proj3 (1)

August 2, 2021

1 Project: Predicting NYC Taxi Ride Duration

1.1 This Assignment

In this project, I will create a regression model that predicts the travel time of a taxi ride in New York.

After this project, I 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:

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

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
```

1.2 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 I've placed a simple random sample of the data into `taxi.db`, a SQLite database. .

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

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

1.3 Part 1: Data Selection and Cleaning

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

Use a SQL query to load the `taxi` table from `taxi.db` into a Pandas DataFrame called `all_taxi`.

Only include 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)

[3]:
```
import sqlite3

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

sql = """
SELECT * FROM taxi
WHERE (pickup_lon >= -74.03) AND (pickup_lon <= -73.75)
AND (pickup_lat <= 40.88) AND (pickup_lat >= 40.6)
AND (dropoff_lon >= -74.03) AND (dropoff_lon <= -73.75)
AND (dropoff_lat <=40.88) AND (dropoff_lat >= 40.6)
"""


all_taxi = pd.read_sql(sql, conn)
all_taxi.head()
```

[3]:

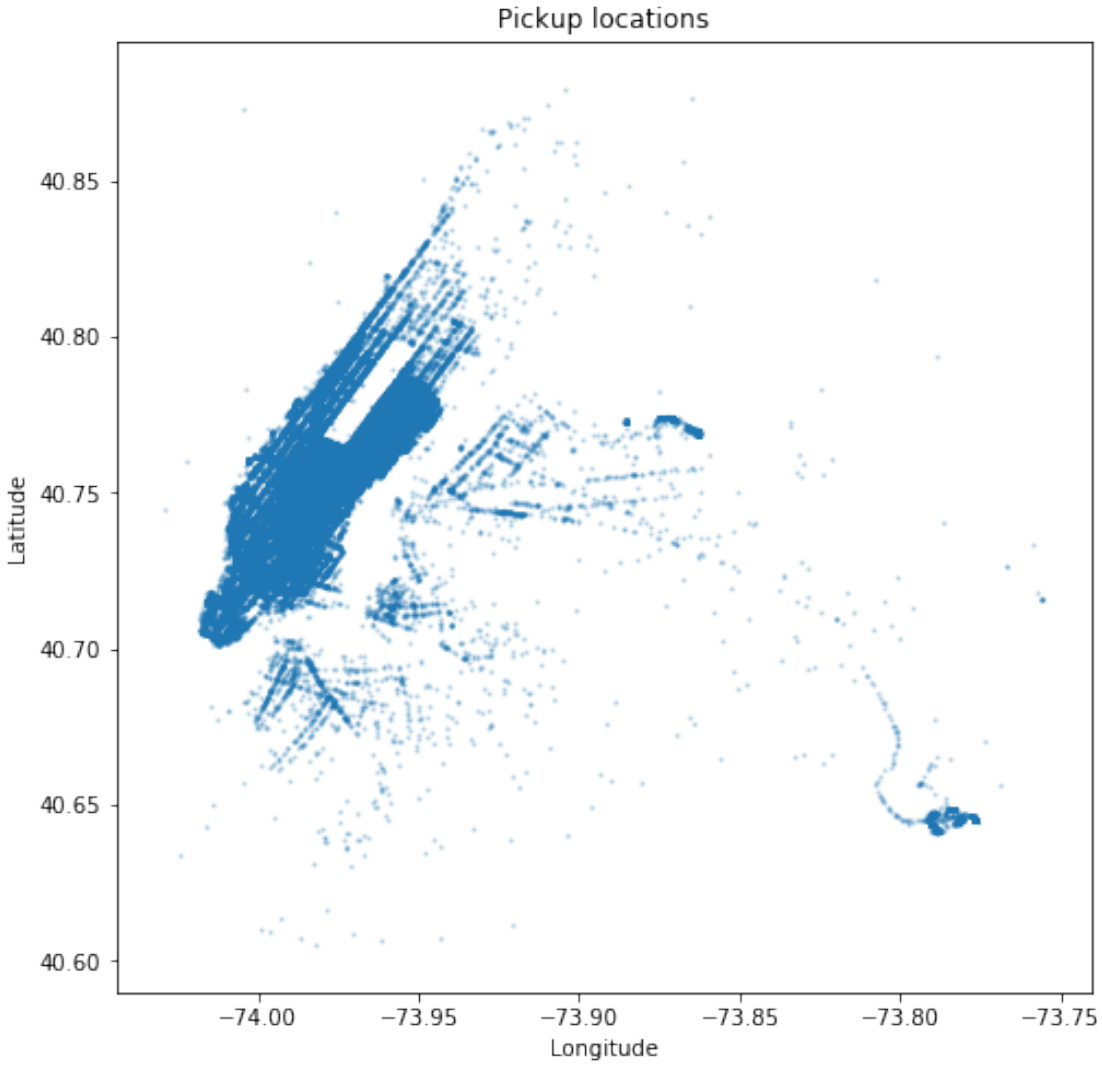
	pickup_datetime	dropoff_datetime	pickup_lon	pickup_lat	\
0	2016-01-30 22:47:32	2016-01-30 23:03:53	-73.988251	40.743542	
1	2016-01-04 04:30:48	2016-01-04 04:36:08	-73.995888	40.760010	
2	2016-01-07 21:52:24	2016-01-07 21:57:23	-73.990440	40.730469	
3	2016-01-01 04:13:41	2016-01-01 04:19:24	-73.944725	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.

[5]:
```
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.

