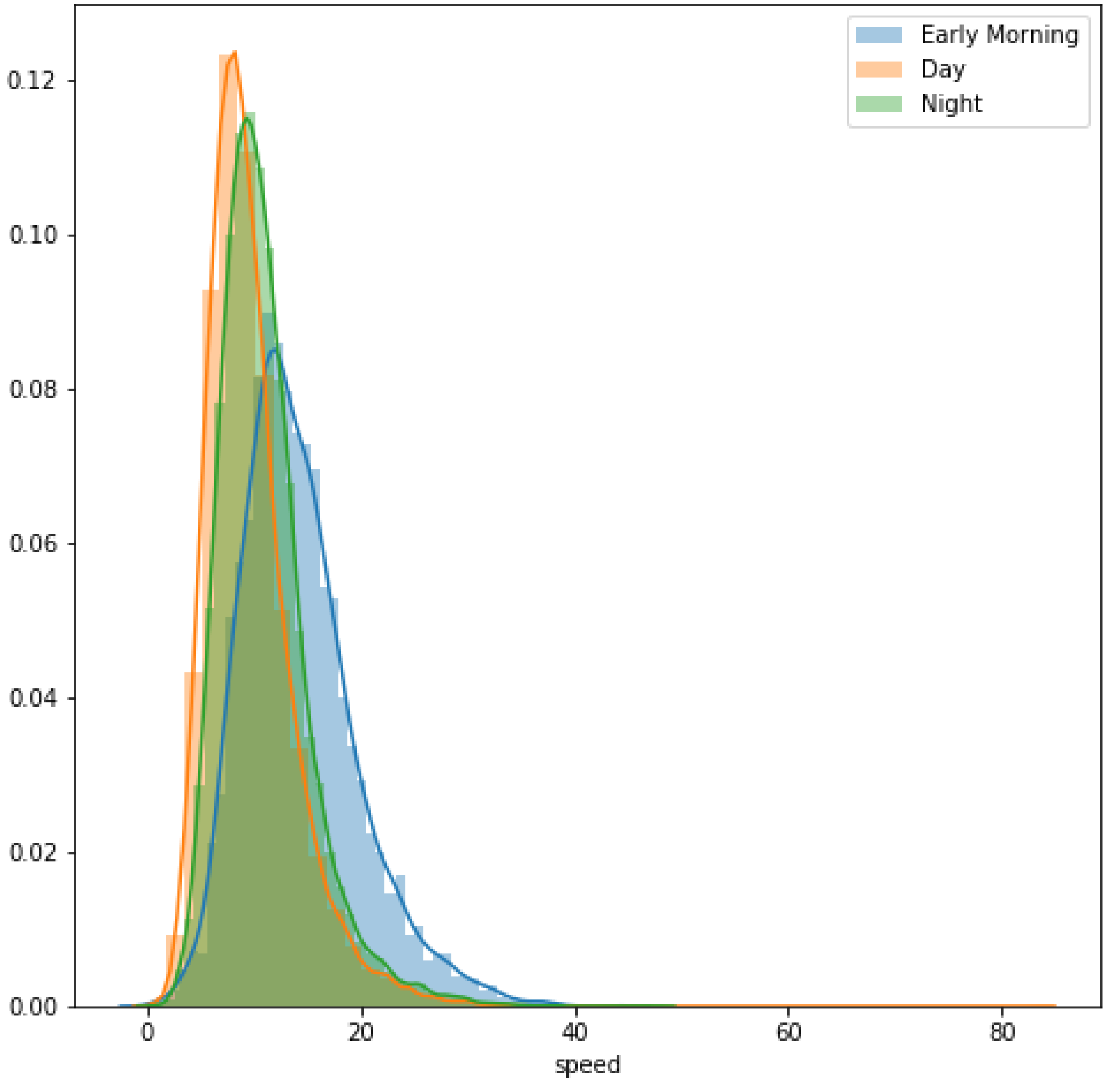
[21]:
```
pickup_datetime     2016-01-21 18:02:20
dropoff_datetime    2016-01-21 18:27:54
pickup_lon                     -73.9942
pickup_lat                       40.751
dropoff_lon                    -73.9637
dropoff_lat                     40.7711
passengers                            1
distance                           2.77
duration                           1534
date                         2016-01-21
hour                                 18
day                                   3
weekend                               0
period                                3
speed                           6.50065
Name: 14043, dtype: object
```

