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

Project 3: Predicting Taxi Ride Duration

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: Andrew Grove (304785991)

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
5b	7
5c	3
Total	43

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:

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

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
```

The Data

Attributes of all [yellow taxi (https://en.wikipedia.org/wiki/Taxicabs_of_New_York_City)](https://en.wikipedia.org/wiki/Taxicabs_of_New_York_City) trips in January 2016 are published by the [NYC Taxi and Limosine Commission (https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page)](https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page).

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.

Part 1: Data Selection and Cleaning

In this part, you will limit the data to trips that began and ended on Manhattan Island (map (https://www.google.com/maps/place/Manhattan,+New+York,+NY/@40.7590402,-74.0394431,12z/data=!3m1!4b 73.9712488)).

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.

In [3]:

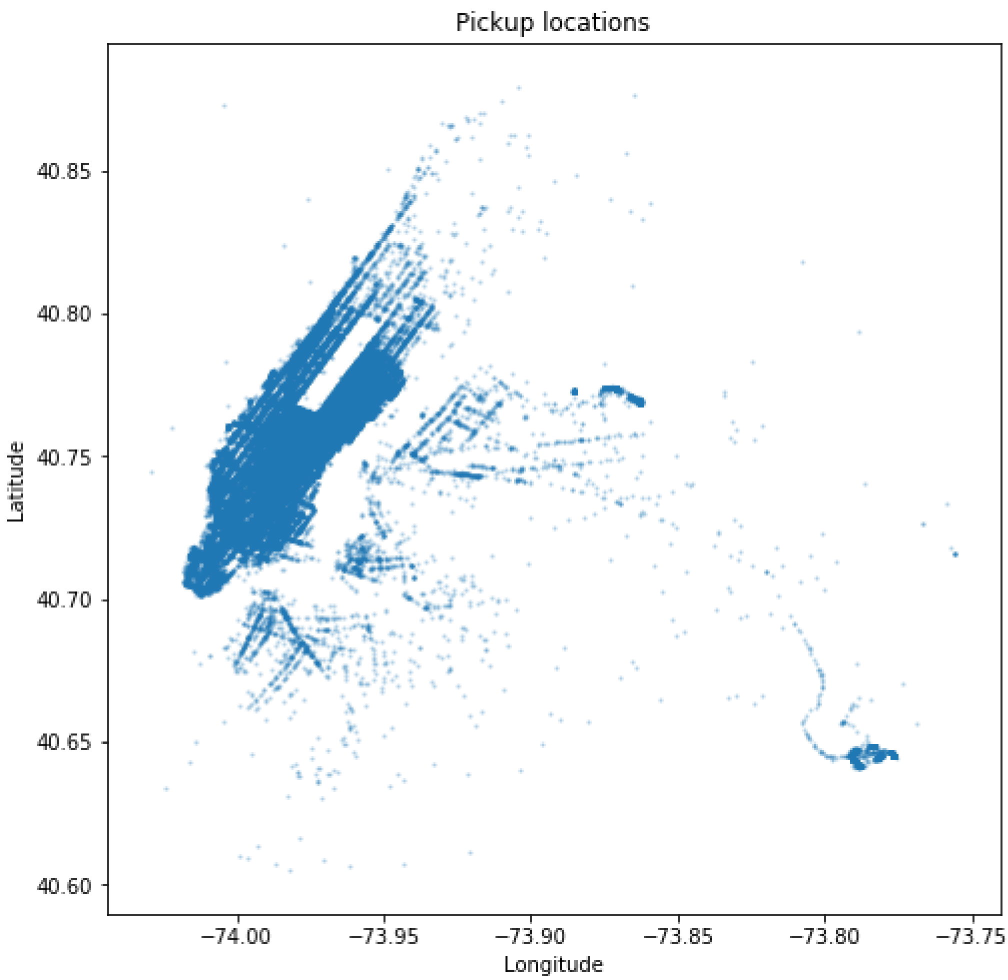
```python
import sqlite3

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()
```

Out[3]:

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

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.

```
In [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.

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.

In [5]:
```
clean_taxi = all_taxi[all_taxi.passengers >= 1]
clean_taxi = clean_taxi[clean_taxi.distance > 0]
clean_taxi = clean_taxi[clean_taxi.duration >=60]
clean_taxi = clean_taxi[clean_taxi.duration <=3600]
clean_taxi = clean_taxi[(clean_taxi.distance*3600)/(clean_taxi.duration)
<=100]
```

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

Out[6]: All tests passed!

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 (https://www.google.com/maps/place/Manhattan,+New+York,+NY/@40.7590402,-74.0394431,12z/data=!3m1!4b 73.9712488)](https://www.google.com/maps/place/Manhattan,+New+York,+NY/@40.7590402,-74.0394431,12z/data=!3m1!4b).

The vertices of this polygon are defined in `manhattan.csv` as (latitude, longitude) pairs, which are [published here (https://gist.github.com/baygross/5430626)](https://gist.github.com/baygross/5430626).

An efficient way to test if a point is contained within a polygon is [described on this page (http://alienryderflex.com/polygon/)](http://alienryderflex.com/polygon/). 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.

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

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

polyX = polygon["lon"]
polyY = polygon["lat"]

multiple = np.empty([polyX.size])
constant = np.empty([polyX.size])

def precalc_values():
    i = polyX.size - 1
    j = polyX.size - 1

    for i in range(polyX.size):
        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 in_manhattan(x, y):
    """Whether a longitude-latitude (x, y) pair is in the Manhattan polyg
on."""
    oddNodes = False;
    current = polyY[polyX.size - 1] > y

    for i in range(polyX.size):
        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_ta
xi.

precalc_values()
manhattan_taxi = clean_taxi[clean_taxi.apply(lambda x : in_manhattan(x["p
ickup_lon"], x["pickup_lat"]), axis=1)]
manhattan_taxi = manhattan_taxi[manhattan_taxi.apply(lambda x : in_manhat
tan(x["dropoff_lon"], x["dropoff_lat"]), axis=1)]

manhattan_taxi.head()
```

Out[7]:

	pickup_datetime	dropoff_datetime	pickup_lon	pickup_lat	dropoff_lon	dropoff_lat	passengers
0	2016-01-30 22:47:32	2016-01-30 23:03:53	-73.988251	40.743542	-74.015251	40.709808	1
1	2016-01-04 04:30:48	2016-01-04 04:36:08	-73.995888	40.760010	-73.975388	40.782200	1
2	2016-01-07 21:52:24	2016-01-07 21:57:23	-73.990440	40.730469	-73.985542	40.738510	1
4	2016-01-08 18:46:10	2016-01-08 18:54:00	-74.004494	40.706989	-74.010155	40.716751	5
5	2016-01-02 12:39:57	2016-01-02 12:53:29	-73.958214	40.760525	-73.983360	40.760406	1

In [8]:

```
grader.check("q1c")
```

Out[8]: All tests passed!

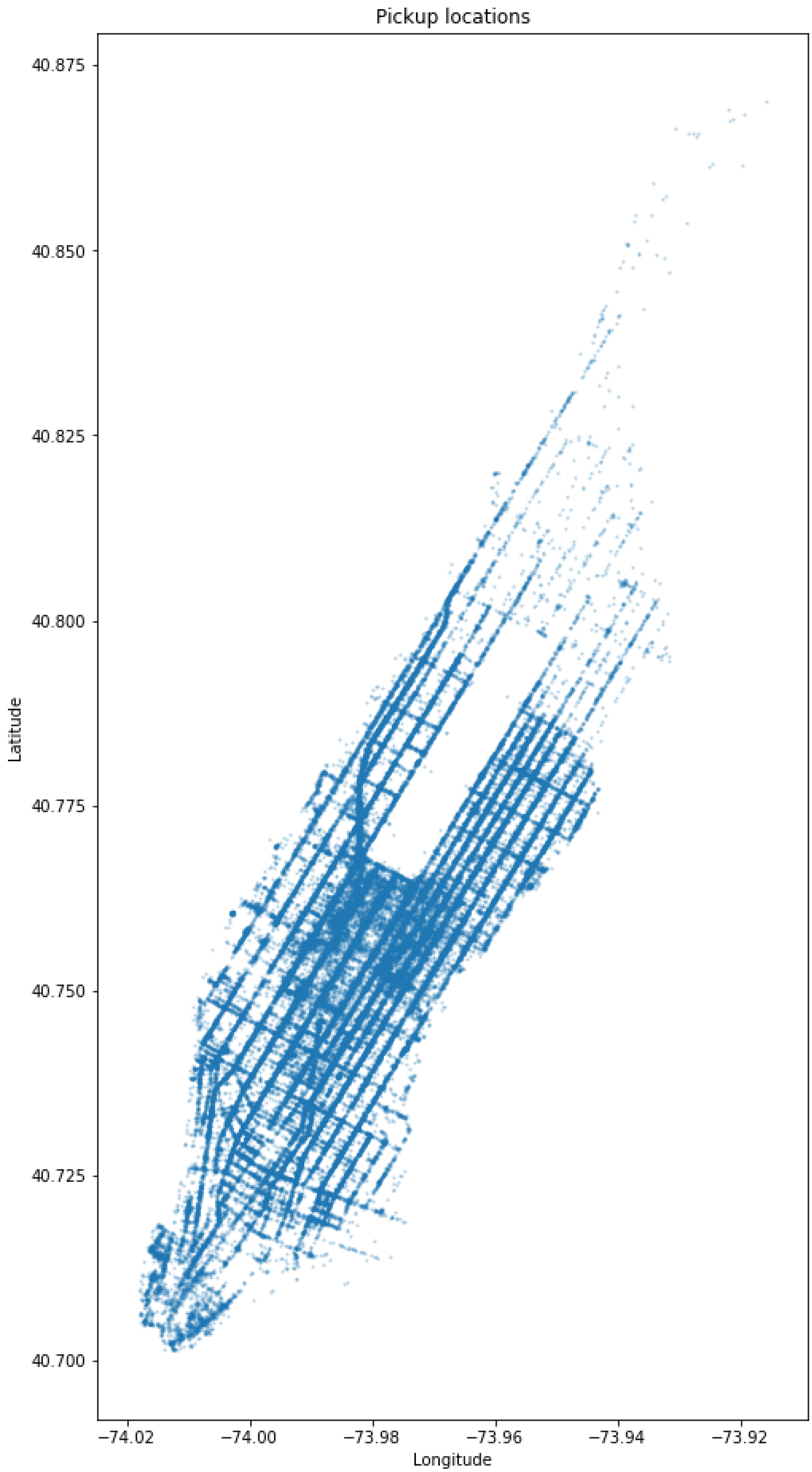
If you are unable to solve the problem above, have trouble with the tests, or want to work on the rest of the project before solving it, run the following cell to load the cleaned Manhattan data directly. (Note that you may not solve the previous problem just by loading this data file; you have to actually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

In [9]:

```
manhattan_taxi = pd.read_csv('manhattan_taxi.csv')
```

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

In [10]:
```
plt.figure(figsize=(8, 16))
pickup_scatter(manhattan_taxi)
```


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

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.

In [11]:
```
original = all_taxi.size
difference = all_taxi.size - clean_taxi.size
percent = round(difference / original * 100, 1)
manhattan = manhattan_taxi.size

print(f"Of the original {original}, trips, {difference} trips ({percent}
%) were removed through data cleaning and \
then {manhattan} trips within Manhattan were selected for further analysi
s")
```

```
Of the original 879228, trips, 11223 trips (1.3%) were removed through
data cleaning and then 745200 trips within Manhattan were selected for
further analysis
```

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 (https://en.wikipedia.org/wiki/January_2016_United_States_blizzard)](https://en.wikipedia.org/wiki/January_2016_United_States_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.

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 (https://docs.python.org/3/library/datetime.html#date-objects)](https://docs.python.org/3/library/datetime.html#date-objects)).

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

In [12]:
```
import calendar
import re
from datetime import datetime
```

In [13]:
```
manhattan_taxi["date"] = manhattan_taxi["pickup_datetime"].apply(lambda x
 : datetime.strptime(x[0:10], '%Y-%m-%d'))
manhattan_taxi.head()
manhattan_taxi.shape
```

Out[13]:
```
(82800, 10)
```

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

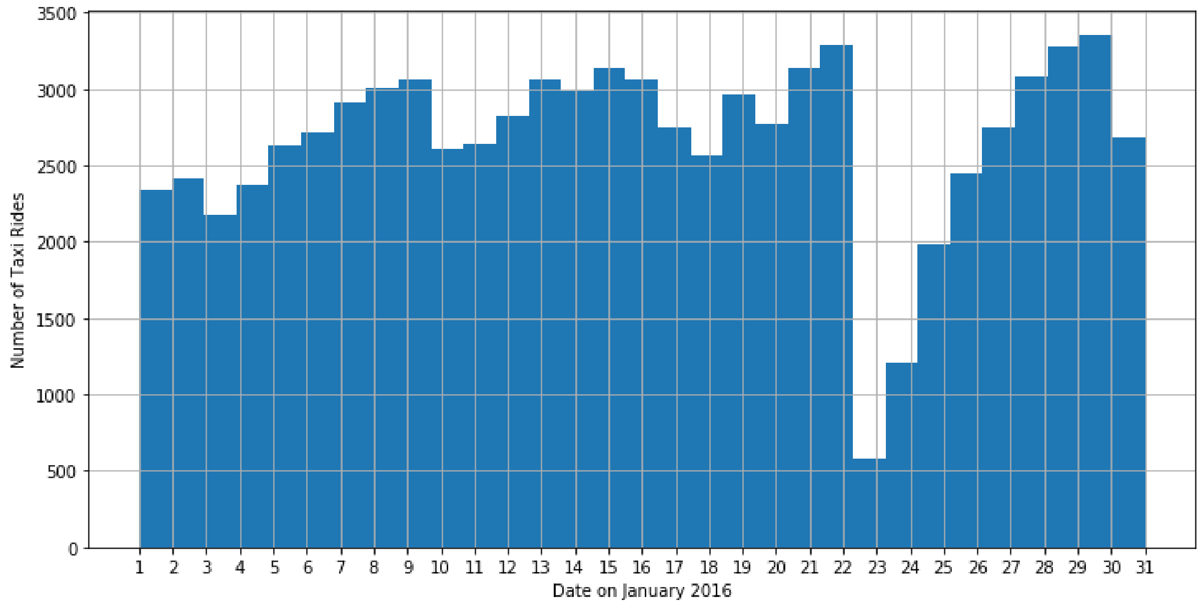
Out[14]: All tests passed!

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?

In [15]:
```
start_date = datetime.strptime('2016-01-01', '%Y-%m-%d')
end_date = datetime.strptime('2016-01-31', '%Y-%m-%d')

jan2016 = manhattan_taxi.date[manhattan_taxi.date >= start_date]
jan2016 = jan2016[jan2016 <= end_date]

counts = jan2016.value_counts().sort_index()
days = jan2016.apply(lambda x : x.day)

days.hist(bins=31, figsize=(12,6))
plt.xlabel('Date on January 2016')
plt.ylabel('Number of Taxi Rides')
plt.xticks(range(1,32,1))
plt.show()

plt.figure(figsize=(12,6))
plt.plot(range(1,32), counts)
plt.grid()
plt.xlabel('Date on January 2016')
plt.ylabel('Number of Taxi Rides')
plt.title('Number of Taxi Rides on Jan 2016')
plt.xticks(range(1, 32, 1))
plt.show()
```



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.)

```
In [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 atypi
         cal]
         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.

```
In [17]: # Optional: More EDA here
```

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.

```
In [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)
```