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.

[6]:
```
clean_taxi = all_taxi[all_taxi['passengers'] > 0]
clean_taxi = clean_taxi[clean_taxi['distance'] > 0]
clean_taxi = clean_taxi[clean_taxi['duration'].between(60,3600)]
clean_taxi = clean_taxi[clean_taxi['distance']/clean_taxi['duration'] * 3600 <=␣
↪100]
clean_taxi
```

[6]:

	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	
…	…	…	…	…	
97687	2016-01-31 02:59:16	2016-01-31 03:09:23	-73.997391	40.721027	
97688	2016-01-14 22:48:10	2016-01-14 22:51:27	-73.988037	40.718761	
97689	2016-01-08 04:46:37	2016-01-08 04:50:12	-73.984390	40.754978	
97690	2016-01-31 12:55:54	2016-01-31 13:01:07	-74.008675	40.725979	
97691	2016-01-05 08:28:16	2016-01-05 08:54:04	-73.968086	40.799915	

	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
…	…	…	…	…	…
97687	-73.978447	40.745277	1	2.17	607
97688	-73.983337	40.726162	1	0.60	197
97689	-73.985909	40.751820	4	0.79	215
97690	-74.009598	40.716003	1	0.85	313
97691	-73.972290	40.765533	5	3.30	1548

[96445 rows x 9 columns]

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 the work on a small sample of `clean_taxi` before processing the whole thing. (To check if my code is working, draw a scatter diagram of the (lon, lat) pairs of the result; the scatter diagram should have the shape of Manhattan.)

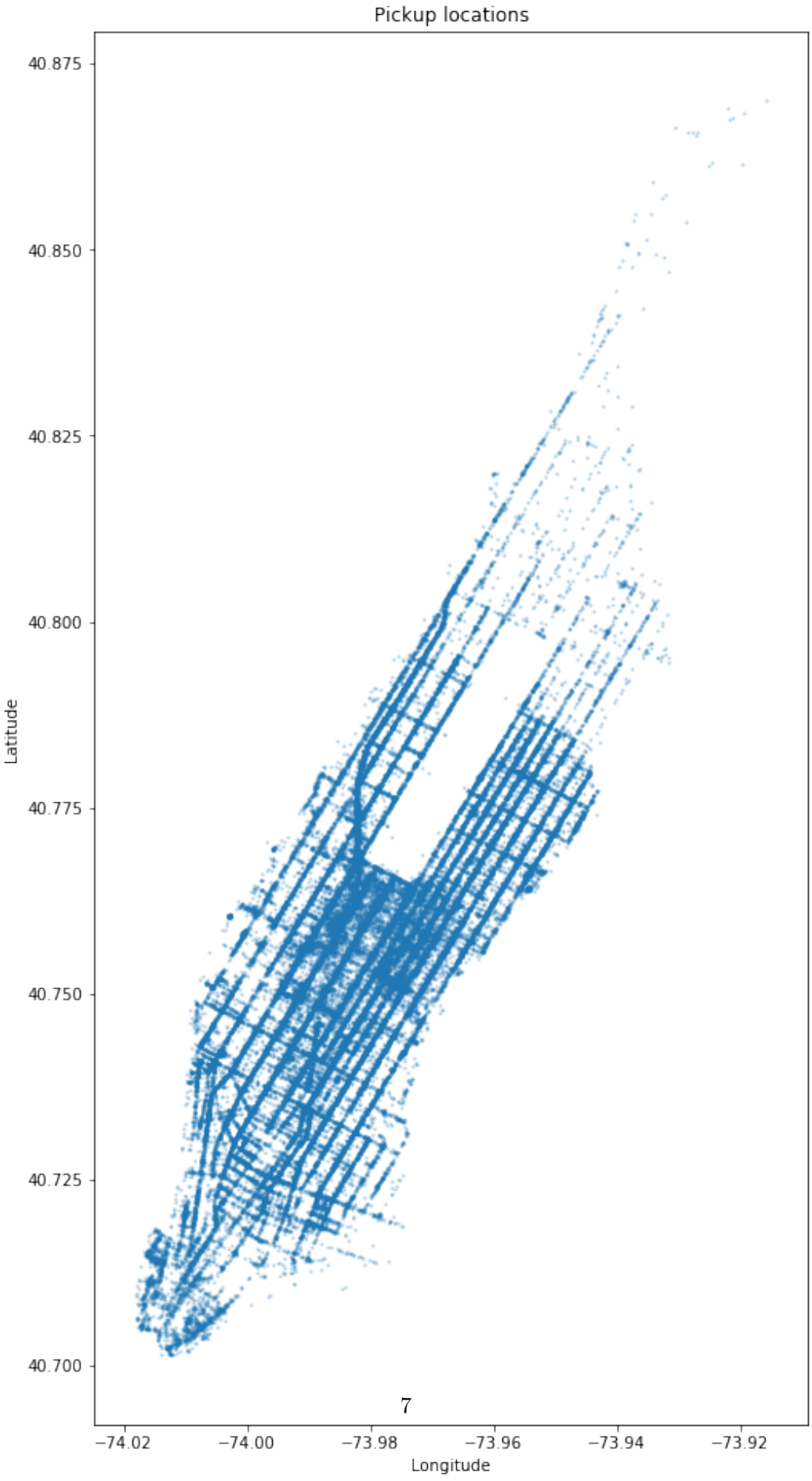
[8]:
```
polygon = pd.read_csv('manhattan.csv')
def in_manhattan(x, y):
    """Whether a longitude-latitude (x, y) pair is in the Manhattan polygon."""
    lat = list(polygon['lat'])
    lon = list(polygon['lon'])

    j = len(lat) - 1
    oddNodes = False

    for i in range(0, j+1):
        if (lon[i] + (y - lat[i]) / (lat[j] - lat[i]) * (lon[j] - lon[i])) < x:
            if ((lat[i] < y <= lat[j]) and (lon[i] <= x)):
                oddNodes = not oddNodes
            elif ((lat[i] < y <= lat[j]) and (lon[j] <= x)):
                oddNodes = not oddNodes
            elif (lat[j] < y <= lat[i]) and (lon[i] <= x):
                oddNodes = not oddNodes
            elif (lat[j] < y <= lat[i]) and (lon[j] <= x):
                oddNodes = not oddNodes
        j = i
    return oddNodes

manhattan_taxi = clean_taxi.copy()
def dropoff(x):
    return in_manhattan(x['dropoff_lon'], x['dropoff_lat'])
manhattan_taxi['drop_off_Manhattan']  = clean_taxi.apply(dropoff, axis =1)
def pickup(x):
    return in_manhattan(x['pickup_lon'], x['pickup_lat'])
manhattan_taxi['pickup_Manhattan'] = clean_taxi.apply(pickup, axis=1)

manhattan_taxi = manhattan_taxi.loc[ (manhattan_taxi['drop_off_Manhattan'] ==␣
↪True) & (manhattan_taxi['pickup_Manhattan'] == True), :]

manhattan_taxi = manhattan_taxi.drop(columns=['drop_off_Manhattan',␣
↪'pickup_Manhattan'])
```