[22]:
```
train.head()
```

[22]:
```
           pickup_datetime     dropoff_datetime  pickup_lon  pickup_lat  \
14043  2016-01-21 18:02:20  2016-01-21 18:27:54  -73.994202   40.751019
9122   2016-01-29 06:18:36  2016-01-29 06:21:32  -73.990402   40.756344
9291   2016-01-04 20:34:21  2016-01-04 20:42:33  -74.006554   40.732922
76214  2016-01-09 12:12:58  2016-01-09 12:20:26  -73.992065   40.750313
46314  2016-01-13 10:57:45  2016-01-13 11:02:06  -73.959358   40.771824

       dropoff_lon  dropoff_lat  passengers  distance  duration        date  \
14043   -73.963692    40.771069           1      2.77      1534  2016-01-21
9122    -73.984161    40.761757           3      0.69       176  2016-01-29
9291    -74.001175    40.751366           1      1.60       492  2016-01-04
76214   -73.982803    40.755829           1      0.90       448  2016-01-09
46314   -73.964661    40.770443           1      0.40       261  2016-01-13

       hour  day  weekend  period      speed
14043    18    3        0       3   6.500652
9122      6    4        0       2  14.113636
9291     20    0        0       3  11.707317
76214    12    5        1       2   7.232143
46314    10    2        0       2   5.517241
```

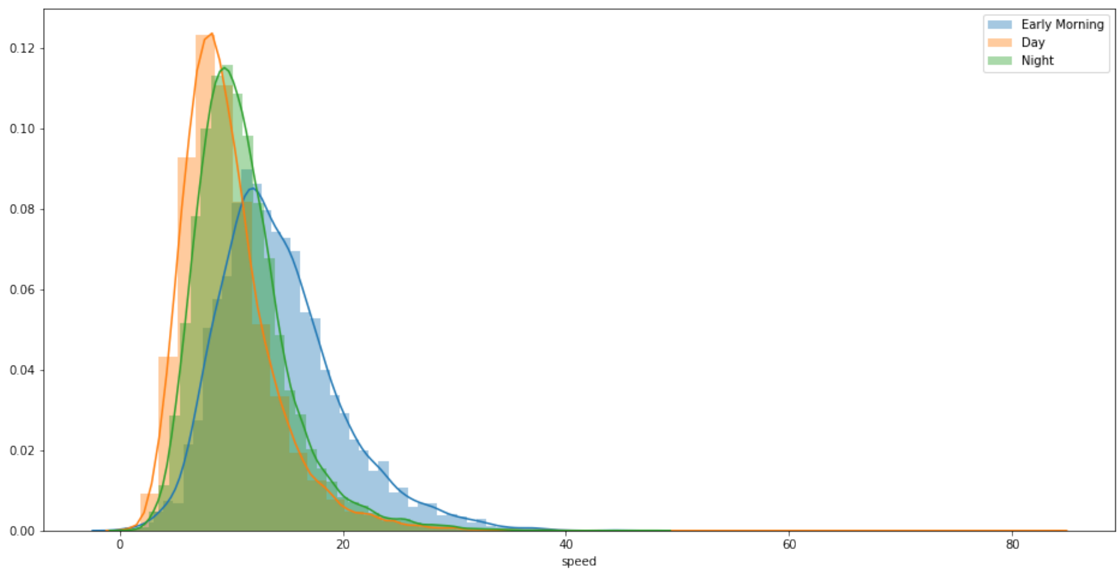
1.7.3 Question 3c

Use `sns.distplot` to 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). Your plot should look like this:



[23]:
```
plt.figure(figsize=(16, 8))
sns.distplot(train[train['period'] == 1]['speed'], label='Early Morning')
sns.distplot(train[train['period'] == 2]['speed'], label='Day')
sns.distplot(train[train['period'] == 3]['speed'], label='Night')
plt.legend()
```

[23]:
```
<matplotlib.legend.Legend at 0x1a27c4ac90>
```



It looks like the time of day is associated with the average speed of a taxi ride.

1.7.4 Question 3d

Manhattan can roughly be divided into Lower, Midtown, and Upper regions. Instead of studying a map, let's 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.

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.