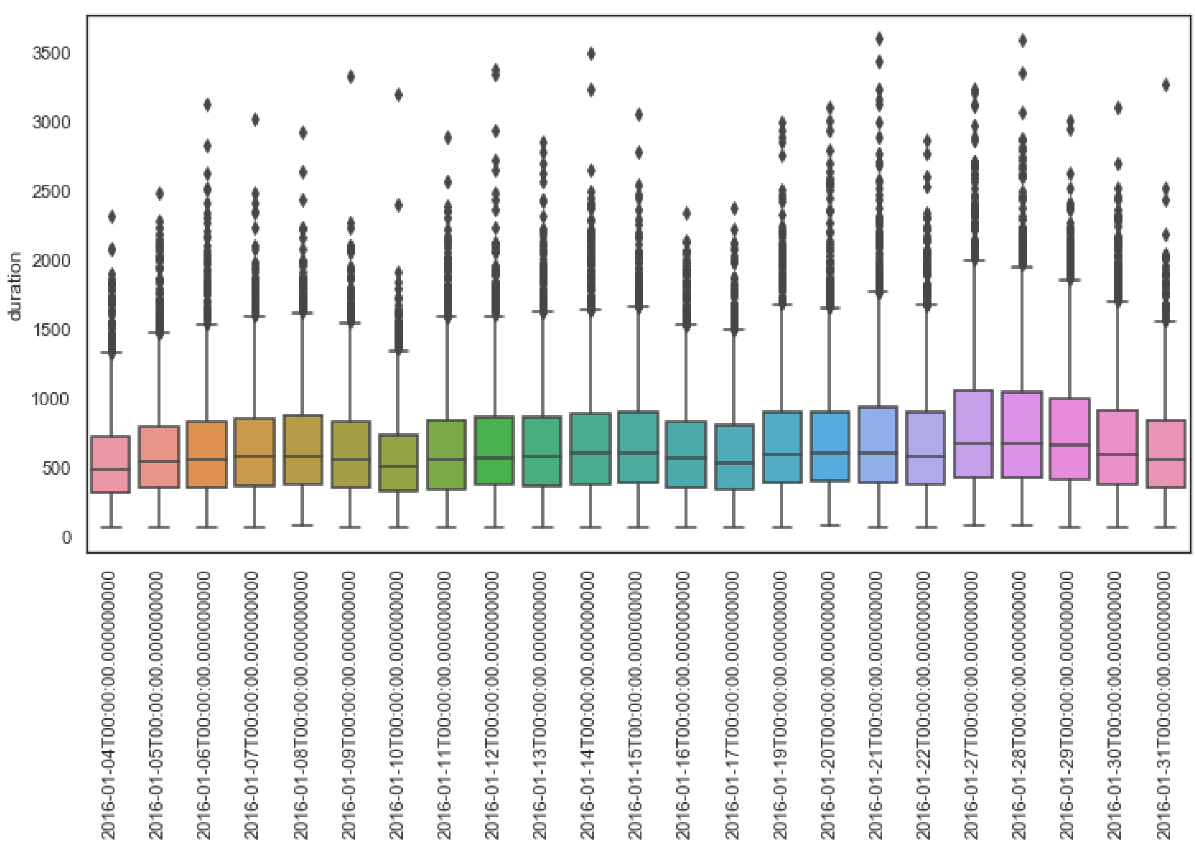
Question 3a

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`

```
In [19]: sns.set(rc={'figure.figsize':(12,6)})
         sns.set(style="white")
         date_train = train.sort_values(by='date')
         sns.boxplot(x="date", y="duration", data=date_train)
         plt.xticks(rotation=90)
         plt.xlabel('')
         plt.plot()
```

```
Out[19]: []
```



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.

From the boxplot, it looks like the duration of taxi trips during weekdays are higher than that of the weekends. This makes sense as during the weekdays, everyone is on the road at the same time (morning 8-9 am, evening 5-7 pm) which causes congestion and increases the taxi trip duration.

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.

In [20]:
```python
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
```

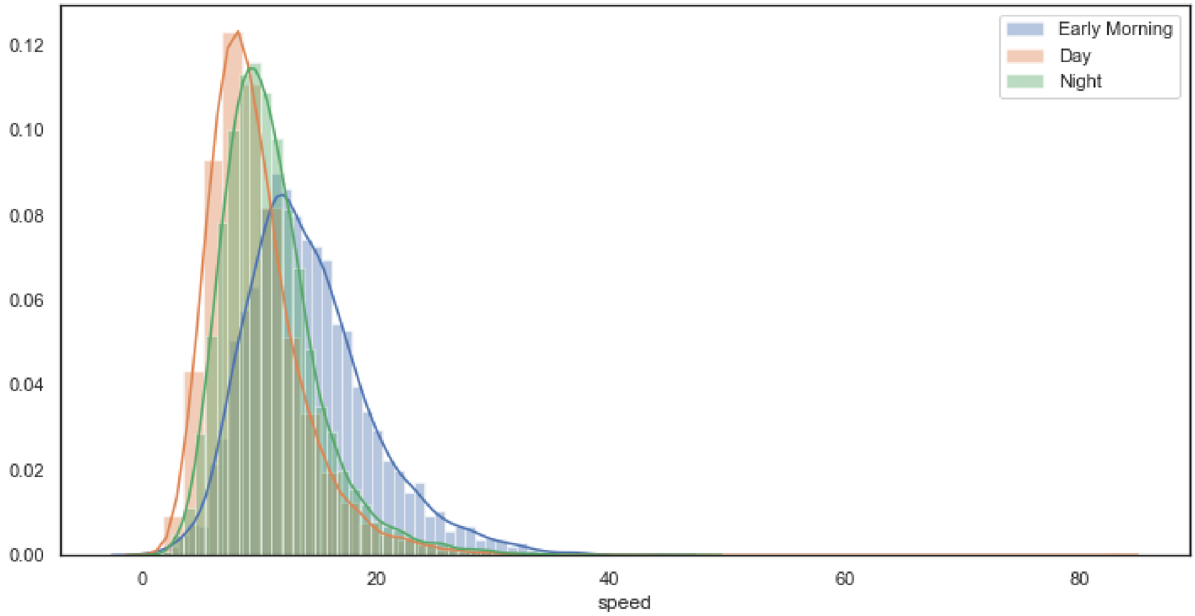
Out[20]:
```
pickup_datetime     2016-01-21 18:02:20
dropoff_datetime    2016-01-21 18:27:54
pickup_lon                     -73.9942
pickup_lat                       40.751
dropoff_lon                    -73.9637
dropoff_lat                     40.7711
passengers                            1
distance                           2.77
duration                           1534
date                2016-01-21 00:00:00
hour                                 18
day                                   3
weekend                               0
period                                3
speed                           6.50065
Name: 14043, dtype: object
```

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:

In [21]:
```
morning = train[train.period == 1]['speed']
day = train[train.period == 2]['speed']
night = train[train.period == 3]['speed']

sns.distplot(morning, label='Early Morning')
sns.distplot(day, label='Day')
sns.distplot(night, label='Night')
plt.legend()
```

Out[21]:
```
<matplotlib.legend.Legend at 0x128058c10>
```



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

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 (https://en.wikipedia.org/wiki/Principal_component_analysis)](https://en.wikipedia.org/wiki/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` (https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.qcut.html)](https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.qcut.html), 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.

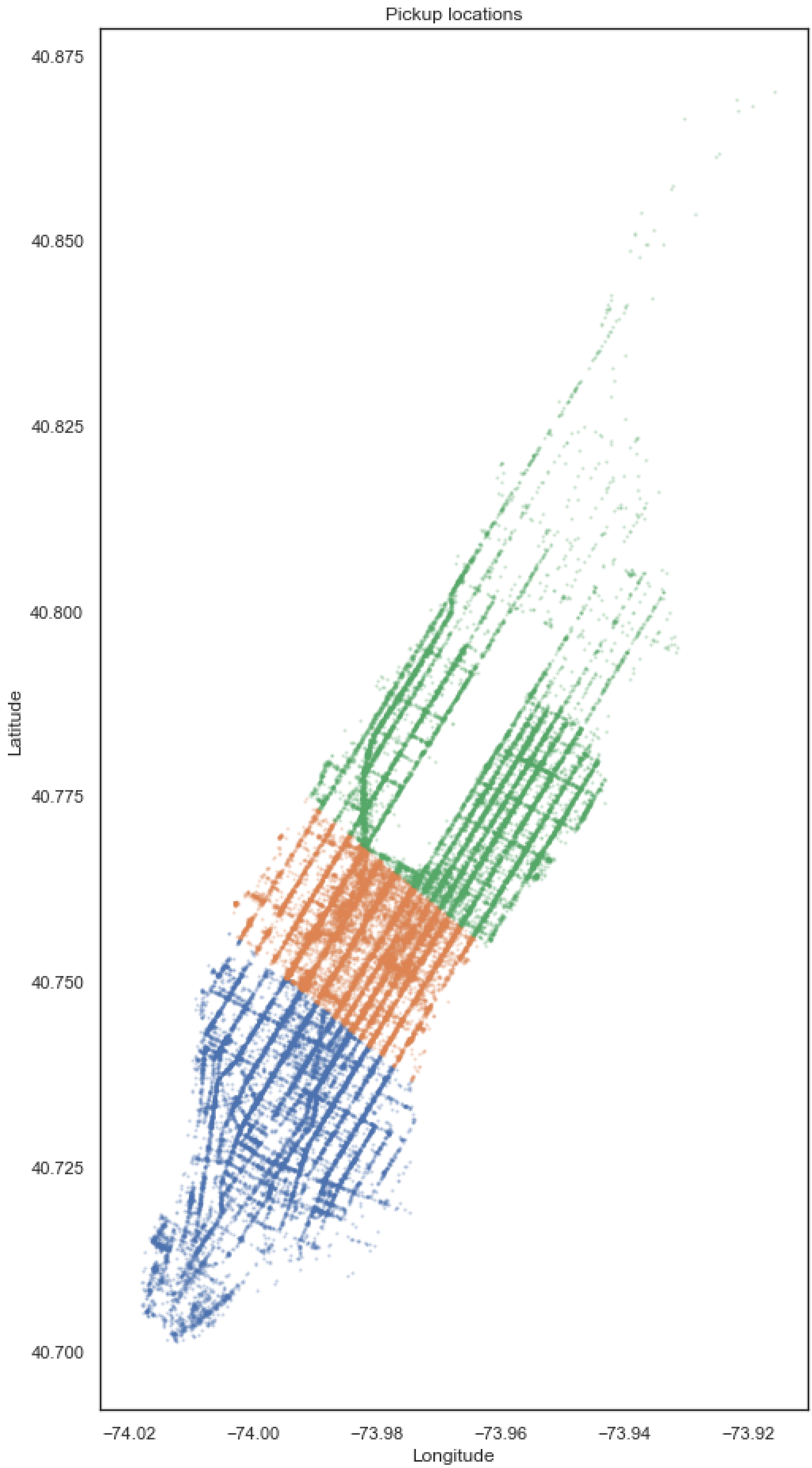
The provided tests ensure that you have answered the question correctly.