[9]:
```
inMan = []
for x in range(0, len(clean_taxi)):
    inMan.append(in_manhattan(clean_taxi.iloc[x]['pickup_lon'],
                              clean_taxi.iloc[x]['pickup_lat']) and
                              in_manhattan(clean_taxi.iloc[x]['dropoff_lon'],␣
↪clean_taxi.iloc[x]['dropoff_lat']))
manhattan_taxi = clean_taxi.iloc[inMan]
```

[11]:
```
manhattan_taxi = pd.read_csv('manhattan_taxi.csv')
```

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

[12]:
```
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

In the following cell, a summary of the data selection and cleaning performed.

[13]:
```
original_trip = len(all_taxi)
cleaned_trip = len(clean_taxi)
removed_trip = original_trip - cleaned_trip
percentage_removed = removed_trip / original_trip *100
manhattan_trip = len(manhattan_taxi)

print(f"Of the original {original_trip} trips, {removed_trip} anomolous trips␣
 ↪({percentage_removed})"
      "\n"f"were removed through data cleaning, and then the {manhattan_trip}␣
 ↪trips "
      "\n"f" within Manhattan were selected for further analysis")
```

```
Of the original 97692 trips, 1247 anomolous trips (1.276460713262089)
were removed through data cleaning, and then the 82800 trips
 within Manhattan were selected for further analysis
```

1.4 Part 2: Exploratory Data Analysis

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

my 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 Years 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.

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).

[14]:
```
manhattan_taxi['date'] = pd.to_datetime(manhattan_taxi['pickup_datetime']).dt.
 ↪date
manhattan_taxi.head()
```

[14]:

	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	

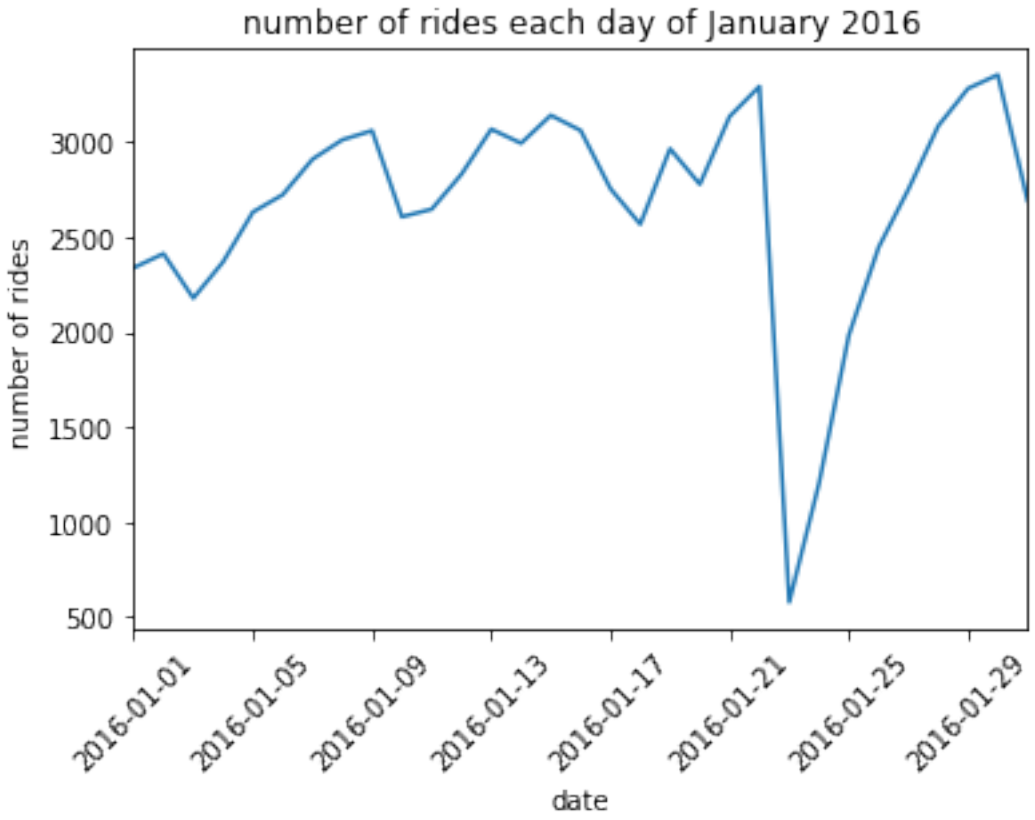
	dropoff_lon	dropoff_lat	passengers	distance	duration	date

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