Read the documentation of `pd.qcut`, which categorizes points in a distribution into equal-frequency bins.

You don't need to add any lines to this solution. Just fill in the assignment statements to complete the implementation.

Before implementing PCA, it is important to scale and shift your values. The line with `np.linalg.svd` will return your transformation matrix, among other things. You can then use this matrix to convert points in (lat, lon) space into (PC1, PC2) space.

Hint: If you are failing the tests, try visualizing your processed data to understand what your code might be doing wrong.

The provided tests ensure that you have answered the question correctly.

[24]:
```
# Find the first principle component
D = train[['pickup_lon', 'pickup_lat']]
pca_n = len(D)
pca_means = np.mean(D)
X = (D - pca_means) / np.sqrt(pca_n)
u, s, vt = np.linalg.svd(X, full_matrices=False)

def add_region(t):
    """Add a region column to t based on vt above."""
    D = t[['pickup_lon', 'pickup_lat']]
    assert D.shape[0] == t.shape[0], 'You 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.to_numpy().dot(vt)[:, 0]
    t.loc[:,'region'] = pd.qcut(first_pc, 3, labels=[0, 1, 2])

add_region(train)
add_region(test)
```

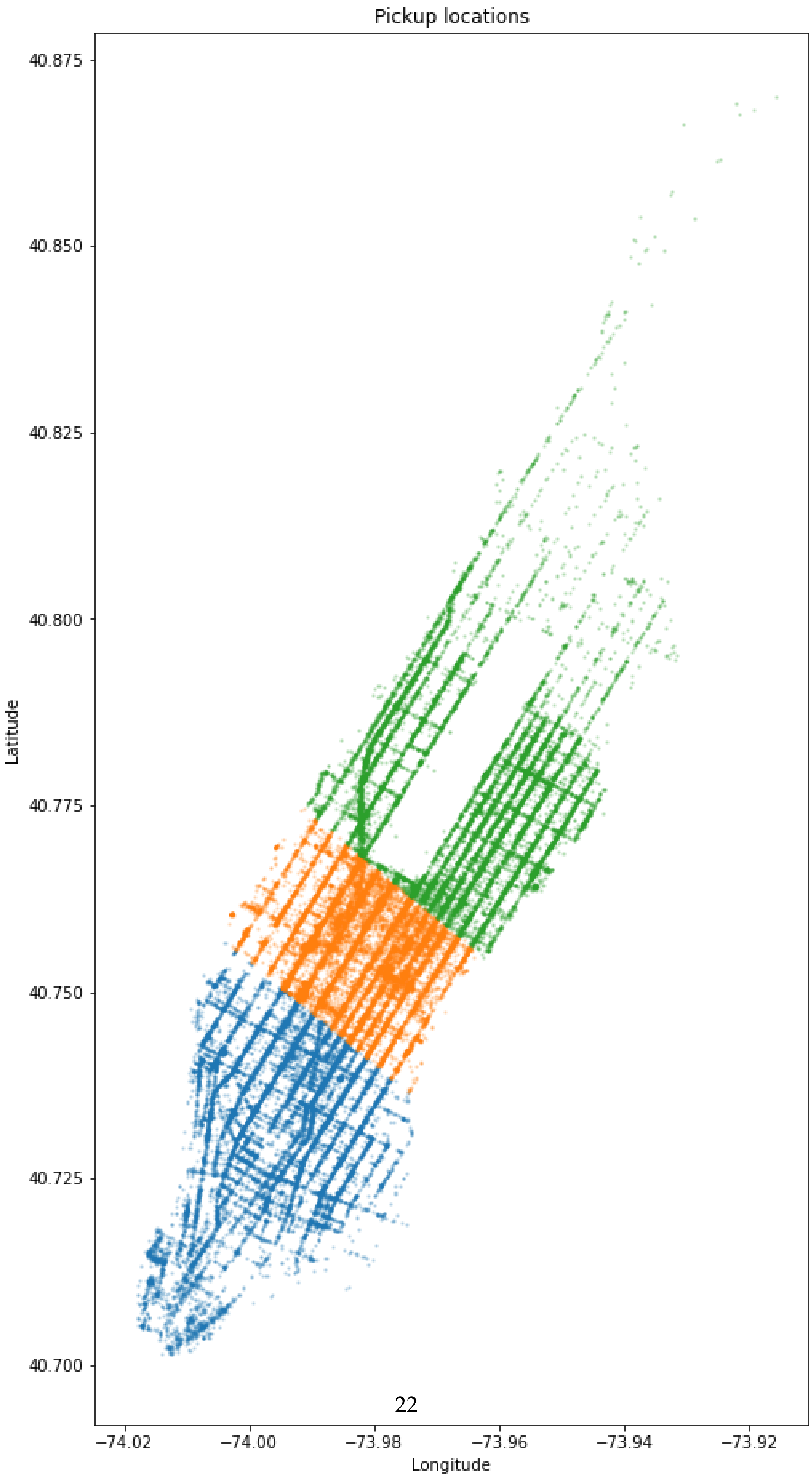
[25]:
```
grader.check("q3d")
```

[25]:

```
All tests passed!
```