In [22]:
```
# Find the first principle component
D = pd.concat([train["pickup_lat"], train["pickup_lon"]], axis=1)
pca_n = D["pickup_lon"].size
pca_means = D.mean(axis=0)
X = (D - pca_means) / np.sqrt(pca_n)
u, s, vt = np.linalg.svd(X, full_matrices=False)
u.shape, s.shape, vt.shape

def add_region(t):
    """Add a region column to t based on vt above."""
    D = pd.concat([t["pickup_lat"], t["pickup_lon"]], axis=1)
    assert D.shape[0] == t.shape[0], 'You set D using the incorrect tabl
e'
    # Always use the same data transformation used to compute vt
    X = (D - pca_means) / np.sqrt(pca_n)
    first_pc = (X.values @ vt.T)[:,0]
    t.loc[:,'region'] = pd.qcut(first_pc, 3, labels=[0, 1, 2])


add_region(train)
add_region(test)
```

In [23]:
```
grader.check("q3d")
```
Out[23]: 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!

In [24]:
```
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

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?

In [25]:
```
...
```

Out[25]:
```
Ellipsis
```

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.

In [26]:
```
from sklearn.preprocessing import StandardScaler

num_vars = ['pickup_lon', 'pickup_lat', 'dropoff_lon', 'dropoff_lat', 'di
stance']
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 uni
ts
    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,:]
```

Out[26]:
```
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
```

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.*

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.

```
In [27]: def rmse(errors):
             """Return the root mean squared error."""
             return np.sqrt(np.mean(errors ** 2))

         constant_rmse = rmse(test["duration"] - test.mean(axis=0)["duration"])
         constant_rmse
```

```
Out[27]: 399.03723106267665
```

```
In [28]: grader.check("q4a")
```

Out[28]: All tests passed!

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.

```
In [29]: from sklearn.linear_model import LinearRegression

         model = LinearRegression()
         np_train = np.array(train["distance"]).reshape(-1,1)
         reg = model.fit(np_train, train["duration"])
         simple_rmse = rmse(test["duration"] - reg.predict(np.array(test["distanc
         e"]).reshape(-1,1)))
         simple_rmse
```

```
Out[29]: 276.7841105000342
```

```
In [30]: grader.check("q4b")
```

Out[30]: All tests passed!

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.

In [31]:
```
model = LinearRegression()
design_matrix_train = design_matrix(train.drop("duration", axis=1))
reg = model.fit(design_matrix_train, train["duration"])
linear_rmse = rmse(test["duration"] - reg.predict(design_matrix(test.drop
("duration", axis=1))))
linear_rmse
```

Out[31]: 255.19146631882754

In [32]:
```
grader.check("q4c")
```

Out[32]: All tests passed!

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.

In [33]:
```
model = LinearRegression()
errors = []

for v in np.unique(train['period']):
    tr = train[train.period == v].drop("period", axis=1)
    te = test[test.period == v].drop("period", axis=1)
    reg = model.fit(design_matrix(tr.drop("duration", axis=1)), tr["durat
ion"])
    error = te["duration"] - reg.predict(design_matrix(te.drop("duration"
, axis=1)))
    errors += error.tolist()


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

Out[33]: 246.62868831165173

In [34]:
```
grader.check("q4d")
```

Out[34]: 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.

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.

This is due to the fact that there are different distribution for each different period (not just shifted ver of the same distrbution). We separate the regressions so that we can have different values for other variables in the regression. Thus, we account for the distributions better, something that linear regression does not really take into account.

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.

In [35]:
```
model = LinearRegression()

y_train = train["duration"]
speed_train = train["speed"]
x_train = train.drop("duration", axis=1).drop("speed", axis=1)

y_test = test["duration"]
speed_test = test["speed"]
x_test = test.drop("duration", axis=1).drop("speed", axis=1)

reg = model.fit(design_matrix(x_train), speed_train)
pred_test = reg.predict(design_matrix(x_test))

speed_rmse = rmse(y_test - (x_test["distance"] / pred_test) * 3600)
speed_rmse
```

Out[35]: 243.0179836851495

In [36]:
```
grader.check("q4f")
```

Out[36]: 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.

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.

In [37]:
```
model = LinearRegression()
choices = ['period', 'region', 'weekend']

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

def speed_error(predictions, observations):
    """Duration error between speed predictions and duration observation
s"""
    return (observations['distance'] / predictions * 3600)  - observation
s['duration']

def tree_regression_errors(outcome='duration', error_fn=duration_error):
    """Return errors for all examples in test using a tree regression mod
el."""
    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]

        reg = model.fit(design_matrix(v_train.drop(outcome, axis=1)), v_t
rain[outcome])
        y_pred = reg.predict(design_matrix(v_test.drop(outcome, axis=1)))

        error = error_fn(y_pred, v_test)
        errors += error.tolist()

    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)
print('Speed:', tree_speed_rmse)
```

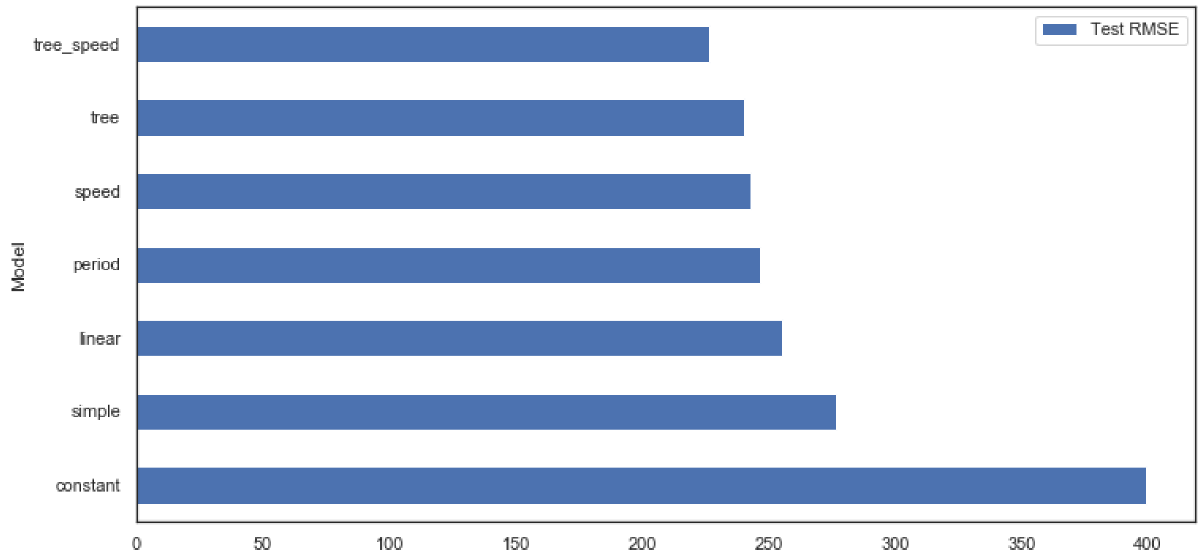
```
Duration: 240.3395219270353
Speed: 226.90793945018308
```

In [38]:
```
grader.check("q4g")
```

Out[38]: All tests passed!

Here's a summary of your results:

In [39]:
```
models = ['constant', 'simple', 'linear', 'period', 'speed', 'tree', 'tre
e_speed']
pd.DataFrame.from_dict({
    'Model': models,
    'Test RMSE': [eval(m + '_rmse') for m in models]
}).set_index('Model').plot(kind='barh');
```



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.