Create a data visualization that allows 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.

[16]:
```
df = manhattan_taxi.groupby('date').size()
df.plot()
plt.xticks(rotation = 45)
plt.ylabel('number of rides')
plt.title('number of rides each day of January 2016')
```

[16]:
```
Text(0.5, 1.0, 'number of rides each day of January 2016')
```



Finally, I 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.

[17]:
```
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
```

Could do more EDA here but I skipped

1.5 Part 3: Feature Engineering

In this part, I will create a design matrix (i.e., feature matrix) for the linear regression model. I 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.*).

I 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 I 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.

[19]:
```
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)
```

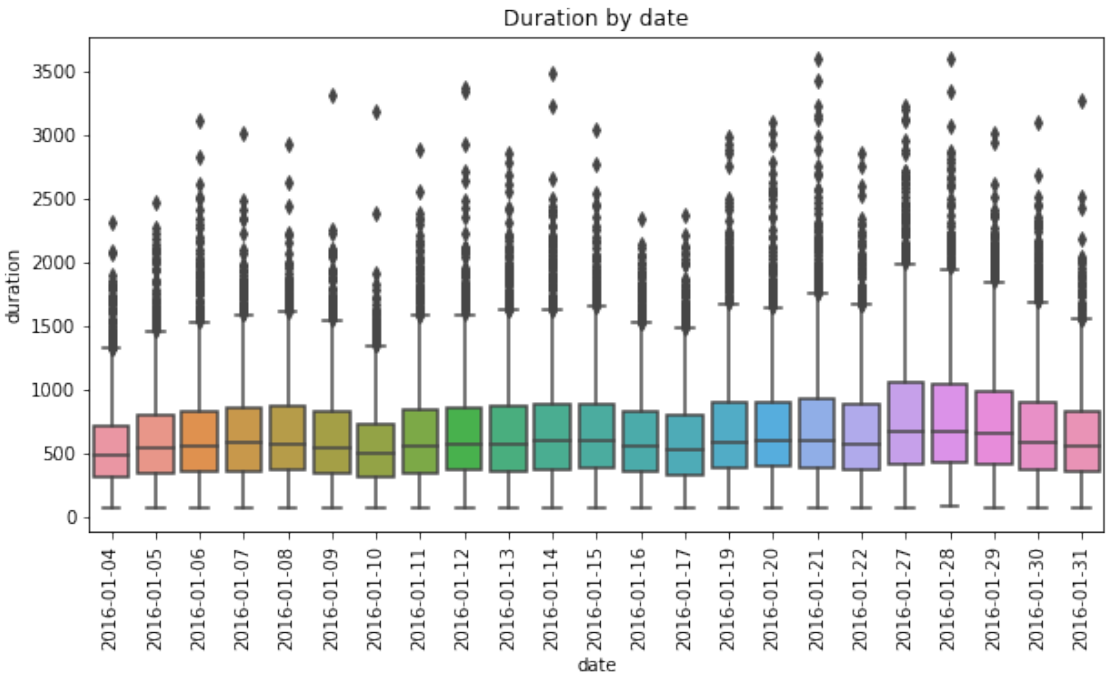
```
Train: (53680, 10) Test: (13421, 10)
```

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.

[20]:
```
plt.figure(figsize=(10, 5))
data = train.sort_values('date')
sns.boxplot('date', 'duration', data=data)
plt.xticks(rotation = 90)
plt.title('Duration by date')
```

[20]:
```
Text(0.5, 1.0, 'Duration by date')
```



In one or two sentences, describe the assocation between the day of the week and the duration of a taxi trip.

For the first two week, the week follows a similar trend of each other. There seems to be shorter duration trips on the weekends than in weekdays. We see the decrease in the duration and it moves from weekday into weekend for each week. In the boxplot, Sunday seems to be the day with lowest duration (given the week starts on Monday). Moreover, duration on Thursdays are relatively higher than other days, range of duration was less during weekends compared to weekdays.

[21]:
```
print('Typical dates:\n')
pat = '  [1-3]|18 | 23| 24|25 |26 '
print(re.sub(pat, '   ', calendar.month(2016, 1)))
```