Let's see how PCA divided the trips into three groups. These regions do roughly correspond to Lower Manhattan (below 14th street), Midtown Manhattan (between 14th and the park), and Upper Manhattan (bordering Central Park). No prior knowledge of New York geography was required!

[26]:
```
plt.figure(figsize=(8, 16))
for i in [0, 1, 2]:
    pickup_scatter(train[train['region'] == i])
```

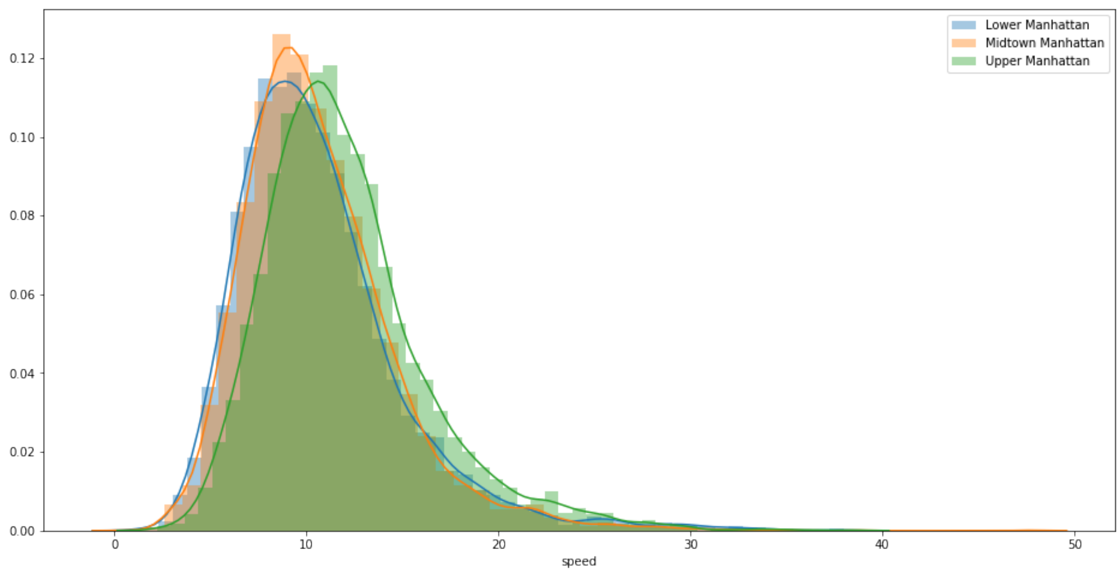

Pickup locations
40.875
40.850
40.825
40.800
40.775
40.750
40.725
40.700
Latitude
−74.02
−74.00
−73.98
−73.96
−73.94
−73.92
Longitude

1.7.5 Question 3e (ungraded)

Use `sns.distplot` to create an overlaid histogram comparing the distribution of speeds for nighttime taxi rides (6pm-12am) in the three different regions defined above. Does it appear that there is an association between region and average speed during the night?

[27]:
```
plt.figure(figsize=(16, 8))
sns.distplot(train[(train['period'] == 3) & (train['region'] == 0)]['speed'],␣
 ↪label='Lower Manhattan')
sns.distplot(train[(train['period'] == 3) & (train['region'] == 1)]['speed'],␣
 ↪label='Midtown Manhattan')
sns.distplot(train[(train['period'] == 3) & (train['region'] == 2)]['speed'],␣
 ↪label='Upper Manhattan')
plt.legend()
```

[27]: `<matplotlib.legend.Legend at 0x1a2740d9d0>`



Does it appear that there is an association between region and average speed during the night? – Yes!