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

In [40]:
```
from keras.models import Sequential
from keras.layers import Dense
from keras import optimizers
from keras.layers import Dropout
from keras import regularizers
import tensorflow as tf
from tensorflow import keras

#model 1
# model = Sequential([
#     Dense(32, activation='relu', input_shape=(36,)),
#     Dense(32, activation='relu'),
#     Dense(1, activation='softmax'),
# ])

# sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

# model.compile(optimizer='sgd',
#                 loss='mean_squared_error',
#                 metrics=['accuracy'])

# hist = model.fit(design_matrix(x_train), y_train,
#             batch_size=32, epochs=100)

#model 2

# model_2 = Sequential([
#     Dense(1000, activation='relu', input_shape=(12,)),
#     Dense(1000, activation='relu'),
#     Dense(1000, activation='relu'),
#     Dense(1000, activation='relu'),
#     Dense(1, activation='sigmoid'),])

# model_2.compile(optimizer='sgd',
#                 loss='mean_squared_error',
#                 metrics=['accuracy'])

# hist_2 = model_2.fit(X, Y, batch_size=256, epochs=100)


#model3
# model_3 = Sequential([
#     Dense(1000, activation='relu', kernel_regularizer=regularizers.l2
(0.01), input_shape=(12,)),
#     Dropout(0.3),
#     Dense(1000, activation='relu', kernel_regularizer=regularizers.l2
(0.01)),
#     Dropout(0.3),
#     Dense(1000, activation='relu', kernel_regularizer=regularizers.l2
(0.01)),
#     Dropout(0.3),
#     Dense(1, activation='sigmoid', kernel_regularizer=regularizers.l2
(0.01)),
# ])

# sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
```

```
# model_3.compile(loss='mean_squared_error', optimizer='sgd')


# hist_3 = model_3.fit(X, Y ,batch_size=64, epochs=100)

#model4

# model_4 = Sequential([
#     Dense(32, activation='relu', kernel_regularizer=regularizers.l2(0.0
1), input_shape=(12,)),
#     Dropout(0.3),
#     Dense(32, activation='relu', kernel_regularizer=regularizers.l2(0.0
1)),
#     Dropout(0.3),
#     Dense(32, activation='relu', kernel_regularizer=regularizers.l2(0.0
1)),
#     Dropout(0.3),
#     Dense(1, activation='sigmoid', kernel_regularizer=regularizers.l2
(0.01)),
# ])

# sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
# model_4.compile(loss='mean_squared_error', optimizer='sgd')


# hist_4 = model_4.fit(X, Y ,batch_size=32, epochs=100)


# model = Sequential()
# model.add(Dense(1000, input_dim=36, kernel_regularizer=regularizers.l2
(0.01), activation='relu'))
# model.add(Dropout(0.3))
# model.add(Dense(300, kernel_regularizer=regularizers.l2(0.01), activati
on='relu'))
# model.add(Dropout(0.3))
# model.add(Dense(128, kernel_regularizer=regularizers.l2(0.01), activati
on='relu'))
# model.add(Dropout(0.3))
# model.add(Dense(1, kernel_initializer='normal'))

# sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
# adam = keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.
999, amsgrad=False)
# model.compile(loss='mean_squared_error', optimizer='adam')


# model.fit(design_matrix(x_train), y_train, epochs=50, batch_size=32, ve
rbose=1)
```

```
Using TensorFlow backend.
```