```
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
```

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.

[22]:
```
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
```

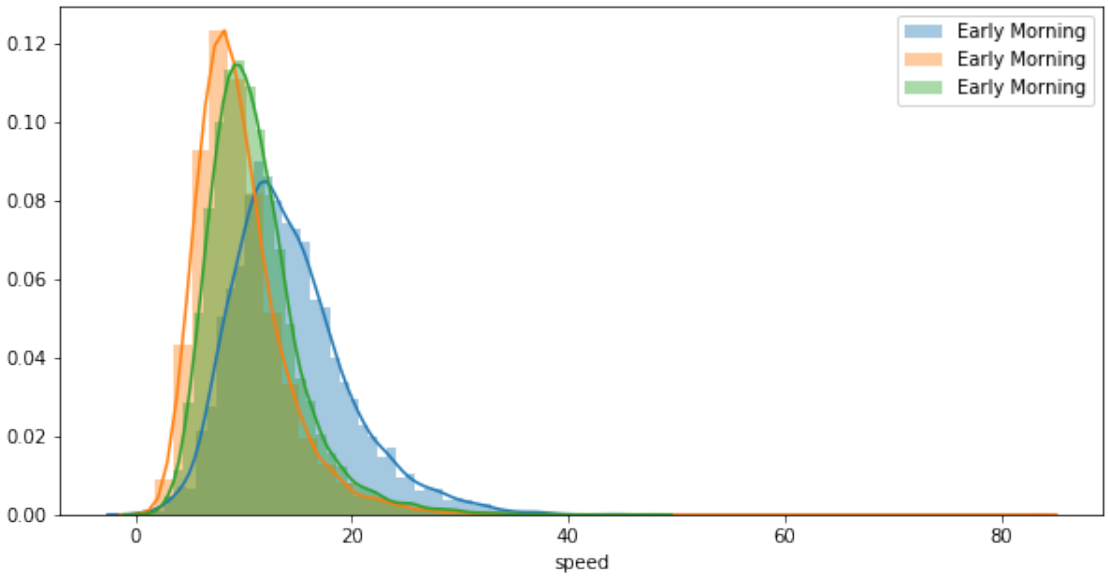
[22]:
```
pickup_datetime     2016-01-21 18:02:20
dropoff_datetime    2016-01-21 18:27:54
pickup_lon                     -73.9942
pickup_lat                       40.751
dropoff_lon                    -73.9637
dropoff_lat                     40.7711
passengers                            1
distance                           2.77
duration                           1534
date                         2016-01-21
hour                                 18
day                                   3
```