Speed in Upper Manhattan > Speed in Midtown Manhattan > Speed in Lower Manhattan

Finally, we 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.

[28]:
```
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):
    """Create 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,:]
```

[28]:
```
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: 14043, dtype: float64
```

1.8 Part 4: Model Selection

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

Important: *Tests in this part do not confirm that you have answered correctly. Instead, they check that you're somewhat close in order to detect major errors. It is up to you to calculate the results correctly based on the question descriptions.*

1.8.1 Question 4a

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.

[29]:
```
def rmse(errors):
    """Return the root mean squared error."""
    return np.sqrt(np.mean(errors ** 2))

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

[29]:
```
399.1437572352677
```

[30]:
```
grader.check("q4a")
```

[30]:
```
    All tests passed!
```

1.8.2 Question 4b

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).

Terminology Note: Simple linear regression means that there is only one covariate. Multiple linear regression means that there is more than one. In either case, you can use the `LinearRegression` model from `sklearn` to fit the parameters to data.

[31]:
```
design_matrix(train).head()
```

[31]:
```
       pickup_lon  pickup_lat  dropoff_lon  dropoff_lat  distance  hour_1  \
14043   -0.805821   -0.171761     0.954062     0.624203  0.626326       0
9122    -0.570281    0.077978    -0.231391     0.234068 -0.826569       0
9291    -1.571566   -1.020443    -1.216691    -0.201297 -0.190927       0
76214   -0.673389   -0.204856    -0.152744    -0.014301 -0.679882       0
46314    1.354249    0.803938     0.897948     0.597992 -1.029136       0

       hour_2  hour_3  hour_4  hour_5  ...  hour_22  hour_23  day_1  day_2  \
14043       0       0       0       0  ...        0        0      0      0
9122        0       0       0       0  ...        0        0      0      0
9291        0       0       0       0  ...        0        0      0      0
76214       0       0       0       0  ...        0        0      0      0
46314       0       0       0       0  ...        0        0      0      1

       day_3  day_4  day_5  day_6  region_1  region_2
14043      1      0      0      0         1         0
9122       0      1      0      0         1         0
9291       0      0      0      0         0         0
76214      0      0      1      0         1         0
46314      0      0      0      0         0         1

[5 rows x 36 columns]
```

[32]:
```
from sklearn.linear_model import LinearRegression

model = LinearRegression()

X_train = design_matrix(train)['distance'].values.reshape(-1, 1)
y_train = train['duration']
model.fit(X_train, y_train)

X_test = design_matrix(test)['distance'].values.reshape(-1, 1)
y_test = test['duration']

simple_rmse = rmse(y_test - model.predict(X_test))
simple_rmse
```

[32]:
```
276.78411050003365
```

[33]:
```
grader.check("q4b")
```

[33]:

```
All tests passed!
```

1.8.3 Question 4c

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.

The provided tests check that you have answered the question correctly and that your `design_matrix` function is working as intended.

[34]:
```
model = LinearRegression()

X_train = design_matrix(train)
y_train = train['duration']
model.fit(X_train, y_train)

X_test = design_matrix(test)
y_test = test['duration']

linear_rmse = rmse(y_test - model.predict(X_test))
linear_rmse
```

[34]: 255.19146631882776

[35]:
```
grader.check("q4c")
```

[35]:

```
All tests passed!
```

1.8.4 Question 4d

For each possible value of `period`, fit an unregularized linear regression model to the subset of the training set in that `period`. 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, fit to the training set and use the `design_matrix` function for features.

[36]:
```
model = LinearRegression()
errors = []

for v in np.unique(train['period']):
    v_train = train[train['period'] == v]
    v_test = test[test['period'] == v]
```

```
    X_train = design_matrix(v_train)
    y_train = v_train['duration']
    model.fit(X_train, y_train)

    X_test = design_matrix(v_test)
    y_test = v_test['duration']

    error = y_test - model.predict(X_test)
    # concatenate array of errors into single array
    errors = np.concatenate((errors, error), axis=None)

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

[36]: 246.62868831165173

[37]:
```
grader.check("q4d")
```

[37]:

```
All tests passed!
```

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.