In [41]:
```
num_vars = ['pickup_lon', 'pickup_lat', 'dropoff_lon', 'dropoff_lat', 'di
stance']
cat_vars = ['hour', 'day', 'region']

print(train.shape)

scaler = StandardScaler()
scaler.fit(train[num_vars])
X = augment(train)
X = X.drop("pickup_datetime", axis=1).drop("dropoff_datetime", axis=1).dr
op("date", axis=1).drop("speed", axis=1)

scaled = X[num_vars].copy()
scaled.iloc[:,:] = scaler.transform(scaled) # Convert to standard units

scaled.head()
X = X.drop(num_vars, axis=1)
X = pd.concat([scaled, X], axis=1)
# print(X.head())

# split into input (X) and output (Y) variables
X = X.drop("duration", axis=1).values
Y = train["duration"].values

import xgboost as xgb

Xtr, Xv, ytr, yv = sklearn.model_selection.train_test_split(X, Y, test_si
ze=0.2, random_state=42)
dtrain = xgb.DMatrix(Xtr, label=ytr)
dvalid = xgb.DMatrix(Xv, label=yv)

print(ytr.shape)
print(yv.shape)

watchlist = [(dtrain, 'train'), (dvalid, 'valid')]

xgb_pars = {'min_child_weight': 100, 'eta': 0.01, 'colsample_bytree': 1,
'max_depth': 10,
            'subsample': 0.8, 'lambda': 1., 'nthread': 4, 'booster' : 'gb
tree', 'silent': 1,
            'eval_metric': 'rmse', 'objective': 'reg:linear'}

model = xgb.train(xgb_pars, dtrain, 10000, watchlist, early_stopping_roun
ds=50,
                  maximize=False, verbose_eval=10)

dtrain = xgb.DMatrix(X)
train_preds = model.predict(dtrain)
train_rmse = rmse(Y - train_preds)
train_rmse
```

```
(53680, 16)
(42944,)
(10736,)
[0]	train-rmse:769.66760	valid-rmse:767.47974
Multiple eval metrics have been passed: 'valid-rmse' will be used for e
arly stopping.

Will train until valid-rmse hasn't improved in 50 rounds.
[10]	train-rmse:702.69141	valid-rmse:700.80023
[20]	train-rmse:642.59485	valid-rmse:640.95044
[30]	train-rmse:588.72613	valid-rmse:587.35315
[40]	train-rmse:540.53180	valid-rmse:539.40826
[50]	train-rmse:497.50671	valid-rmse:496.63885
[60]	train-rmse:459.20325	valid-rmse:458.64380
[70]	train-rmse:425.13187	valid-rmse:424.90198
[80]	train-rmse:394.87622	valid-rmse:394.94913
[90]	train-rmse:368.18335	valid-rmse:368.58179
[100]	train-rmse:344.67224	valid-rmse:345.39014
[110]	train-rmse:323.99966	valid-rmse:325.07922
[120]	train-rmse:305.95142	valid-rmse:307.40192
[130]	train-rmse:290.17368	valid-rmse:292.03357
[140]	train-rmse:276.46487	valid-rmse:278.71152
[150]	train-rmse:264.55191	valid-rmse:267.18970
[160]	train-rmse:254.21693	valid-rmse:257.25699
[170]	train-rmse:245.38065	valid-rmse:248.85190
[180]	train-rmse:237.74429	valid-rmse:241.65524
[190]	train-rmse:231.11462	valid-rmse:235.43394
[200]	train-rmse:225.42476	valid-rmse:230.17267
[210]	train-rmse:220.57460	valid-rmse:225.68915
[220]	train-rmse:216.36713	valid-rmse:221.85472
[230]	train-rmse:212.73436	valid-rmse:218.60054
[240]	train-rmse:209.64668	valid-rmse:215.83707
[250]	train-rmse:206.89973	valid-rmse:213.41930
[260]	train-rmse:204.51666	valid-rmse:211.33537
[270]	train-rmse:202.38231	valid-rmse:209.51829
[280]	train-rmse:200.61571	valid-rmse:208.02260
[290]	train-rmse:198.99310	valid-rmse:206.69060
[300]	train-rmse:197.49037	valid-rmse:205.49800
[310]	train-rmse:196.17629	valid-rmse:204.46089
[320]	train-rmse:195.01010	valid-rmse:203.57468
[330]	train-rmse:193.92145	valid-rmse:202.74243
[340]	train-rmse:192.99100	valid-rmse:202.04224
[350]	train-rmse:192.04022	valid-rmse:201.36130
[360]	train-rmse:191.17410	valid-rmse:200.76453
[370]	train-rmse:190.37648	valid-rmse:200.21609
[380]	train-rmse:189.60053	valid-rmse:199.71902
[390]	train-rmse:188.87321	valid-rmse:199.24571
[400]	train-rmse:188.24771	valid-rmse:198.84376
[410]	train-rmse:187.67697	valid-rmse:198.49130
[420]	train-rmse:187.13774	valid-rmse:198.15846
[430]	train-rmse:186.64374	valid-rmse:197.86883
[440]	train-rmse:186.14796	valid-rmse:197.57077
[450]	train-rmse:185.68138	valid-rmse:197.27492
[460]	train-rmse:185.29134	valid-rmse:197.04478
[470]	train-rmse:184.82657	valid-rmse:196.76123
[480]	train-rmse:184.43117	valid-rmse:196.53915
[490]	train-rmse:184.03458	valid-rmse:196.31558
```

```
[500]	train-rmse:183.69116	valid-rmse:196.08266
[510]	train-rmse:183.41010	valid-rmse:195.92049
[520]	train-rmse:183.04411	valid-rmse:195.70308
[530]	train-rmse:182.73306	valid-rmse:195.54828
[540]	train-rmse:182.42738	valid-rmse:195.39026
[550]	train-rmse:182.10678	valid-rmse:195.20579
[560]	train-rmse:181.83206	valid-rmse:195.06271
[570]	train-rmse:181.63068	valid-rmse:194.98366
[580]	train-rmse:181.39258	valid-rmse:194.86171
[590]	train-rmse:181.14441	valid-rmse:194.71478
[600]	train-rmse:180.89873	valid-rmse:194.59164
[610]	train-rmse:180.71060	valid-rmse:194.50151
[620]	train-rmse:180.51811	valid-rmse:194.41618
[630]	train-rmse:180.24403	valid-rmse:194.28420
[640]	train-rmse:180.03799	valid-rmse:194.20039
[650]	train-rmse:179.77063	valid-rmse:194.07643
[660]	train-rmse:179.55148	valid-rmse:193.97072
[670]	train-rmse:179.42866	valid-rmse:193.91206
[680]	train-rmse:179.22632	valid-rmse:193.81117
[690]	train-rmse:179.07748	valid-rmse:193.75737
[700]	train-rmse:178.86220	valid-rmse:193.67412
[710]	train-rmse:178.68379	valid-rmse:193.59380
[720]	train-rmse:178.50438	valid-rmse:193.50043
[730]	train-rmse:178.32831	valid-rmse:193.40076
[740]	train-rmse:178.15311	valid-rmse:193.31139
[750]	train-rmse:178.05545	valid-rmse:193.27116
[760]	train-rmse:177.84164	valid-rmse:193.17841
[770]	train-rmse:177.69212	valid-rmse:193.11096
[780]	train-rmse:177.56838	valid-rmse:193.07477
[790]	train-rmse:177.45023	valid-rmse:193.02910
[800]	train-rmse:177.30640	valid-rmse:192.98299
[810]	train-rmse:177.17720	valid-rmse:192.92041
[820]	train-rmse:176.95763	valid-rmse:192.81638
[830]	train-rmse:176.78346	valid-rmse:192.76057
[840]	train-rmse:176.65703	valid-rmse:192.71359
[850]	train-rmse:176.55687	valid-rmse:192.67255
[860]	train-rmse:176.37650	valid-rmse:192.60089
[870]	train-rmse:176.23717	valid-rmse:192.54799
[880]	train-rmse:176.07989	valid-rmse:192.47824
[890]	train-rmse:175.87326	valid-rmse:192.38867
[900]	train-rmse:175.72339	valid-rmse:192.31934
[910]	train-rmse:175.63919	valid-rmse:192.29367
[920]	train-rmse:175.52490	valid-rmse:192.25166
[930]	train-rmse:175.36171	valid-rmse:192.20291
[940]	train-rmse:175.23282	valid-rmse:192.15053
[950]	train-rmse:175.08164	valid-rmse:192.08592
[960]	train-rmse:174.98071	valid-rmse:192.04123
[970]	train-rmse:174.86826	valid-rmse:192.01048
[980]	train-rmse:174.77004	valid-rmse:191.98032
[990]	train-rmse:174.61522	valid-rmse:191.92488
[1000]	train-rmse:174.45642	valid-rmse:191.85719
[1010]	train-rmse:174.31819	valid-rmse:191.80556
[1020]	train-rmse:174.23032	valid-rmse:191.76117
[1030]	train-rmse:174.09447	valid-rmse:191.73595
[1040]	train-rmse:173.99000	valid-rmse:191.69975
[1050]	train-rmse:173.88316	valid-rmse:191.67415
[1060]	train-rmse:173.78171	valid-rmse:191.63460
```

```
[1070]	train-rmse:173.67549	valid-rmse:191.60356
[1080]	train-rmse:173.55467	valid-rmse:191.55013
[1090]	train-rmse:173.43382	valid-rmse:191.49553
[1100]	train-rmse:173.30862	valid-rmse:191.45441
[1110]	train-rmse:173.23579	valid-rmse:191.43974
[1120]	train-rmse:173.14455	valid-rmse:191.42172
[1130]	train-rmse:173.05286	valid-rmse:191.37955
[1140]	train-rmse:172.95947	valid-rmse:191.35319
[1150]	train-rmse:172.85998	valid-rmse:191.32201
[1160]	train-rmse:172.73752	valid-rmse:191.26778
[1170]	train-rmse:172.62508	valid-rmse:191.24103
[1180]	train-rmse:172.48663	valid-rmse:191.19125
[1190]	train-rmse:172.40250	valid-rmse:191.16744
[1200]	train-rmse:172.31737	valid-rmse:191.14288
[1210]	train-rmse:172.24281	valid-rmse:191.12752
[1220]	train-rmse:172.16484	valid-rmse:191.09103
[1230]	train-rmse:172.10284	valid-rmse:191.07571
[1240]	train-rmse:171.98204	valid-rmse:191.02545
[1250]	train-rmse:171.90598	valid-rmse:190.99303
[1260]	train-rmse:171.80800	valid-rmse:190.96979
[1270]	train-rmse:171.72519	valid-rmse:190.94167
[1280]	train-rmse:171.60254	valid-rmse:190.90573
[1290]	train-rmse:171.51211	valid-rmse:190.87617
[1300]	train-rmse:171.43777	valid-rmse:190.85771
[1310]	train-rmse:171.33470	valid-rmse:190.82108
[1320]	train-rmse:171.25143	valid-rmse:190.79028
[1330]	train-rmse:171.18323	valid-rmse:190.77924
[1340]	train-rmse:171.10811	valid-rmse:190.75539
[1350]	train-rmse:170.97906	valid-rmse:190.68742
[1360]	train-rmse:170.89053	valid-rmse:190.65292
[1370]	train-rmse:170.80687	valid-rmse:190.63510
[1380]	train-rmse:170.71670	valid-rmse:190.59999
[1390]	train-rmse:170.61675	valid-rmse:190.57300
[1400]	train-rmse:170.47394	valid-rmse:190.51585
[1410]	train-rmse:170.40611	valid-rmse:190.50023
[1420]	train-rmse:170.34111	valid-rmse:190.47067
[1430]	train-rmse:170.23331	valid-rmse:190.42674
[1440]	train-rmse:170.10928	valid-rmse:190.37431
[1450]	train-rmse:170.06718	valid-rmse:190.37073
[1460]	train-rmse:169.96526	valid-rmse:190.33176
[1470]	train-rmse:169.88647	valid-rmse:190.31621
[1480]	train-rmse:169.80370	valid-rmse:190.29387
[1490]	train-rmse:169.73717	valid-rmse:190.26813
[1500]	train-rmse:169.65836	valid-rmse:190.24057
[1510]	train-rmse:169.57507	valid-rmse:190.22852
[1520]	train-rmse:169.46925	valid-rmse:190.19458
[1530]	train-rmse:169.34462	valid-rmse:190.15082
[1540]	train-rmse:169.28926	valid-rmse:190.12291
[1550]	train-rmse:169.16534	valid-rmse:190.10130
[1560]	train-rmse:169.08758	valid-rmse:190.08800
[1570]	train-rmse:169.01296	valid-rmse:190.06795
[1580]	train-rmse:168.93465	valid-rmse:190.04321
[1590]	train-rmse:168.84335	valid-rmse:190.00935
[1600]	train-rmse:168.75610	valid-rmse:189.99347
[1610]	train-rmse:168.66783	valid-rmse:189.98320
[1620]	train-rmse:168.55580	valid-rmse:189.94846
[1630]	train-rmse:168.48341	valid-rmse:189.92767
```

```
[1640]	train-rmse:168.37766	valid-rmse:189.89499
[1650]	train-rmse:168.29407	valid-rmse:189.87566
[1660]	train-rmse:168.19185	valid-rmse:189.83539
[1670]	train-rmse:168.10969	valid-rmse:189.81421
[1680]	train-rmse:168.03961	valid-rmse:189.79932
[1690]	train-rmse:167.92226	valid-rmse:189.77280
[1700]	train-rmse:167.83806	valid-rmse:189.75386
[1710]	train-rmse:167.79303	valid-rmse:189.76051
[1720]	train-rmse:167.71233	valid-rmse:189.73074
[1730]	train-rmse:167.66425	valid-rmse:189.72122
[1740]	train-rmse:167.56465	valid-rmse:189.70285
[1750]	train-rmse:167.50040	valid-rmse:189.68036
[1760]	train-rmse:167.43787	valid-rmse:189.66363
[1770]	train-rmse:167.36144	valid-rmse:189.63150
[1780]	train-rmse:167.29295	valid-rmse:189.62607
[1790]	train-rmse:167.22792	valid-rmse:189.61696
[1800]	train-rmse:167.13448	valid-rmse:189.58766
[1810]	train-rmse:167.07040	valid-rmse:189.57266
[1820]	train-rmse:167.01817	valid-rmse:189.55633
[1830]	train-rmse:166.97745	valid-rmse:189.54840
[1840]	train-rmse:166.90701	valid-rmse:189.54649
[1850]	train-rmse:166.83318	valid-rmse:189.52446
[1860]	train-rmse:166.74677	valid-rmse:189.49892
[1870]	train-rmse:166.65758	valid-rmse:189.46718
[1880]	train-rmse:166.59793	valid-rmse:189.46347