```
weekend                              0
period                               3
speed                          6.50065
Name: 14043, dtype: object
```

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).

[23]:
```
plt.figure(figsize=(10,5))
sns.distplot(train.loc[train['period']==1, 'speed'], kde=True, label ='Early␣
↪Morning')
sns.distplot(train.loc[train['period']==2, 'speed'], kde=True, label ='Early␣
↪Morning')
sns.distplot(train.loc[train['period']==3, 'speed'], kde=True, label ='Early␣
↪Morning')
plt.legend()
```

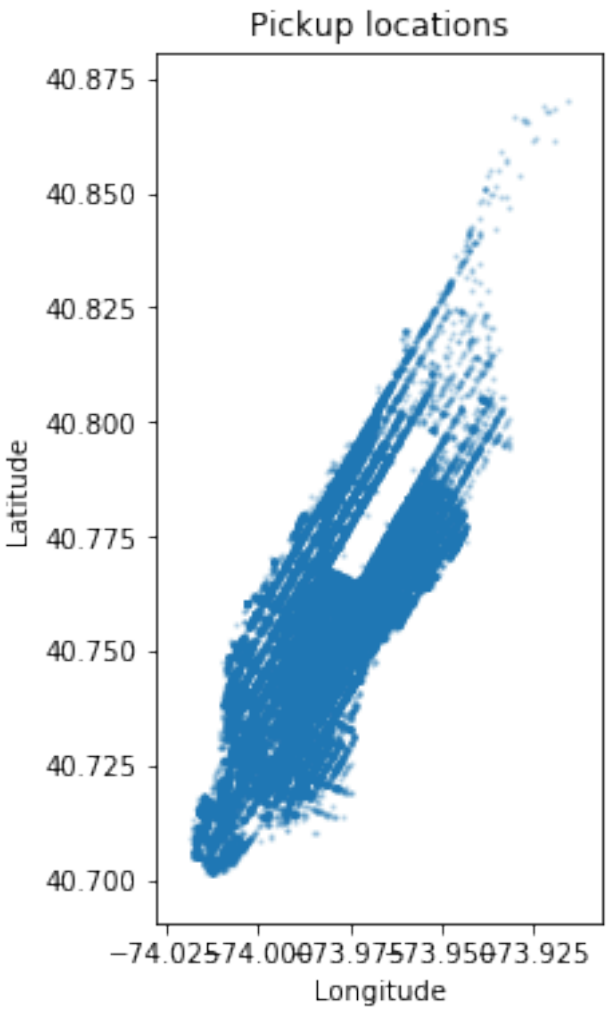
[23]:
```
<matplotlib.legend.Legend at 0x7f7e7f366d30>
```



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

Manhattan can roughly be divided into Lower, Midtown, and Upper regions. Instead of studying a map, I will approximate by finding the first principal component of the pick-up location (latitude and longitude). Before doing that, I will first take a look at a scatterplot of trips in Manhattan:

[24]:
```
plt.figure(figsize=(3, 6))
pickup_scatter(manhattan_taxi)
```



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.

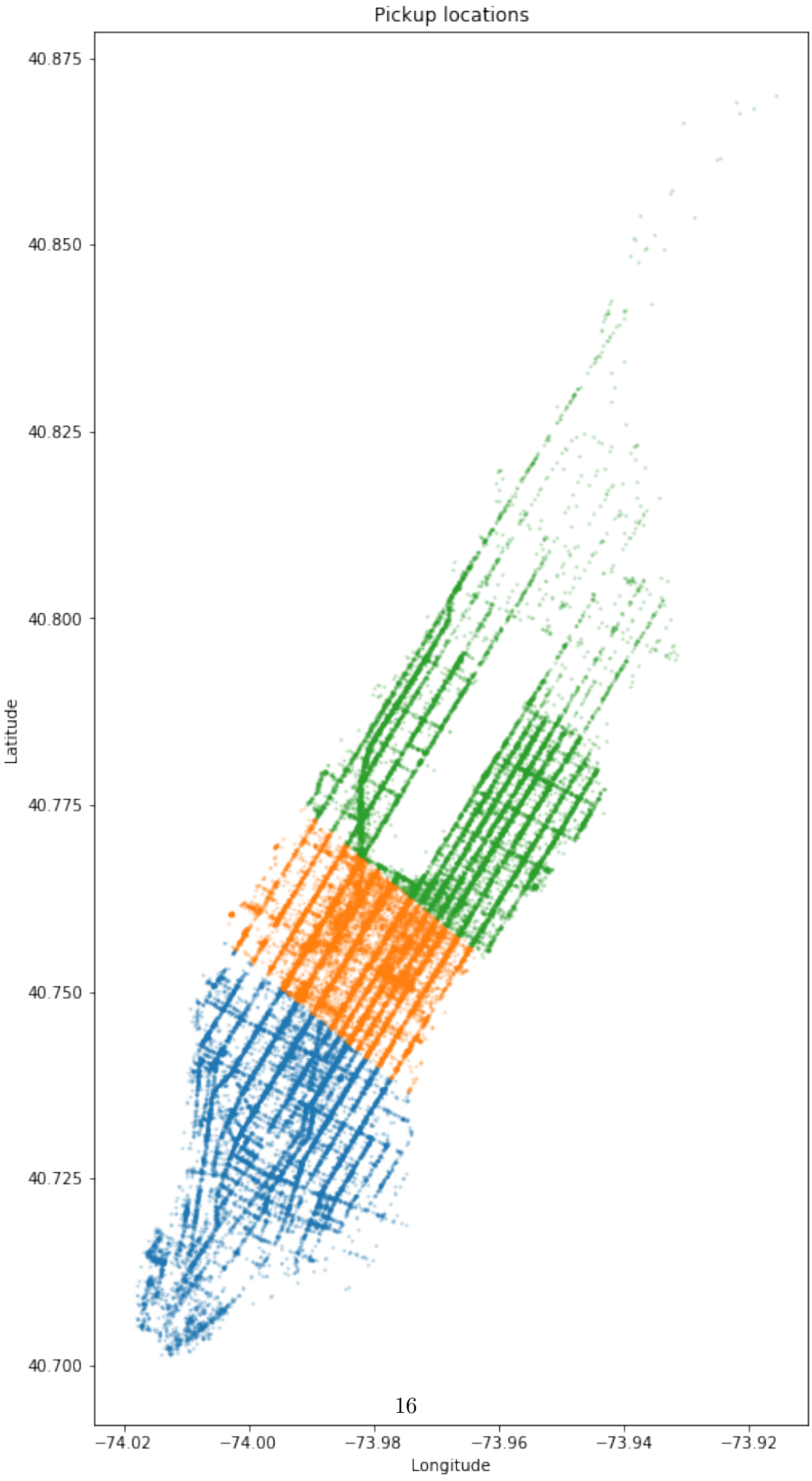
Read the documentation of `pd.qcut`, which categorizes points in a distribution into equal-frequency bins.

[25]:
```
# Find the first principal component
D = train[['pickup_lon', 'pickup_lat']].values
pca_n = D.shape[0]
pca_means = np.mean(D, axis=0)
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']].values
    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@vt.T[:,0]
    t.loc[:,'region'] = pd.qcut(first_pc, 3, labels=[0, 1, 2])

add_region(train)
add_region(test)
```

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!

[27]:
```
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