1.8.5 Question 4e

In one or two sentences, explain how the `period` regression model above could possibly outperform linear regression when the design matrix for linear regression already includes one feature for each possible hour, which can be combined linearly to determine the `period` value.

Yes, simply adding a feature that is linearly dependent to other features don't help. However, we are doing something different here. Instead of having only one model, we are creating different models for each period. Thus, it produces better result because each model is tailored toward corresponding period.

1.8.6 Question 4f

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.

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.

Hint: Speed is in miles per hour, but duration is measured in seconds. You'll need the fact that there are 60 * 60 = 3,600 seconds in an hour.

[38]:
```
model = LinearRegression()

X_train = design_matrix(train)
y_train = train['speed']
model.fit(X_train, y_train)

X_test = design_matrix(test)
y_test = test['speed']
speed_rmse = rmse(3600 * test['distance'] / y_test - 3600 * test['distance'] /
 ↪model.predict(X_test))
speed_rmse
```

[38]:
```
243.01798368514974
```

[39]:
```
grader.check("q4f")
```

[39]:
```
All tests passed!
```

Optional: Explain why predicting speed leads to a more accurate regression model than predicting duration directly. You don't need to write this down.

1.8.7 Question 4g

Finally, complete the function `tree_regression_errors` (and helper function `speed_error`) that combines the ideas from the two previous models and generalizes to multiple categorical variables.

The `tree_regression_errors` should: - 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 (`outcome` will be `'duration'` or `'speed'`); - Use the specified `error_fn` (either `duration_error` or `speed_error`) to compute the error in predicted duration using the predicted outcome; - Aggregate those errors over the whole test set and return them.

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

If you're stuck, try putting print statements in the skeleton code to see what it's doing.

[40]:
```
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"""
    return (observations['distance'] / predictions * 3600) -␣
↪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
        for v, c in zip(vs, choices):
            v_train = v_train[v_train[c] == v]
            v_test = v_test[v_test[c] == v]

        X_train = design_matrix(v_train)
        y_train = v_train[outcome]
        model = LinearRegression()
        model.fit(X_train, y_train)

        X_test = design_matrix(v_test)

        error = error_fn(model.predict(X_test), v_test)
        errors.extend(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: 240.33952192703526
Speed: 226.90793945018308
```

[41]:
```
grader.check("q4g")
```

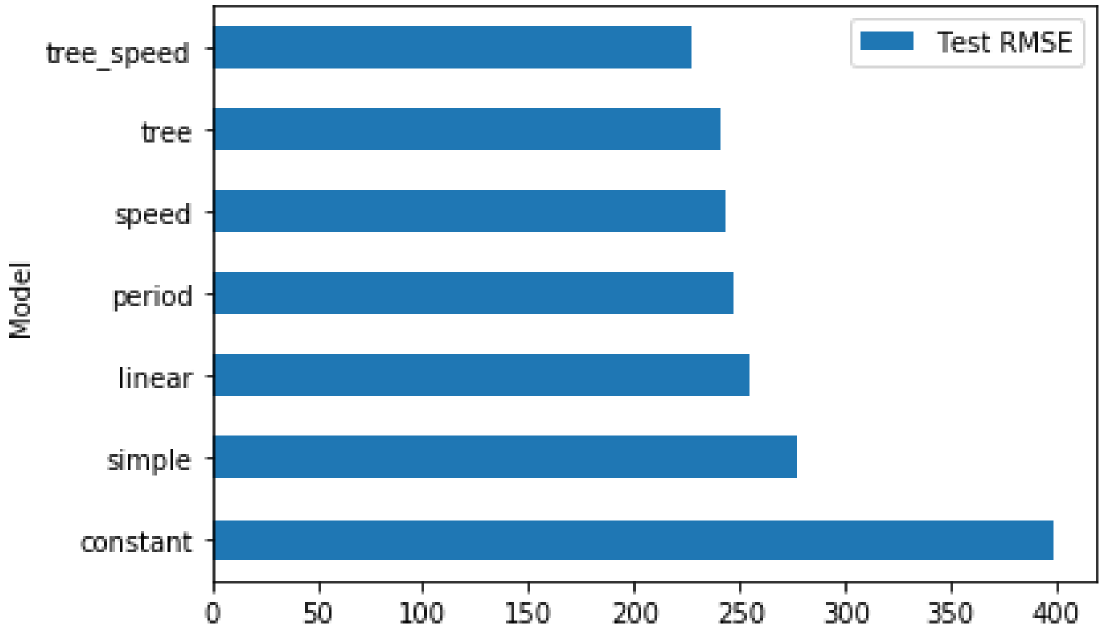
[41]:

```
All tests passed!
```

Here's a summary of your results:

[42]:
```
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');
```



1.9 Part 5: Building on your own

In this part you'll build a regression model of your own design, with the goal of achieving even higher performance than you've seen already. You will be graded on your performance relative to others in the class, with higher performance (lower RMSE) receiving more points.