[1890]	train-rmse:166.47925	valid-rmse:189.41660
[1900]	train-rmse:166.39619	valid-rmse:189.39496
[1910]	train-rmse:166.32227	valid-rmse:189.38039
[1920]	train-rmse:166.25365	valid-rmse:189.36598
[1930]	train-rmse:166.18950	valid-rmse:189.35713
[1940]	train-rmse:166.10890	valid-rmse:189.32838
[1950]	train-rmse:166.06363	valid-rmse:189.31699
[1960]	train-rmse:166.00354	valid-rmse:189.31187
[1970]	train-rmse:165.91563	valid-rmse:189.28320
[1980]	train-rmse:165.83453	valid-rmse:189.26453
[1990]	train-rmse:165.72792	valid-rmse:189.23247
[2000]	train-rmse:165.63925	valid-rmse:189.21175
[2010]	train-rmse:165.54169	valid-rmse:189.19394
[2020]	train-rmse:165.46738	valid-rmse:189.16597
[2030]	train-rmse:165.37311	valid-rmse:189.13266
[2040]	train-rmse:165.29173	valid-rmse:189.10425
[2050]	train-rmse:165.19644	valid-rmse:189.08078
[2060]	train-rmse:165.14430	valid-rmse:189.07430
[2070]	train-rmse:165.09123	valid-rmse:189.05885
[2080]	train-rmse:165.00096	valid-rmse:189.04411
[2090]	train-rmse:164.94295	valid-rmse:189.02988
[2100]	train-rmse:164.87759	valid-rmse:189.01842
[2110]	train-rmse:164.82783	valid-rmse:189.02187
[2120]	train-rmse:164.77266	valid-rmse:189.00626
[2130]	train-rmse:164.70419	valid-rmse:188.97458
[2140]	train-rmse:164.66647	valid-rmse:188.96050
[2150]	train-rmse:164.61395	valid-rmse:188.95268
[2160]	train-rmse:164.53294	valid-rmse:188.93677
[2170]	train-rmse:164.49301	valid-rmse:188.93254
[2180]	train-rmse:164.43639	valid-rmse:188.91899
[2190]	train-rmse:164.36522	valid-rmse:188.90416
[2200]	train-rmse:164.28732	valid-rmse:188.89468
```

```
[2210]	train-rmse:164.23082	valid-rmse:188.88159
[2220]	train-rmse:164.15065	valid-rmse:188.86360
[2230]	train-rmse:164.11530	valid-rmse:188.85210
[2240]	train-rmse:164.06076	valid-rmse:188.83531
[2250]	train-rmse:164.00290	valid-rmse:188.82513
[2260]	train-rmse:163.96536	valid-rmse:188.82426
[2270]	train-rmse:163.87677	valid-rmse:188.79031
[2280]	train-rmse:163.80817	valid-rmse:188.76915
[2290]	train-rmse:163.76706	valid-rmse:188.76743
[2300]	train-rmse:163.69194	valid-rmse:188.75142
[2310]	train-rmse:163.62158	valid-rmse:188.72957
[2320]	train-rmse:163.55402	valid-rmse:188.71042
[2330]	train-rmse:163.49986	valid-rmse:188.70393
[2340]	train-rmse:163.41258	valid-rmse:188.68764
[2350]	train-rmse:163.36720	valid-rmse:188.67096
[2360]	train-rmse:163.29056	valid-rmse:188.65427
[2370]	train-rmse:163.20605	valid-rmse:188.64331
[2380]	train-rmse:163.12497	valid-rmse:188.62238
[2390]	train-rmse:163.04759	valid-rmse:188.60519
[2400]	train-rmse:162.97742	valid-rmse:188.58469
[2410]	train-rmse:162.92583	valid-rmse:188.57361
[2420]	train-rmse:162.83073	valid-rmse:188.56187
[2430]	train-rmse:162.75804	valid-rmse:188.54596
[2440]	train-rmse:162.69205	valid-rmse:188.52771
[2450]	train-rmse:162.61850	valid-rmse:188.51506
[2460]	train-rmse:162.54050	valid-rmse:188.50499
[2470]	train-rmse:162.48961	valid-rmse:188.49696
[2480]	train-rmse:162.42476	valid-rmse:188.49002
[2490]	train-rmse:162.34015	valid-rmse:188.47075
[2500]	train-rmse:162.26758	valid-rmse:188.45081
[2510]	train-rmse:162.22226	valid-rmse:188.44072
[2520]	train-rmse:162.15475	valid-rmse:188.42575
[2530]	train-rmse:162.09766	valid-rmse:188.41737
[2540]	train-rmse:162.05837	valid-rmse:188.40878
[2550]	train-rmse:162.01321	valid-rmse:188.40961
[2560]	train-rmse:161.89197	valid-rmse:188.38444
[2570]	train-rmse:161.83528	valid-rmse:188.35776
[2580]	train-rmse:161.78058	valid-rmse:188.34489
[2590]	train-rmse:161.73347	valid-rmse:188.33081
[2600]	train-rmse:161.65486	valid-rmse:188.31761
[2610]	train-rmse:161.61623	valid-rmse:188.31177
[2620]	train-rmse:161.56438	valid-rmse:188.30675
[2630]	train-rmse:161.51982	valid-rmse:188.29842
[2640]	train-rmse:161.45209	valid-rmse:188.28654
[2650]	train-rmse:161.39029	valid-rmse:188.27333
[2660]	train-rmse:161.34628	valid-rmse:188.26176
[2670]	train-rmse:161.28291	valid-rmse:188.25911
[2680]	train-rmse:161.22786	valid-rmse:188.24298
[2690]	train-rmse:161.16684	valid-rmse:188.23271
[2700]	train-rmse:161.09230	valid-rmse:188.22681
[2710]	train-rmse:161.04150	valid-rmse:188.21457
[2720]	train-rmse:160.98370	valid-rmse:188.20264
[2730]	train-rmse:160.90558	valid-rmse:188.18558
[2740]	train-rmse:160.85362	valid-rmse:188.17186
[2750]	train-rmse:160.80112	valid-rmse:188.16235
[2760]	train-rmse:160.73285	valid-rmse:188.14967
[2770]	train-rmse:160.67541	valid-rmse:188.14668
```

```
[2780]	train-rmse:160.61528	valid-rmse:188.13193
[2790]	train-rmse:160.56328	valid-rmse:188.13165
[2800]	train-rmse:160.49832	valid-rmse:188.11780
[2810]	train-rmse:160.42941	valid-rmse:188.10245
[2820]	train-rmse:160.38275	valid-rmse:188.09924
[2830]	train-rmse:160.33681	valid-rmse:188.08345
[2840]	train-rmse:160.28330	valid-rmse:188.06926
[2850]	train-rmse:160.22122	valid-rmse:188.05653
[2860]	train-rmse:160.13106	valid-rmse:188.03812
[2870]	train-rmse:160.07739	valid-rmse:188.03554
[2880]	train-rmse:160.01544	valid-rmse:188.02272
[2890]	train-rmse:159.95770	valid-rmse:188.01404
[2900]	train-rmse:159.91151	valid-rmse:187.99878
[2910]	train-rmse:159.86810	valid-rmse:187.99762
[2920]	train-rmse:159.80911	valid-rmse:187.98816
[2930]	train-rmse:159.75690	valid-rmse:187.98355
[2940]	train-rmse:159.68599	valid-rmse:187.97804
[2950]	train-rmse:159.64316	valid-rmse:187.97572
[2960]	train-rmse:159.58588	valid-rmse:187.96489
[2970]	train-rmse:159.51112	valid-rmse:187.96518
[2980]	train-rmse:159.44431	valid-rmse:187.95410
[2990]	train-rmse:159.40437	valid-rmse:187.94434
[3000]	train-rmse:159.35080	valid-rmse:187.93600
[3010]	train-rmse:159.30652	valid-rmse:187.92641
[3020]	train-rmse:159.27176	valid-rmse:187.92047
[3030]	train-rmse:159.22060	valid-rmse:187.91081
[3040]	train-rmse:159.14305	valid-rmse:187.89282
[3050]	train-rmse:159.07136	valid-rmse:187.88441
[3060]	train-rmse:159.02757	valid-rmse:187.87619
[3070]	train-rmse:158.96890	valid-rmse:187.85577
[3080]	train-rmse:158.90813	valid-rmse:187.84897
[3090]	train-rmse:158.86403	valid-rmse:187.83987
[3100]	train-rmse:158.80278	valid-rmse:187.81839
[3110]	train-rmse:158.74051	valid-rmse:187.80783
[3120]	train-rmse:158.70226	valid-rmse:187.80997
[3130]	train-rmse:158.65240	valid-rmse:187.80541
[3140]	train-rmse:158.59160	valid-rmse:187.78783
[3150]	train-rmse:158.54323	valid-rmse:187.77776
[3160]	train-rmse:158.47530	valid-rmse:187.76491
[3170]	train-rmse:158.39714	valid-rmse:187.74626
[3180]	train-rmse:158.33878	valid-rmse:187.73071
[3190]	train-rmse:158.31250	valid-rmse:187.72540
[3200]	train-rmse:158.27185	valid-rmse:187.71786
[3210]	train-rmse:158.19756	valid-rmse:187.70305
[3220]	train-rmse:158.13200	valid-rmse:187.69608
[3230]	train-rmse:158.07951	valid-rmse:187.69412
[3240]	train-rmse:158.01608	valid-rmse:187.70100
[3250]	train-rmse:157.94823	valid-rmse:187.69432
[3260]	train-rmse:157.90985	valid-rmse:187.69412
[3270]	train-rmse:157.85031	valid-rmse:187.69353
[3280]	train-rmse:157.79533	valid-rmse:187.68442
[3290]	train-rmse:157.75697	valid-rmse:187.68292
[3300]	train-rmse:157.68247	valid-rmse:187.67596
[3310]	train-rmse:157.63216	valid-rmse:187.67578
[3320]	train-rmse:157.61006	valid-rmse:187.68274
[3330]	train-rmse:157.55330	valid-rmse:187.67618
[3340]	train-rmse:157.51395	valid-rmse:187.66951
```

```
[3350]	train-rmse:157.45619	valid-rmse:187.66254
[3360]	train-rmse:157.40053	valid-rmse:187.66081
[3370]	train-rmse:157.34833	valid-rmse:187.65358
[3380]	train-rmse:157.30777	valid-rmse:187.65839
[3390]	train-rmse:157.26150	valid-rmse:187.65240
[3400]	train-rmse:157.19635	valid-rmse:187.64667
[3410]	train-rmse:157.14224	valid-rmse:187.64661
[3420]	train-rmse:157.09317	valid-rmse:187.63393
[3430]	train-rmse:157.04419	valid-rmse:187.62962
[3440]	train-rmse:156.99338	valid-rmse:187.62521
[3450]	train-rmse:156.92171	valid-rmse:187.60750
[3460]	train-rmse:156.87021	valid-rmse:187.60098
[3470]	train-rmse:156.80072	valid-rmse:187.59099
[3480]	train-rmse:156.74435	valid-rmse:187.58781
[3490]	train-rmse:156.70465	valid-rmse:187.58209
[3500]	train-rmse:156.63669	valid-rmse:187.57806
[3510]	train-rmse:156.59271	valid-rmse:187.57091
[3520]	train-rmse:156.53874	valid-rmse:187.56677
[3530]	train-rmse:156.45401	valid-rmse:187.55606
[3540]	train-rmse:156.40559	valid-rmse:187.55847
[3550]	train-rmse:156.35472	valid-rmse:187.55931
[3560]	train-rmse:156.30714	valid-rmse:187.55490
[3570]	train-rmse:156.24519	valid-rmse:187.54982
[3580]	train-rmse:156.18784	valid-rmse:187.54494
[3590]	train-rmse:156.14394	valid-rmse:187.53998
[3600]	train-rmse:156.09888	valid-rmse:187.54291
[3610]	train-rmse:156.05142	valid-rmse:187.53186
[3620]	train-rmse:156.01140	valid-rmse:187.53217
[3630]	train-rmse:155.94858	valid-rmse:187.53079
[3640]	train-rmse:155.89865	valid-rmse:187.52743
[3650]	train-rmse:155.86562	valid-rmse:187.53249
[3660]	train-rmse:155.81760	valid-rmse:187.52882
[3670]	train-rmse:155.76656	valid-rmse:187.52231
[3680]	train-rmse:155.72176	valid-rmse:187.51669
[3690]	train-rmse:155.66661	valid-rmse:187.50803
[3700]	train-rmse:155.62236	valid-rmse:187.50603
[3710]	train-rmse:155.55299	valid-rmse:187.50070
[3720]	train-rmse:155.50323	valid-rmse:187.49788
[3730]	train-rmse:155.45027	valid-rmse:187.49451
[3740]	train-rmse:155.38216	valid-rmse:187.48747
[3750]	train-rmse:155.32455	valid-rmse:187.47699
[3760]	train-rmse:155.26181	valid-rmse:187.47124
[3770]	train-rmse:155.19441	valid-rmse:187.47128
[3780]	train-rmse:155.13393	valid-rmse:187.46120
[3790]	train-rmse:155.08702	valid-rmse:187.46115
[3800]	train-rmse:155.01315	valid-rmse:187.44710
[3810]	train-rmse:154.96565	valid-rmse:187.45151
[3820]	train-rmse:154.92006	valid-rmse:187.44113
[3830]	train-rmse:154.86412	valid-rmse:187.43117
[3840]	train-rmse:154.82805	valid-rmse:187.43723
[3850]	train-rmse:154.78378	valid-rmse:187.42956
[3860]	train-rmse:154.73454	valid-rmse:187.42290
[3870]	train-rmse:154.64958	valid-rmse:187.39407
[3880]	train-rmse:154.60361	valid-rmse:187.39053
[3890]	train-rmse:154.55129	valid-rmse:187.38162
[3900]	train-rmse:154.51353	valid-rmse:187.37852
[3910]	train-rmse:154.46863	valid-rmse:187.37761
```

```
[3920]	train-rmse:154.44511	valid-rmse:187.37337
[3930]	train-rmse:154.39621	valid-rmse:187.36835
[3940]	train-rmse:154.34074	valid-rmse:187.35939
[3950]	train-rmse:154.29334	valid-rmse:187.36078
[3960]	train-rmse:154.23880	valid-rmse:187.34750
[3970]	train-rmse:154.18576	valid-rmse:187.34587
[3980]	train-rmse:154.13367	valid-rmse:187.34448
[3990]	train-rmse:154.07292	valid-rmse:187.33556
[4000]	train-rmse:154.01794	valid-rmse:187.33128
[4010]	train-rmse:153.96924	valid-rmse:187.33611
[4020]	train-rmse:153.92445	valid-rmse:187.33199
[4030]	train-rmse:153.86583	valid-rmse:187.32924
[4040]	train-rmse:153.83315	valid-rmse:187.33902
[4050]	train-rmse:153.79039	valid-rmse:187.33334
[4060]	train-rmse:153.74114	valid-rmse:187.33501
Stopping. Best iteration:
[4014]	train-rmse:153.94971	valid-rmse:187.32817
```

Out[41]: 160.99945536716012

In [42]:

```
X_test = augment(test)
X_test = X_test.drop("pickup_datetime", axis=1).drop("dropoff_datetime",
axis=1).drop("date", axis=1).drop("speed",axis=1)

scaled = X_test[num_vars].copy()
scaled.iloc[:,:] = scaler.transform(scaled) # Convert to standard units

scaled.head()
X_test = X_test.drop(num_vars, axis=1)
X_test = pd.concat([scaled, X_test], axis=1)

# split into input (X) and output (Y) variables
X_test = X_test.drop("duration", axis=1).values
Y_test = test["duration"].values

dtest = xgb.DMatrix(X_test)

preds = model.predict(dtest)
print(preds)
print(Y_test)
#print((preds-Y_test)[1:50])
test_rmse = rmse(Y_test - preds)
test_rmse
```

```
[1565.874   1002.5495   334.05957 ...  377.90903  399.56726  719.47437]
[1380 1128  372 ...  382  363  482]
```

Out[42]: 187.4018633543184

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.

In [43]:
```
print(f"We augmented the data using our own methods and we were able to t
rain a xgboost model (extreme gradient boosting tree) and got a training
 RMSE of {train_rmse}. This was done by splitting the training \
set into a training set and validation set. Once we ran the testing set t
hrough this model, we found\
a testing RMSE of {test_rmse}.")
```

```
We augmented the data using our own methods and we were able to train a
xgboost model (extreme gradient boosting tree) and got a training RMSE
of 160.99945536716012. This was done by splitting the training set into
a training set and validation set. Once we ran the testing set through
this model, we founda testing RMSE of 187.4018633543184.
```

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

There are 2 reasons for why we ended up choosing xgboost instead of some fancy neural network. Firstly, it was much faster to run than any other complex models (it took around 2 minutes to run locally on my mac). This can be compared to the neural network (which is commented out) which took forever to run. Secondly, this mdoel deals with categorial data much better tha neural networks and regression (doesn't need one-hot encoding). Since we didn't have to one-hot encode the data, dimensionality was decreased through our own augmentation. We also added a validation set so we could measure when to stop training to avoid overfitting. Lastly, we also made use of the low runtime by increasing the number of iterations that xgboost does (since each iteration occurs extremely fast), allowing it to converge to the lowest validation set. All of this helped us to achieve a test RMSE of around 187 with a relatively fast runtime.

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.

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?

In [44]:

```
# 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)
```

Your file has been exported. Download it here (proj3.pdf)!