Finally, I create a design matrix that includes many of these features. Quantitative features are converted to standard units, while categorical features are converted to dummy variables using one-hot encoding. The `period` is not included because it is a linear combination of the `hour`. The `weekend` variable is not included because it is a linear combination of the `day`. The `speed` is not included because it was computed from the `duration`; it's impossible to know the speed without knowing the duration, given that I know the distance.

[29]:
```
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)

design_matrix(train).iloc[0,:]
```

[29]:
```
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
```

1.6 Part 4: Model Selection

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

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.

[30]:
```
def rmse(errors):
    """Return the root mean squared error."""
    return np.sqrt(np.mean(errors ** 2))

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

[30]:
```
399.1437572352666
```

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).

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

model = LinearRegression()
model.fit(pd.DataFrame(train['distance']), train['duration'])
prediction = model.predict(pd.DataFrame(test['distance']))
simple_rmse = rmse(test['duration'] - prediction)
simple_rmse
```

[32]:
```
276.7841105000342
```

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.

[34]:
```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(pd.DataFrame(design_matrix(train)), train['duration'])
prediction = model.predict(pd.DataFrame(design_matrix(test)))
linear_rmse = rmse(test['duration'] - prediction)
linear_rmse
```

[34]:
```
255.19146631882754
```

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']):
    model.fit(design_matrix(train.loc[train['period']==v, :]), train.
 ↪loc[train['period']==v, 'duration'])
    predictions = np.array(model.predict(design_matrix(test.
 ↪loc[test['period']==v, :])))
    e = ((predictions - test.loc[test['period']==v, 'duration']).values).
 ↪tolist()
    errors = errors + e
period_rmse = rmse(np.array(errors))
period_rmse
```

[36]:
```
246.62868831165173
```

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.

In one or two sentences, explain how the `period` regression model could possibly outperform linear regression model, even when the design matrix of the latter includes one feature for each possible hour.

For the period regression model, what we are doing is essentially iterating through each unique values that are in train[‘period’]. This creates multiple regression lines rather than ust fitting the entire data in one plance, hence allowing us better fit the data. This makes the period regression model outperform the linear regression.

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.

[38]:
```
model = LinearRegression()
model.fit(design_matrix(train), train['speed'])
speed_prediction = np.array(model.predict(design_matrix(test)))
speed_rmse = rmse((((test['distance']*3600) /
 ↪speed_prediction)-test['duration']))
speed_rmse
```

[38]:
```
243.01798368514952
```

At this point, think about why predicting speed leads to a more accurate regression model than predicting duration directly.

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.

I 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.

[40]:
```
model = LinearRegression()
choices = ['period', 'region', 'weekend']