1.9.1 Question 5a

In the below cell (feel free to add your own additional cells), train a regression model of your choice on the same train dataset split used above. The model can incorporate anything you've learned from the class so far.

The model you train will be used for questions 5b and 5c

[54]:
```
# https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.
↪MLPRegressor.html
# this will take a while...
from sklearn.neural_network import MLPRegressor
from IPython.display import clear_output

X_train = design_matrix(train)
y_train = train['speed']

X_test = design_matrix(test)
```

```
y_test = test['speed']

model = MLPRegressor(max_iter=2000, early_stopping=True)
model.fit(X_train, y_train)
```

[54]:
```
MLPRegressor(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
             beta_2=0.999, early_stopping=True, epsilon=1e-08,
             hidden_layer_sizes=(100,), learning_rate='constant',
             learning_rate_init=0.001, max_iter=2000, momentum=0.9,
             n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
             random_state=None, shuffle=True, solver='adam', tol=0.0001,
             validation_fraction=0.1, verbose=False, warm_start=False)
```

1.9.2 Question 5b

Print a summary of your model's performance. You **must** include the RMSE on the train and test sets. Do not hardcode any values or you won't receive credit.

Don't include any long lines or we won't be able to grade your response.

[55]:
```
speed_rmse_train = rmse(3600 * train['distance'] / y_train - 3600 *␣
 ↪train['distance'] / model.predict(X_train))
speed_rmse_test = rmse(3600 * test['distance'] / y_test - 3600 *␣
 ↪test['distance'] / model.predict(X_test))
print("rmse_train: ", speed_rmse_train)
print("rmse_test: ", speed_rmse_test)
```

```
rmse_train:  192.4421060754966
rmse_test:  198.74159724123942
```

1.9.3 Question 5c

Describe why you selected the model you did and what you did to try and improve performance over the models in section 4.

Responses should be at most a few sentences

I selected neural network because it can perform non linear regression, which is more powerful than linear regression.

In order to improve performance over the models in section 4, I tried to avoid overfitting by doing early stopping.

Congratulations! You've carried out the entire data science lifecycle for a challenging regression problem.

In Part 1 on data selection, you 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, you 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, you 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, you 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 you to predict duration more accurately by first predicting speed.

In Part 5, you made your own model using techniques you've learned throughout the course.

Hopefully, it is apparent that all of these steps are required to reach a reliable conclusion about what inputs and model structure are helpful in predicting the duration of a taxi ride in Manhattan.

1.10 Future Work

Here are some questions to ponder:

- The regression model would have been more accurate if we had used the date itself as a feature instead of just the day of the week. Why didn't we do that?
- Does collecting this information about every taxi ride introduce a privacy risk? The original data also included the total fare; how could someone use this information combined with an individual's credit card records to determine their location?
- Why did we treat `hour` as a categorical variable instead of a quantitative variable? Would a similar treatment be beneficial for latitude and longitude?
- Why are Google Maps estimates of ride time much more accurate than our estimates?

Here are some possible extensions to the project:

- An alternative to throwing out atypical days is to condition on a feature that makes them atypical, such as the weather or holiday calendar. How would you do that?
- Training a different linear regression model for every possible combination of categorical variables can overfit. How would you select which variables to include in a decision tree instead of just using them all?
- Your models use the observed distance as an input, but the distance is only observed after the ride is over. How could you estimate the distance from the pick-up and drop-off locations?
- How would you incorporate traffic data into the model?

[47]:
```
# Save your notebook first, then run this cell to generate a PDF.
# Note, the download link will likely not work.
# Find the pdf in the same directory as your proj3.ipynb
grader.export("proj3.ipynb", filtering=False)
```

[ ]:
```
```