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

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

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.loc[v_train[c] == v, :]
            v_test = v_test.loc[v_test[c] == v, :]
            print(v_train.shape, v_test.shape)
        model.fit(design_matrix(v_train), v_train.loc[:, outcome])
        predictions=np.array(model.predict(design_matrix(v_test)))
        e=error_fn(predictions, v_test).tolist()
        errors=errors+e
    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)
```

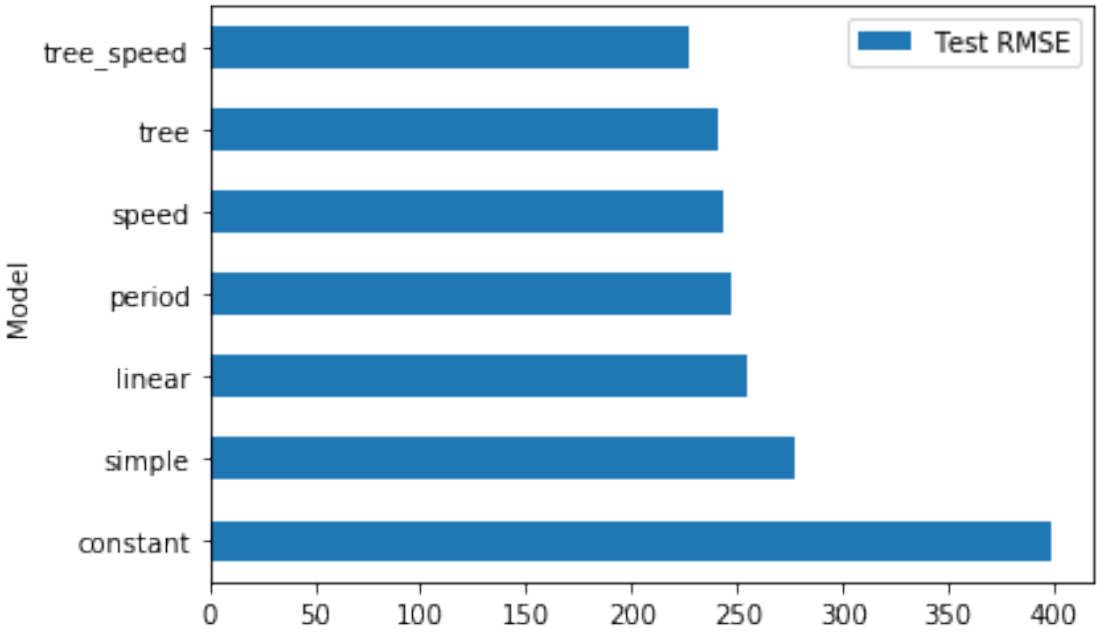
```
(4868, 16) (1264, 16)
(2584, 16) (665, 16)
(980, 16) (250, 16)
(4868, 16) (1264, 16)
(2584, 16) (665, 16)
(1604, 16) (415, 16)
(4868, 16) (1264, 16)
(1450, 16) (373, 16)
(792, 16) (201, 16)
(4868, 16) (1264, 16)
(1450, 16) (373, 16)
(658, 16) (172, 16)
(4868, 16) (1264, 16)
(834, 16) (226, 16)
(453, 16) (121, 16)
(4868, 16) (1264, 16)
(834, 16) (226, 16)
(381, 16) (105, 16)
(30591, 16) (7585, 16)
(8687, 16) (2165, 16)
(6508, 16) (1613, 16)
(30591, 16) (7585, 16)
(8687, 16) (2165, 16)
(2179, 16) (552, 16)
(30591, 16) (7585, 16)
(9927, 16) (2472, 16)
(7728, 16) (1942, 16)
(30591, 16) (7585, 16)
(9927, 16) (2472, 16)
(2199, 16) (530, 16)
(30591, 16) (7585, 16)
(11977, 16) (2948, 16)
(9283, 16) (2258, 16)
```

```
(30591, 16) (7585, 16)
(11977, 16) (2948, 16)
(2694, 16) (690, 16)
(18221, 16) (4572, 16)
(6623, 16) (1644, 16)
(4905, 16) (1224, 16)
(18221, 16) (4572, 16)
(6623, 16) (1644, 16)
(1718, 16) (420, 16)
(18221, 16) (4572, 16)
(6516, 16) (1628, 16)
(5135, 16) (1285, 16)
(18221, 16) (4572, 16)
(6516, 16) (1628, 16)
(1381, 16) (343, 16)
(18221, 16) (4572, 16)
(5082, 16) (1300, 16)
(3900, 16) (1006, 16)
(18221, 16) (4572, 16)
(5082, 16) (1300, 16)
(1182, 16) (294, 16)
(4868, 16) (1264, 16)
(2584, 16) (665, 16)
(980, 16) (250, 16)
(4868, 16) (1264, 16)
(2584, 16) (665, 16)
(1604, 16) (415, 16)
(4868, 16) (1264, 16)
(1450, 16) (373, 16)
(792, 16) (201, 16)
(4868, 16) (1264, 16)
(1450, 16) (373, 16)
(658, 16) (172, 16)
(4868, 16) (1264, 16)
(834, 16) (226, 16)
(453, 16) (121, 16)
(4868, 16) (1264, 16)
(834, 16) (226, 16)
(381, 16) (105, 16)
(30591, 16) (7585, 16)
(8687, 16) (2165, 16)
(6508, 16) (1613, 16)
(30591, 16) (7585, 16)
(8687, 16) (2165, 16)
(2179, 16) (552, 16)
(30591, 16) (7585, 16)
(9927, 16) (2472, 16)
(7728, 16) (1942, 16)
```

```
(30591, 16) (7585, 16)
(9927, 16) (2472, 16)
(2199, 16) (530, 16)
(30591, 16) (7585, 16)
(11977, 16) (2948, 16)
(9283, 16) (2258, 16)
(30591, 16) (7585, 16)
(11977, 16) (2948, 16)
(2694, 16) (690, 16)
(18221, 16) (4572, 16)
(6623, 16) (1644, 16)
(4905, 16) (1224, 16)
(18221, 16) (4572, 16)
(6623, 16) (1644, 16)
(1718, 16) (420, 16)
(18221, 16) (4572, 16)
(6516, 16) (1628, 16)
(5135, 16) (1285, 16)
(18221, 16) (4572, 16)
(6516, 16) (1628, 16)
(1381, 16) (343, 16)
(18221, 16) (4572, 16)
(5082, 16) (1300, 16)
(3900, 16) (1006, 16)
(18221, 16) (4572, 16)
(5082, 16) (1300, 16)
(1182, 16) (294, 16)
Duration: 240.3395219270353
Speed: 226.90793945018314
```

Here's a summary of my 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');
```



1.7 Conclusion

I've carried out the entire data science lifecycle for a challenging regression problem.

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

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

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

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

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.

1.8 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?

[ ]: