

Data Science Lab

Lab 10 solution

As usual, let's begin importing some useful libraries. Note the presence of several visualization customizations. `plt.style.use()` lets us specify a style family among some predefined samples (see other options on the [matplotlib style reference page](#)). Then, the `rcParams` dictionary can be used to change the default style settings and behaviors of matplotlib. In this case, we are increasing the default figure size to 12x6 inches. Finally, setting the `max_rows` attribute for pandas forces the number of rows to be displayed whenever a DataFrame is printed.

```
[1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

plt.style.use('seaborn-notebook')
pd.options.display.max_rows = 10
```

```
[2]: plt.rcParams["figure.figsize"] = (12, 6)
```

0.1 Exercise 1: Data exploration of Point of Interest

0.1.1 Exercise 1.1

Let's load the two files using pandas. The first file contains several information related to Points of Interest (POI) in the New York area. The second one contains the IDs of those POIs located in the New York municipality. To retain only the latter points, we can use the DataFrame index functionalities. The operation is equivalent to the set operator of intersection between the two indices.

```
[3]: ny_pois_ids = np.loadtxt("ny_municipality_pois_id.csv")

# This dictionary maps attributes in the table with correct data types
d_types = {'@type':str, '@lat':float, '@lon':float, 'amenity':str, 'name':str,
           'shop':str, 'public_transport':str, 'highway':str}
all_pois_df = pd.read_csv("pois_all_info", sep='\t', index_col='@id',
                        dtype=d_types)

# Filter only POIS in NY municipality: intersection on indices
pois_df = all_pois_df.loc[ny_pois_ids]
```

```
pois_df
```

```
[3]:
```

	@type	@lat	@lon	amenity	\
@id					
42432939	node	40.814104	-73.949623		NaN
42448838	node	40.761647	-73.949864		NaN
42723103	node	40.852182	-73.772677	ferry_terminal	
42764076	node	40.867164	-73.882158		NaN
42811266	node	40.704807	-73.772734		NaN
...
2553096114	node	40.736006	-73.713202		NaN
2553096138	node	40.736020	-73.713063		NaN
2553096143	node	40.736024	-73.713046		NaN
2553096154	node	40.736030	-73.713089		NaN
6146672068	node	40.735901	-73.713000		NaN

		name	shop	public_transport	highway		
@id							
42432939			NaN	NaN	stop_position	NaN	
42448838			NaN	NaN	stop_position	NaN	
42723103	Hart Island Ferry Terminal		NaN		NaN	NaN	
42764076		Botanical Garden	NaN		stop_position	NaN	
42811266			NaN	NaN	stop_position	NaN	
...	
2553096114			NaN	NaN		NaN	NaN
2553096138			NaN	NaN		NaN	NaN
2553096143			NaN	NaN		NaN	NaN
2553096154			NaN	NaN		NaN	NaN
6146672068			NaN	NaN		NaN	NaN

```
[53550 rows x 8 columns]
```

Remember that pandas offers a small set of methods to quickly inspect any pandas DataFrame. Let's apply some.

```
[4]: pois_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 53550 entries, 42432939 to 6146672068
Data columns (total 8 columns):
@type          53550 non-null object
@lat           53550 non-null float64
@lon           53550 non-null float64
amenity        24712 non-null object
name           30550 non-null object
shop           8717 non-null object
public_transport 8376 non-null object
highway        7559 non-null object
```

```
dtypes: float64(2), object(6)
memory usage: 3.7+ MB
```

We have 8 columns. While `@type`, `@lat`, `@lon` are present in every record, the other ones contain many missing values.

```
[5]: pois_df.describe()
```

```
[5]:
```

	@lat	@lon
count	53550.000000	53550.000000
mean	40.720742	-73.936320
std	0.065739	0.078965
min	40.502422	-74.252791
25%	40.679971	-73.987065
50%	40.721492	-73.955369
75%	40.760094	-73.893479
max	40.913907	-73.700112

The `describe` method instead provides a few statistical characteristics of each column (e.g. mean, standard deviation, min and max values). As you might have expected, it includes only numerical variables in the report.

Additional note: the attribute `@type` can be misleading, it does not contain any type-related information. Indeed, it has a single value in the dataset: we can ignore it.

```
[6]: pois_df['@type'].unique()
```

```
[6]: array(['node'], dtype=object)
```

0.1.2 Exercise 1.2

Another way to count manually the amount of missing values per attribute is by concatenating methods that operate column-wise (the default behavior for many DataFrame operations).

```
[7]: pois_df.isna().sum()
```

```
[7]: @type          0
@lat             0
@lon             0
amenity         28838
name            23000
shop            44833
public_transport 45174
highway         45991
dtype: int64
```

There is something interesting here. It seems that the information is not encoded in a canonical way: the columns “amenity”, “shop”, “public_transport” and “highway” may correspond to certain

categories (let’s put “name” aside for now), each containing several possible values. The high number of missing values might suggest us that each record belongs to one category, leaving empty the other fields. Let’s quickly check that.

```
[8]: from collections import Counter

def get_categories():
    return ['amenity', 'shop', 'public_transport', 'highway']

cats = get_categories()

# Count NaNs per row and inspect their frequencies
check_df = pois_df[cats]
row_nans = check_df.isna().sum(axis=1)
print(Counter(row_nans))
```

Counter({3: 37320, 4: 10208, 2: 6022})

Every row that has a NaN count equal to 3 contains a single non-missing category value. Our assumption was almost correct. Yet, there are rows with two non-empty values. Let’s see some of them.

```
[9]: pois_df[row_nans == 2].head()
```

```
[9]:
```

@id	@type	@lat	@lon	amenity	\
418520887	node	40.636888	-74.076675	cafe	
419363225	node	40.718576	-73.945141	NaN	
419363978	node	40.673832	-74.011733	NaN	
502792663	node	40.743007	-73.825372	NaN	
502793612	node	40.756968	-73.828784	NaN	

@id	name	shop	public_transport	highway
418520887	Everything Goes Book Cafe	books	NaN	NaN
419363225	NaN	NaN	platform	bus_stop
419363978	Dwight Street & Van Dyke Street	NaN	platform	bus_stop
502792663	Main Street & 60th Avenue	NaN	platform	bus_stop
502793612	Main Street & Sanford Avenue	NaN	platform	bus_stop

The couples “cafe” and “books”, and “platform” and “bus_stop” appear together. That seems plausible. So, we can assume that each POI can have one or two categories. Note that we also have 10208 records with no category at all.

0.1.3 Exercise 1.3

We can now carry out a deeper analysis on our four categories. To do so, we can plot the distribution of each type within a given category. For quick charts, one can leverage on pandas itself (that, in

turn, uses matplotlib internally). Series and DataFrame objects expose handy [visualization APIs](#). However, keep in mind that pandas plotting functionalities are limited: for complex tasks and customizations matplotlib or seaborn are better choices.

For the sake of simplicity, we can narrow down the analysis only to the most frequent types. Obviously, this is not a common approach: pruning the visual exploration can be dangerous, useful information can be lost or missed. Focusing on the most frequent data points is typically limited to simplify visualization. In this exercise, we can retain only the top 80% frequent types of each category. Let's define a function that evaluates the percentage share of each type within a category and filters out the ones below a given threshold.

Cumulative sum and percentage filter We will use the method `cumsum` to obtain the percentage share of each type within a category. Let's see briefly how it works on a pandas Series and how it can be used to filter with a percentage threshold.

In our introductory example, we can imagine a bag of colored balls. We represent their occurrences in the bag as a Series.

```
[10]: s = pd.Series([10, 12, 20, 2], index=['red', 'green', 'blue', 'yellow'])
s
```

```
[10]: red      10
      green   12
      blue   20
      yellow   2
      dtype: int64
```

We can now sort in descending order the occurrences and pass to the percentage amount each color accounts for by dividing by the total amount of balls.

```
[11]: so = s.sort_values(ascending=False)
so
```

```
[11]: blue      20
      green   12
      red     10
      yellow   2
      dtype: int64
```

```
[12]: sp = so / so.sum()
sp
```

```
[12]: blue      0.454545
      green   0.272727
      red     0.227273
      yellow   0.045455
      dtype: float64
```

We now can compute the cumulative sum.

```
[13]: spc = sp.cumsum()
      spc
```

```
[13]: blue      0.454545
      green    0.727273
      red      0.954545
      yellow   1.000000
      dtype: float64
```

Finally, we can create a boolean mask for elements above a given percentage threshold (e.g. elements that accounts for the top $p\%$ occurrences). The first item in the mask with a True value will be the first non-frequent item. We can discover its positional index using `argmax`.

```
[14]: mask = spc > .8
      mask
```

```
[14]: blue      False
      green    False
      red      True
      yellow   True
      dtype: bool
```

```
[15]: mask.values.argmax()
```

```
[15]: 2
```

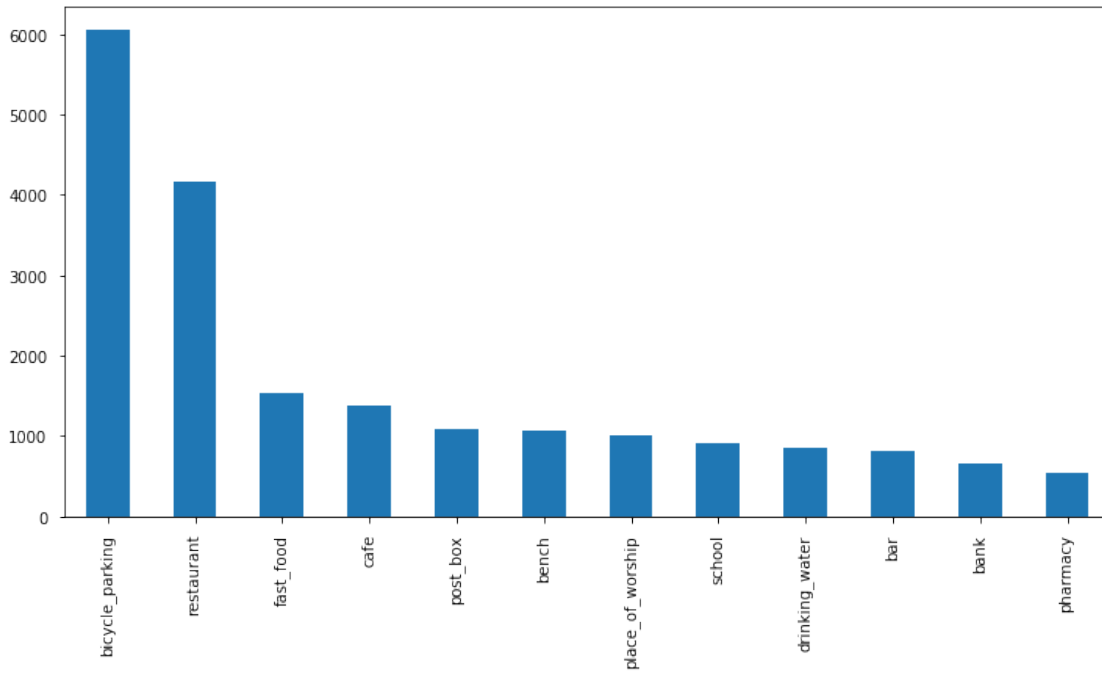
Thus, blue and green balls constitute the maximum subset of balls that does not account more than the 80% in terms of ball occurrences.

Most frequent types in categories

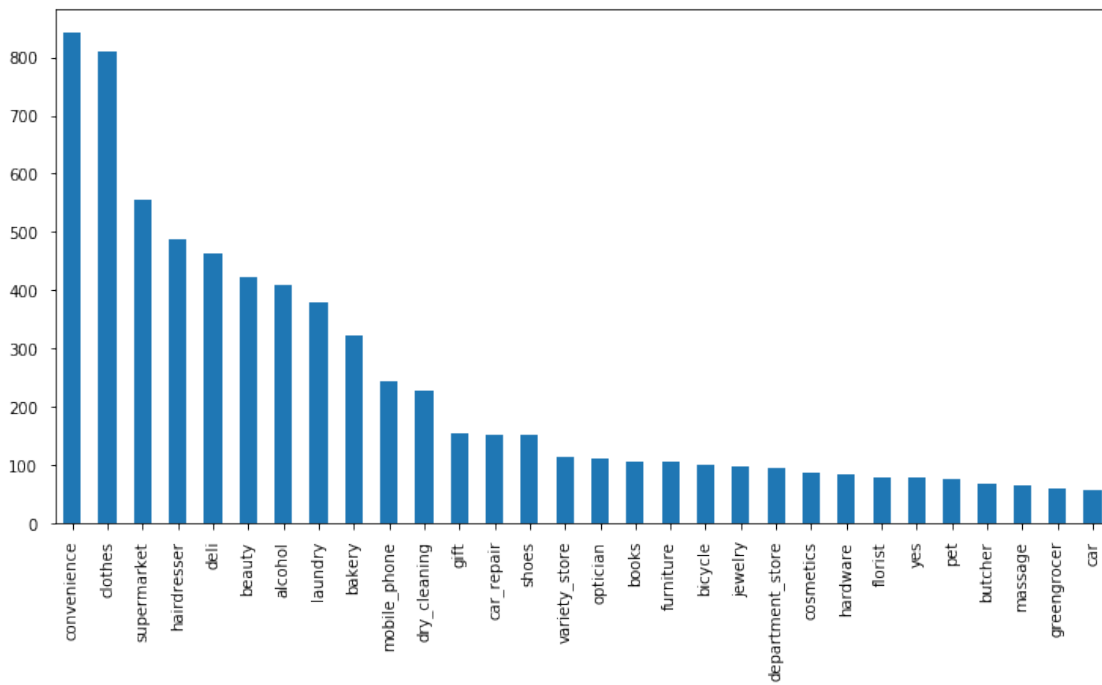
```
[16]: def get_top_perc(series, perc_value=.8):
      perc = series.cumsum() / series.sum()
      arg = (perc > perc_value).values.argmax()
      return series.iloc[:arg+1]

      for col in get_categories():
          p = .8
          valc = pois_df[col].value_counts()
          valf = get_top_perc(valc, p)
          fig, ax = plt.subplots()
          valf.plot(kind='bar', ax=ax)
          ax.set_xticklabels(ax.get_xticklabels(), rotation=90)
          fig.suptitle(f"Top {p*100:.0f}% points in the category: {col}")
```

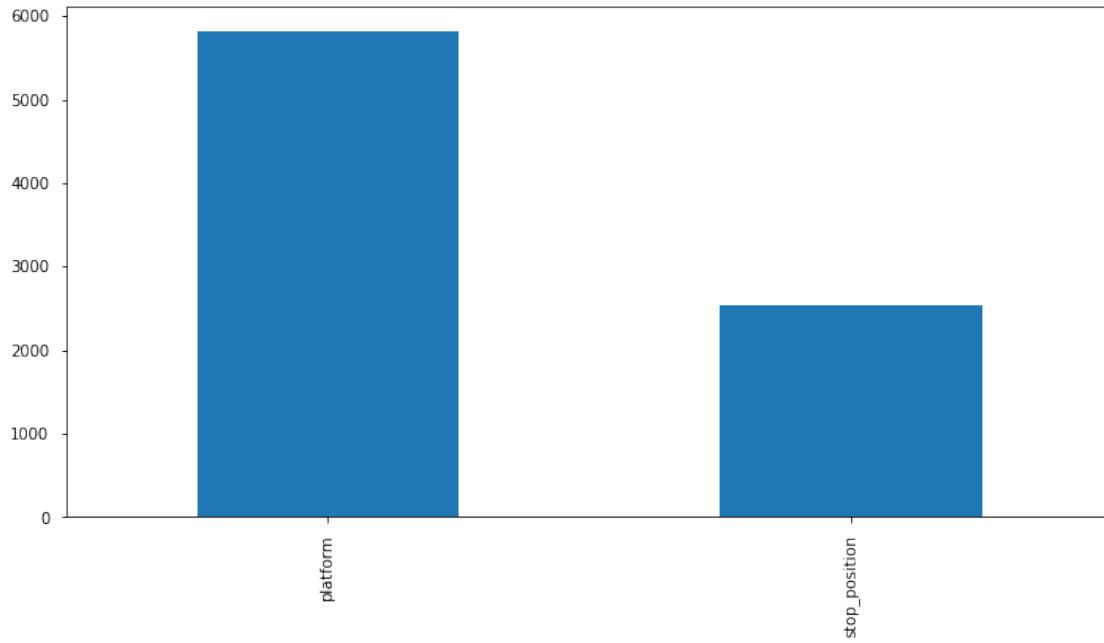
Top 80% points in the category: amenity



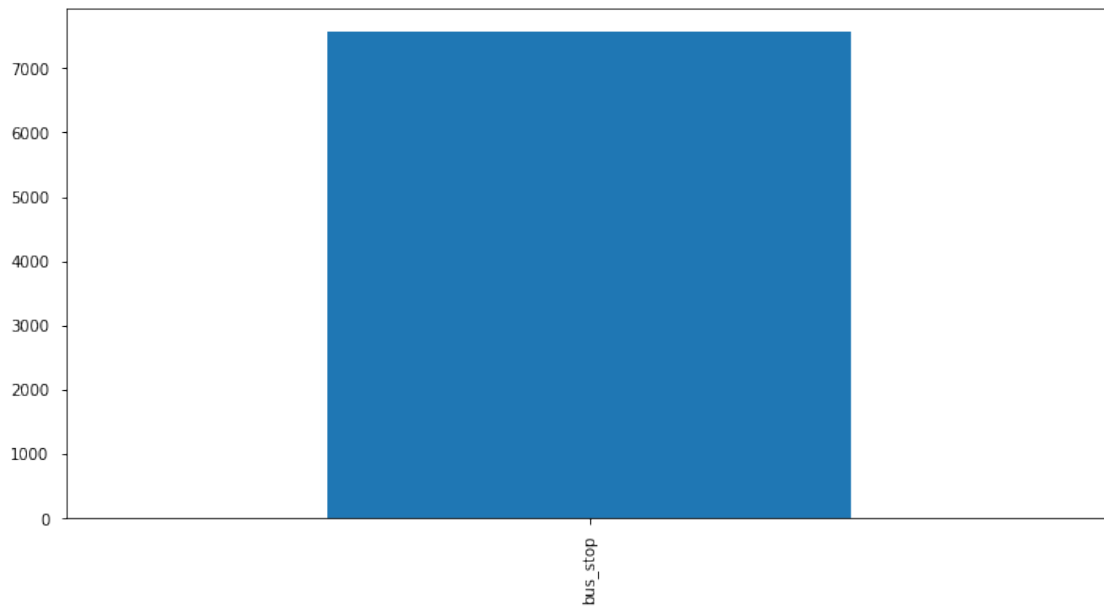
Top 80% points in the category: shop



Top 80% points in the category: public_transport



Top 80% points in the category: highway



From the charts we know that amenity and shop categories have many types (but with quite different occurrences), while public transport and highway contain only a few.

0.1.4 Exercise 1.4

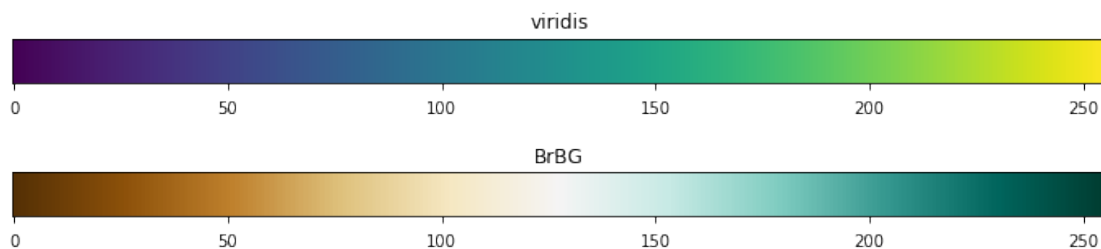
It is time to draw our POIs on a real map. We can define a Map class for the task.

Colormaps We will make use of matplotlib's colormaps. Let's talk about them briefly. By streamlining, every colormap can be seen as a continuous sequence of colors, interpolating between an initial and a final one. The choice of the two edge colors changes the nature of the scale (e.g. sequential, divergent). You can get a colormap object with the `get_cmap`. Let's try with *viridis* and *BrBG*, a Perceptually Uniform Sequential map and a Divergent map respectively.

```
[17]: from matplotlib.cm import get_cmap

x = np.linspace(0, 1, 256)
x = np.vstack((x, x))
fig, axes = plt.subplots(nrows=2, figsize=(12,2))
plt.subplots_adjust(hspace=2) # adjust the vertical space between subplots

for ax, name in zip(axes, ['viridis', 'BrBG']):
    ax.imshow(x, aspect='auto', cmap=plt.get_cmap(name))
    ax.grid(False)
    ax.get_yaxis().set_visible(False)
    ax.set_title(name)
```



As you might have noticed, the values of `x` (i.e. 256 floating point numbers linearly distributed between 0 and 1) are used to pick colors from the colormap.

```
[18]: x[0, :10]
```

```
[18]: array([0.          , 0.00392157, 0.00784314, 0.01176471, 0.01568627,
        0.01960784, 0.02352941, 0.02745098, 0.03137255, 0.03529412])
```

Each color is expressed as four floating point numbers (RGBA notation) between 0 and 1. To get these values, use the callable nature of the colormap object itself.

```
[19]: cmap = plt.get_cmap('viridis')
cmap(x[0, :10])
```

```
[19]: array([[0.267004, 0.004874, 0.329415, 1.      ],
            [0.26851  , 0.009605, 0.335427, 1.      ],
            [0.269944, 0.014625, 0.341379, 1.      ],
            [0.271305, 0.019942, 0.347269, 1.      ],
            [0.272594, 0.025563, 0.353093, 1.      ],
            [0.273809, 0.031497, 0.358853, 1.      ],
            [0.274952, 0.037752, 0.364543, 1.      ],
            [0.276022, 0.044167, 0.370164, 1.      ],
            [0.277018, 0.050344, 0.375715, 1.      ],
            [0.277941, 0.056324, 0.381191, 1.      ]])
```

Map class

```
[20]: import seaborn as sns

class Map:
    def __init__(self, df):
        """ Store Dataset with POIs information. """
        self.pois_df = df
        self.lat_min = df['@lat'].min()
        self.lat_max = df['@lat'].max()
        self.long_min = df['@lon'].min()
        self.long_max = df['@lon'].max()

    def plot_map(self):
        """ Display an image with NY map and return the Axes object. """
        fig, ax = plt.subplots()
        nyc_img = plt.imread('./New_York_City_Map.PNG')
        ax.imshow(nyc_img, zorder=0, extent=[self.long_min,
                                             self.long_max,
                                             self.lat_min,
                                             self.lat_max])

        ax.grid(False)
        return ax

    def plot_pois(self, ax, category, mask):
        """ Plot data on specified Axis. """
        df = self.pois_df.loc[mask]

        # Version 1: using pandas
        types = df[category].unique()
        cmap = get_cmap('viridis')
        colors = cmap(np.linspace(0, 1, types.size))
        for i, t in enumerate(types):
            df_t = df.loc[df[category] == t]
            c = [colors[i]] * df_t.shape[0]
            df_t.plot.scatter(x='@lon', y='@lat', ax=ax, c=c, alpha=.6, label=t)
```

```

    # Version 2: using seaborn
    # sns.scatterplot(df['@lon'], df['@lat'], hue=df[category], ax=ax,
    #                 marker='o', s=3, linewidth=0, palette="viridis",
    → legend='full')

    ax.legend() # show the legend, required by Version 1
    ax.grid(False)
    return ax

```

The class exposes a method that creates a new figure, draws the New York map onto it, and finally returns the Axes object for future plots. Focus on the `extent` parameter: it is used to specify the top, bottom, left and right margins of the image, in terms of coordinates. In this case, it is required to correctly align the background with our latitude and longitude values. Without prior knowledge on that, we can suppose that the image provided with the dataset is bounded to the area that ranges from minimum and maximum values of both latitude and longitude, considering all POIs. We can check if the latter assumption was correct by simply plotting a category POIs.

We would like you to focus on the `plot_pois` method. The actual plotting is presented in two versions: one with pandas functions and one with the seaborn library. The uncommented version exploits pandas. As you can see, the approach requires to: - get the series of distinct types (here we have already filtered the most frequent ones); - get a matplotlib colormap and sample a number of colors out of it equal to the number of different types; - for each type, plot POIs relative to it specifying the color and the label

The commented line (i.e. Version 2) uses the seaborn library to obtain the same result. The `hue` parameter is used to infer the labels and the color of each data point, while `palette` lets you select the desired colormap. Pretty neat, isn't it?

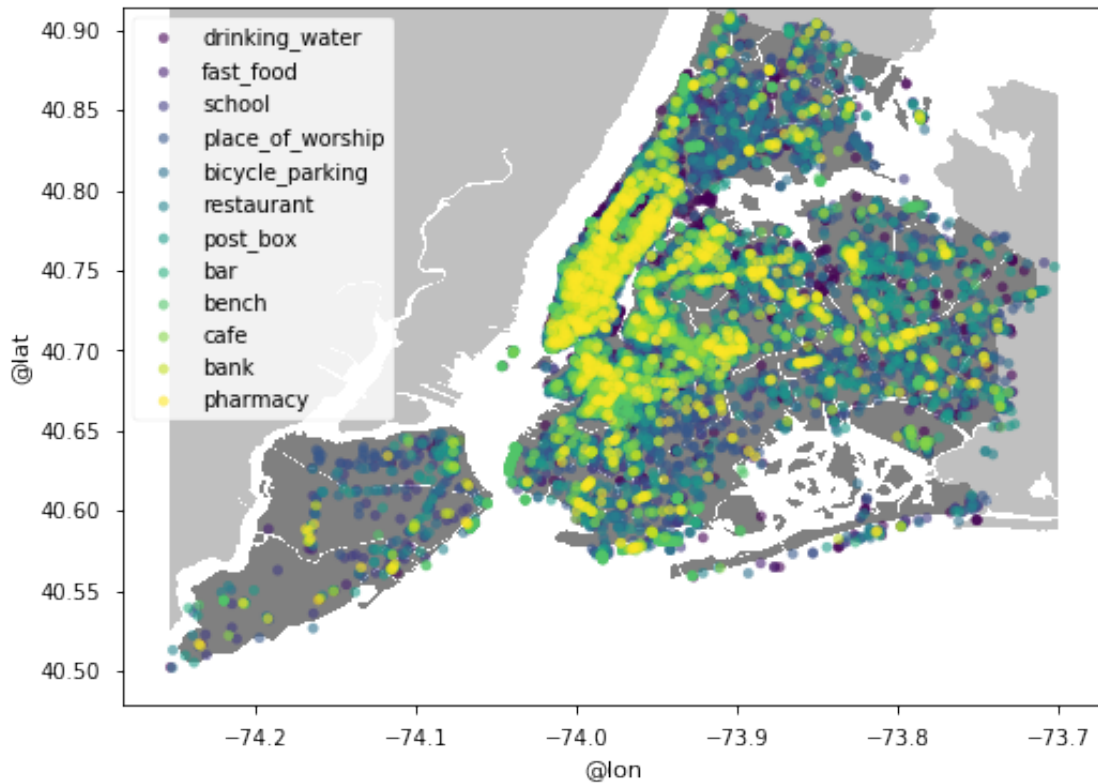
```

[21]: def show_category_on_map(df, column, perc_value):
    """
    Plot the New York map with POIs of a specific category.
    Only the top 'perc_value'% frequent types are plotted.
    """
    counts = df[column].value_counts()
    top_freq = get_top_perc(counts, perc_value)
    ny_map = Map(df)

    ax = ny_map.plot_map()
    mask = df[column].isin(top_freq.index)
    ny_map.plot_pois(ax, column, mask)

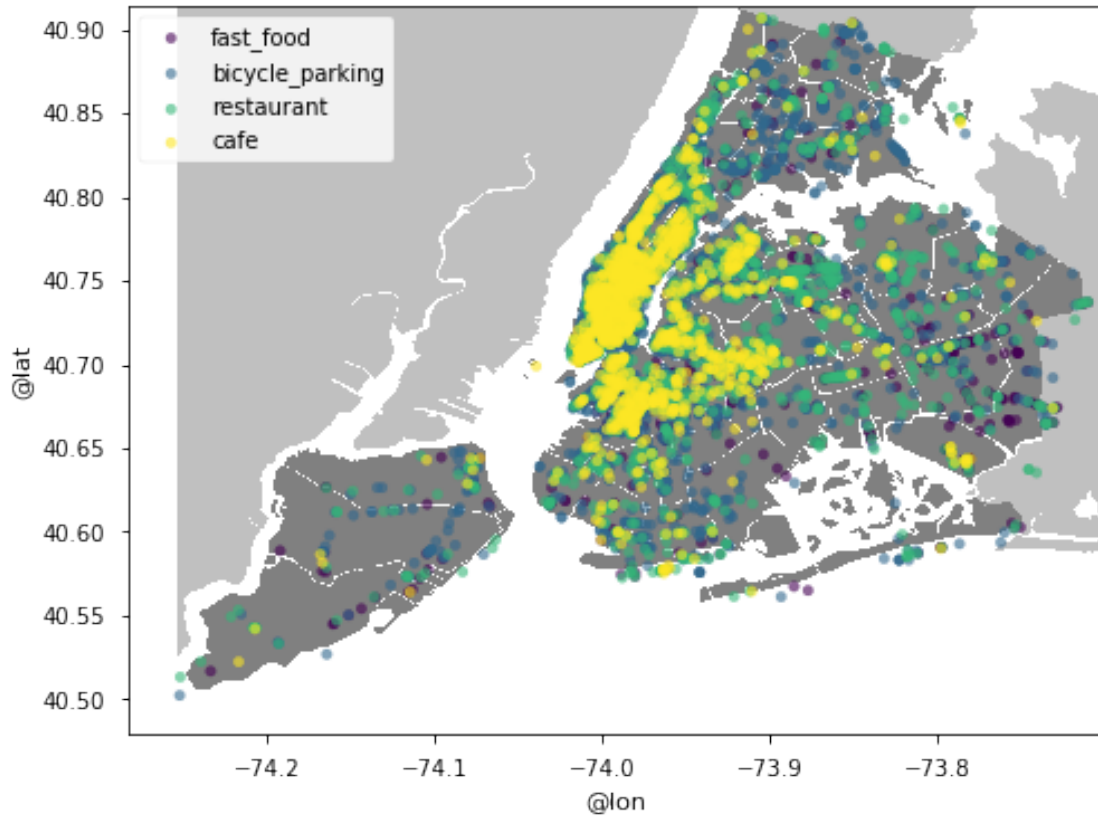
show_category_on_map(pois_df, 'amenity', .8)

```

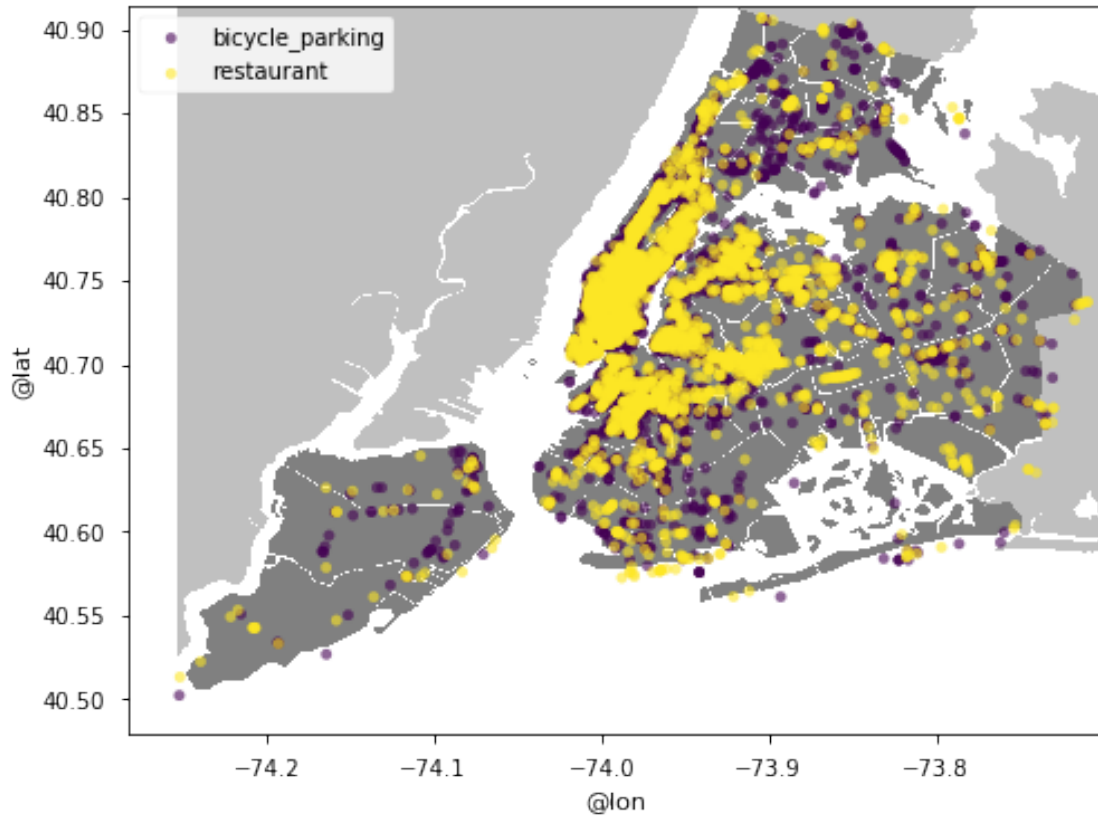


There is an evident concentration of pharmacies in the Manhattan neighborhood. However, we should consider that latest calls may superimpose points on top of the ones drawn by first calls. Let's look at the same chart, lowering the frequency threshold.

```
[22]: show_category_on_map(pois_df, 'amenity', .5)
```



```
[23]: show_category_on_map(pois_df, 'amenity', .3)
```



Manhattan seems to be the core of many of the most frequent types of amenities.

0.1.5 Exercise 1.5

We focus now on a grid-based subdivision system. The rationale here follows the rule of data transformation: we want to explore POIs from another perspective, starting from another representation. Lets see how it can be achieved.

We can decide to subdivide the map into non-overlapping, rectangular-shaped cells based on latitude and longitude. Even this is not the only possibility, a grid like that guarantees that each point will fall into a single cell.

Let's write down a class to map POIs' coordinates to the respective cell (or cell_id).

```
[24]: class Cell_converter:
    def __init__(self, df, n_splits):
        self.lat_min = df['@lat'].min()
        self.lat_max = df['@lat'].max()
        self.long_min = df['@lon'].min()
        self.long_max = df['@lon'].max()
        self.n_splits = n_splits
```

```

def plot_grid(self, ax):
    lat_steps = np.linspace(self.lat_min, self.lat_max, self.n_splits + 1)
    long_steps = np.linspace(self.long_min, self.long_max, self.n_splits + 1)
    ↪1)
    ax.hlines(lat_steps, self.long_min, self.long_max)
    ax.vlines(long_steps, self.lat_min, self.lat_max)

def point_to_cell_coord(self, long, lat):
    x = int((long - self.long_min)/(self.long_max - self.long_min)*self.
    ↪n_splits)
    y = int((lat - self.lat_min)/(self.lat_max - self.lat_min)*self.
    ↪n_splits)
    return x, y

def point_to_cell_id(self, long, lat):
    x, y = self.point_to_cell_coord(long, lat)
    return y * n_splits + x

n_splits = 20
cell_conv = Cell_converter(pois_df, n_splits)

pois_df['cell_id'] = pois_df.apply(lambda x: cell_conv.
    ↪point_to_cell_id(x['@lon'], x['@lat']), axis=1)
pois_df.head()

```

```

[24]:
      @type      @lat      @lon      amenity \
@id
42432939  node  40.814104 -73.949623          NaN
42448838  node  40.761647 -73.949864          NaN
42723103  node  40.852182 -73.772677  ferry_terminal
42764076  node  40.867164 -73.882158          NaN
42811266  node  40.704807 -73.772734          NaN

      name shop public_transport highway  cell_id
@id
42432939          NaN NaN  stop_position  NaN  310
42448838          NaN NaN  stop_position  NaN  250
42723103  Hart Island Ferry Terminal  NaN  NaN  NaN  337
42764076          Botanical Garden  NaN  stop_position  NaN  353
42811266          NaN NaN  stop_position  NaN  197

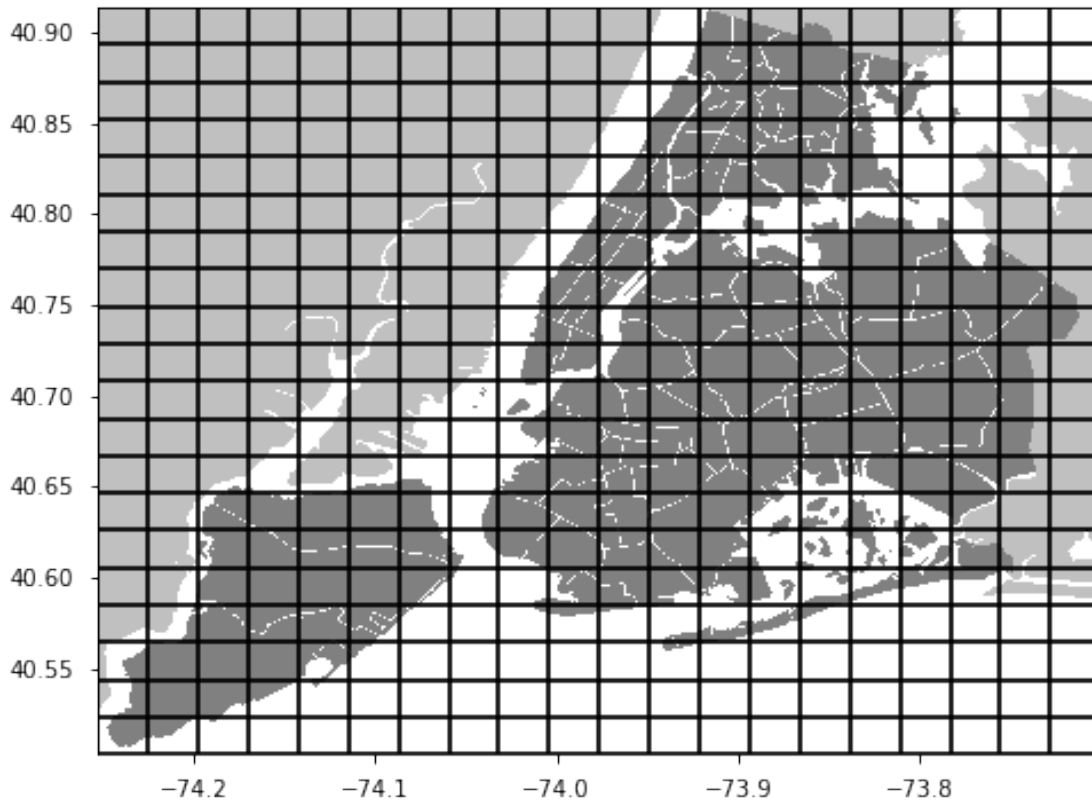
```

The `point_to_cell_id` method creates the final mapping. As you can see, the `cell_id` will be an integer, starting from 0 (the bottom-left cell) and following a row major ordering.

Also, focus on line 27. That is a simply way to assign the result of some operation to a new column, by effectively creating it at the moment.

Bonus As you might have understood, we do like maps, scatter plots, and scatter plots on maps. But we do not dislike grids too. That is the reason for the presence of the `plot_grid` method. Let's apply it.

```
[25]: yet_another_map = Map(pois_df)
ax = yet_another_map.plot_map()
cell_conv.plot_grid(ax)
```



0.1.6 Exercise 1.6

Now that we have the `cell_id` assigned, we are able to carry out a different analysis. The *cell-wise* distribution of POIs re-frames the exploration task, providing a completely new representation of our dataset. The obtained cells (that now represent a specific sub-area in the NY municipality) become atomic units, summarizing part of our data.

As such, we can decide to characterize each cell by counting the POIs contained, for each POI type. The outcome will be a new DataFrame.

```
[26]: def get_df_count(df, column, perc_value):
counts = df[column].value_counts()
top_freq = get_top_perc(counts, perc_value)
```



```

mask = df[column].isin(top_freq.index)
freq_df = df.loc[mask]

# for each cell_id count the number of POIs for each type
count_dframe = []
for cell_id in range(n_splits**2):
    count_vals = freq_df.loc[freq_df['cell_id'] == cell_id][column].
↪value_counts()
    count_vals.name = cell_id
    count_dframe.append(count_vals)

cells_features_df = pd.DataFrame(count_dframe)
cells_features_df = cells_features_df.fillna(0)
return cells_features_df

```

Let's see a few records of the result for frequent amenities.

```
[27]: amenities_df = get_df_count(pois_df, 'amenity', .6)
amenities_df.head()
```

```
[27]:
```

	restaurant	post_box	bicycle_parking	fast_food	cafe	bench
0	2.0	2.0	1.0	1.0	0.0	0.0
1	0.0	0.0	0.0	0.0	1.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0

0.1.7 Exercise 1.7

Now that we have some new features to characterize each cell, we can inspect whether they are correlated or not. This kind of question becomes interesting especially with data from different categories.

Let's begin computing the cell-wise representation for shops as well.

```
[28]: shops_df = get_df_count(pois_df, 'shop', .6)
shops_df.head()
```

```
[28]:
```

	supermarket	convenience	gift	clothes	alcohol	bakery	beauty	laundry	\
0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

	hairdresser	deli	mobile_phone	dry_cleaning
0	0.0	0.0	0.0	0.0

```

1          0.0  0.0          0.0          0.0
2          0.0  0.0          0.0          0.0
3          0.0  0.0          0.0          0.0
4          0.0  0.0          0.0          0.0

```

Now, concatenate it with the DataFrame from amenities. Remember that the *horizontal stacking* operation (i.e. on columns) is obtained with `axis=1`.

```
[29]: final_df = pd.concat([amenities_df, shops_df], axis=1)
```

Pandas provides an handy method to compute the pairwise correlation of columns, excluding null values. We can adopt the resulting matrix to build an heatmap.

```
[30]: final_corr = final_df.corr()
final_corr.head()
```

```
[30]:
```

	restaurant	post_box	bicycle_parking	fast_food	cafe	\
restaurant	1.000000	0.738146	0.933327	0.892643	0.977129	
post_box	0.738146	1.000000	0.757908	0.791383	0.751239	
bicycle_parking	0.933327	0.757908	1.000000	0.823712	0.922268	
fast_food	0.892643	0.791383	0.823712	1.000000	0.931877	
cafe	0.977129	0.751239	0.922268	0.931877	1.000000	

	bench	supermarket	convenience	gift	clothes	\
restaurant	0.325605	0.854280	0.698709	0.748171	0.897885	
post_box	0.464872	0.831700	0.732534	0.572624	0.640424	
bicycle_parking	0.381228	0.858209	0.633704	0.662442	0.839809	
fast_food	0.463410	0.762383	0.707077	0.808271	0.790148	
cafe	0.407526	0.813572	0.694446	0.786491	0.875035	

	alcohol	bakery	beauty	laundry	hairdresser	\
restaurant	0.893299	0.934591	0.641513	0.533456	0.702188	
post_box	0.854463	0.730414	0.744983	0.700626	0.764960	
bicycle_parking	0.873462	0.888446	0.673453	0.522363	0.745809	
fast_food	0.856165	0.759500	0.726217	0.557794	0.758824	
cafe	0.901696	0.874180	0.657689	0.508593	0.732542	

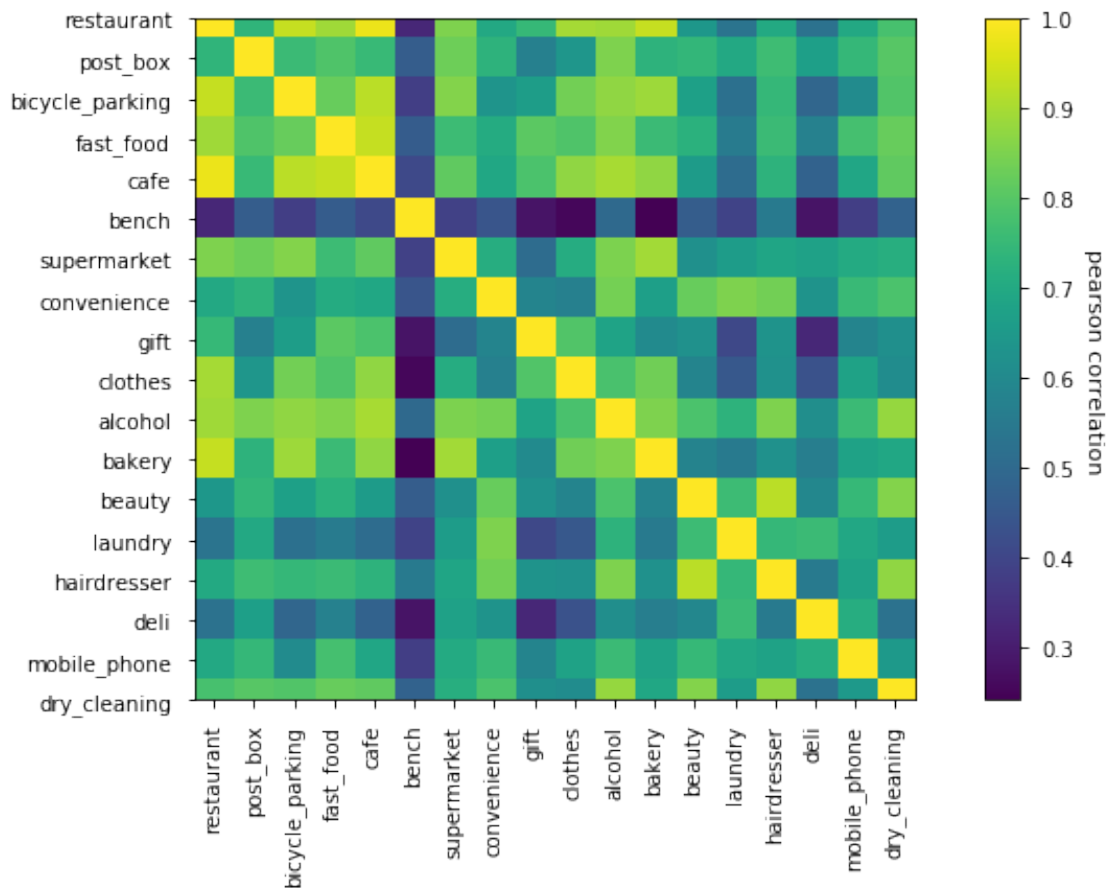
	deli	mobile_phone	dry_cleaning
restaurant	0.527927	0.697886	0.779390
post_box	0.669046	0.747977	0.802623
bicycle_parking	0.490781	0.607225	0.793769
fast_food	0.574192	0.776886	0.822514
cafe	0.480233	0.690048	0.813883

Again, let's develop two versions, to highlight how the seaborn library can significantly simplify your tasks. As you can see, matplotlib does not provide a method *out-of-the-shelf*. You should rely instead on the image show functionality, adjusting axes and adding a colorbar by yourself. Conversely, the seaborn implementation (the commented one), for the same result, would require

a single line of code.

```
[31]: # Version 1
fig, ax = plt.subplots()
im = ax.imshow(final_corr)
ax.set_xticks(np.arange(final_corr.columns.size))
ax.set_yticks(np.arange(final_corr.columns.size))
ax.set_xticklabels(final_corr.columns)
ax.set_yticklabels(final_corr)
plt.setp(ax.get_xticklabels(), rotation=90, ha="right", va="center",
         rotation_mode="anchor") # rotate labels on x-axis
cbar = ax.figure.colorbar(im, ax=ax)
_ = cbar.ax.set_ylabel('pearson correlation', rotation=-90, va="bottom")

# Version 2
# sns.heatmap(final_corr)
```



There are some interesting couples, like restaurant with bicycle parking, fast food, cafe, clothes and bakery. These NY areas are likely related to food activities. Laundry is highly correlated with convenience shops, beauty, hairdresser and dry cleaning. The latter seems to identify cells with

commercial activities and shops.

Bear in mind that these are preliminary results. Cells nature may vary, depending on the parameter `n_steps` used to build the grid. Also, remember that we focused on amenities and shops only and, among them, we filtered out the 40% less frequent POI types. Further analyzes are left to you, as side exercise.

Bonus The main idea behind correlation inspection is to find cells where some activities or shops appear together. We can take the analysis one step further and search for clusters of cells. Given the new representation at our disposal (i.e. cells characterized by the presence of certain POIs), computing clusters of cells would be equivalent to discovering areas of interest in the city.

We left cluster analysis to you, as side exercise. Among all possible research questions, you could focus on:

- which are the POIs that distinguish the obtained clusters the most?
- can you spot areas of interest looking at POIs distribution in each cluster?
- can you spot areas of interest looking at POIs distribution in each cluster?
- are the cells belonging to the same cluster close also geographically?

0.2 Exercise 2: Data exploration and queries on Flight Delay Data

0.2.1 Exercise 2.1

We begin by loading the dataset. The argument `parse_dates` is used to parse a raw date string to a numpy `datetime64` object. Pandas time series / date [functionalities](#) are extremely versatile and should be used anytime a column represents a date or timestamp. Working with datetime types lets you, for example, query specific dates (without using string comparison) or date intervals. That becomes extremely useful when the DataFrame index is of type datetime. The actual class is known as [DatetimeIndex](#).

```
[32]: df = pd.read_csv('data/831394006_T_ONTIME.csv', parse_dates=["FL_DATE"]).  
      ↪ rename(columns=str.lower)
```

0.2.2 Exercise 2.2

Let's call now some descriptive methods.

```
[33]: df.head()
```

```
[33]:
```

	fl_date	unique_carrier	airline_id	tail_num	fl_num	origin_airport_id	\
0	2017-01-01	AA	19805	N787AA	1	12478	
1	2017-01-01	AA	19805	N783AA	2	12892	
2	2017-01-01	AA	19805	N791AA	4	12892	
3	2017-01-01	AA	19805	N391AA	5	11298	
4	2017-01-01	AA	19805	N346AA	6	13830	

```

    origin_airport_seq_id  origin_city_market_id  origin      origin_city_name  \
0          1247803          31703      JFK          New York, NY
1          1289204          32575      LAX          Los Angeles, CA
2          1289204          32575      LAX          Los Angeles, CA
3          1129804          30194      DFW      Dallas/Fort Worth, TX
4          1383002          33830      OGG          Kahului, HI

...  arr_time  arr_delay  cancelled  cancellation_code  carrier_delay  \
0 ...    1209.0     27.0         0.0             NaN             27.0
1 ...    1809.0     42.0         0.0             NaN             34.0
2 ...    2040.0     42.0         0.0             NaN             7.0
3 ...    1749.0     97.0         0.0             NaN             77.0
4 ...     642.0     42.0         0.0             NaN             0.0

    weather_delay  nas_delay  security_delay  late_aircraft_delay  unnamed: 32
0          0.0         0.0         0.0         0.0             NaN
1          0.0         8.0         0.0         0.0             NaN
2          0.0         0.0         0.0        35.0             NaN
3          0.0        20.0         0.0         0.0             NaN
4          0.0        42.0         0.0         0.0             NaN

```

[5 rows x 33 columns]

```
[34]: df.describe()
```

```

[34]:
      airline_id      fl_num  origin_airport_id  origin_airport_seq_id  \
count  450017.000000  450017.000000  450017.000000  4.500170e+05
mean   19900.483275   2079.643193   12698.267568   1.269830e+06
std     385.381448   1722.700045   1534.326936   1.534324e+05
min    19393.000000     1.000000   10135.000000   1.013503e+06
25%    19690.000000     679.000000   11292.000000   1.129202e+06
50%    19805.000000   1602.000000   12892.000000   1.289204e+06
75%    20304.000000   3034.000000   14057.000000   1.405702e+06
max    21171.000000   7439.000000   16218.000000   1.621801e+06

      origin_city_market_id  dest_airport_id  dest_airport_seq_id  \
count          450017.000000  450017.000000  4.500170e+05
mean           31738.603264   12698.024626   1.269805e+06
std            1286.063689   1534.246611   1.534244e+05
min            30070.000000   10135.000000   1.013503e+06
25%            30647.000000   11292.000000   1.129202e+06
50%            31454.000000   12892.000000   1.289204e+06
75%            32467.000000   14057.000000   1.405702e+06
max            35991.000000   16218.000000   1.621801e+06

      dest_city_market_id  crs_dep_time      dep_time  ...  crs_arr_time  \

```

count	450017.000000	450017.000000	441476.000000	...	450017.000000
mean	31738.404640	1327.757407	1333.947392	...	1506.278285
std	1285.791081	481.548320	494.957196	...	501.853851
min	30070.000000	5.000000	1.000000	...	1.000000
25%	30647.000000	917.000000	921.000000	...	1120.000000
50%	31454.000000	1321.000000	1330.000000	...	1529.000000
75%	32467.000000	1730.000000	1738.000000	...	1921.000000
max	35991.000000	2359.000000	2400.000000	...	2359.000000

	arr_time	arr_delay	cancelled	carrier_delay	\
count	440746.000000	439645.000000	450017.000000	97699.000000	
mean	1487.332958	6.954484	0.019746	20.308836	
std	523.821795	49.595992	0.139126	60.649157	
min	1.000000	-98.000000	0.000000	0.000000	
25%	1108.000000	-15.000000	0.000000	0.000000	
50%	1521.000000	-5.000000	0.000000	0.000000	
75%	1921.000000	11.000000	0.000000	17.000000	
max	2400.000000	1944.000000	1.000000	1934.000000	

	weather_delay	nas_delay	security_delay	late_aircraft_delay	\
count	97699.000000	97699.000000	97699.000000	97699.000000	
mean	3.368857	15.742495	0.079745	24.947492	
std	27.809756	32.328181	2.896287	47.850860	
min	0.000000	0.000000	0.000000	0.000000	
25%	0.000000	0.000000	0.000000	0.000000	
50%	0.000000	4.000000	0.000000	3.000000	
75%	0.000000	20.000000	0.000000	31.000000	
max	1934.000000	1457.000000	653.000000	1392.000000	

```

unnamed: 32
count      0.0
mean      NaN
std       NaN
min       NaN
25%      NaN
50%      NaN
75%      NaN
max       NaN

```

[8 rows x 25 columns]

```
[35]: df.info()
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 450017 entries, 0 to 450016
Data columns (total 33 columns):
fl_date      450017 non-null datetime64[ns]

```

```

unique_carrier      450017 non-null object
airline_id         450017 non-null int64
tail_num           449378 non-null object
fl_num             450017 non-null int64
origin_airport_id  450017 non-null int64
origin_airport_seq_id 450017 non-null int64
origin_city_market_id 450017 non-null int64
origin             450017 non-null object
origin_city_name   450017 non-null object
dest_airport_id    450017 non-null int64
dest_airport_seq_id 450017 non-null int64
dest_city_market_id 450017 non-null int64
dest               450017 non-null object
dest_city_name     450017 non-null object
crs_dep_time       450017 non-null int64
dep_time           441476 non-null float64
dep_delay           441476 non-null float64
taxi_out            441244 non-null float64
wheels_off          441244 non-null float64
wheels_on           440746 non-null float64
taxi_in             440746 non-null float64
crs_arr_time       450017 non-null int64
arr_time            440746 non-null float64
arr_delay           439645 non-null float64
cancelled           450017 non-null float64
cancellation_code  8886 non-null object
carrier_delay       97699 non-null float64
weather_delay       97699 non-null float64
nas_delay           97699 non-null float64
security_delay      97699 non-null float64
late_aircraft_delay 97699 non-null float64
unnamed: 32        0 non-null float64
dtypes: datetime64[ns](1), float64(15), int64(10), object(7)
memory usage: 113.3+ MB

```

The dataset is sufficiently large (450k entries, 33 columns and 113 MB of memory footprint). We can easily note that the attribute *unnamed: 32* is null everywhere. Even if it is not strictly necessary, let's get rid of it.

```
[36]: df = df.drop("unnamed: 32", axis=1)
```

Let's list all available carries airports and count them.

```
[37]: df.unique_carrier.unique(), df.unique_carrier.unique().size
```

```
[37]: (array(['AA', 'B6', 'EV', 'HA', 'NK', 'OO', 'UA', 'VX', 'AS', 'WN', 'DL',
           'F9'], dtype=object), 12)
```

```
[38]: distinct_airports = pd.concat([df["origin"], df["dest"]])
print(f"First ten airports: {distinct_airports.unique()[:10]}. Size:␣
↳{distinct_airports.unique().size}")
```

```
First ten airports: ['JFK' 'LAX' 'DFW' 'OGG' 'HNL' 'SFO' 'ORD' 'MIA' 'IAH'
'BOS']. Size: 298
```

Note how we have accessed the DataFrame columns. The first one uses an object's attribute: pandas creates these attributes, matching the column names, at instantiation time. In the latter cell we used the square brackets. That syntax is more verbose but lets you specify a list of columns to slice on, returning a DataFrame as result.

```
[39]: df[["origin", "dest"]]
```

```
[39]:      origin dest
0      JFK  LAX
1      LAX  JFK
2      LAX  JFK
3      DFW  HNL
4      OGG  DFW
...
450012  FLL  MSP
450013  MSP  FLL
450014  ATL  PHL
450015  FLL  ATL
450016  FLL  LGA
```

```
[450017 rows x 2 columns]
```

Since the `fl_date` is now a `datetime64` column, we can ask for something like:

```
[40]: df.fl_date.min(), df.fl_date.max()
```

```
[40]: (Timestamp('2017-01-01 00:00:00'), Timestamp('2017-01-31 00:00:00'))
```

0.2.3 Exercise 2.3

We can filter out canceled flights for the following analysis.

```
[41]: print('Shape before:', df.shape)
df = df.loc[df.cancelled == 0]
print('Shape after:', df.shape)
```

```
Shape before: (450017, 32)
```

```
Shape after: (441131, 32)
```


0.2.4 Exercise 2.4

Using the groupby method we can obtain the two information.

```
[42]: df_by_carrier = df.groupby('unique_carrier')
df_count = df_by_carrier['fl_date'].count()
df_count
```

```
[42]: unique_carrier
AA      72152
AS      14454
B6      24077
DL      69031
EV      33878
...
NK      12129
OO      48266
UA      42171
VX       5645
WN     105472
Name: fl_date, Length: 12, dtype: int64
```

```
[43]: df_by_carrier[['carrier_delay',
                    'weather_delay',
                    'nas_delay',
                    'security_delay',
                    'late_aircraft_delay']].mean()
```

```
[43]:
```

unique_carrier	carrier_delay	weather_delay	nas_delay	security_delay	\
AA	18.736410	2.352168	15.370026	0.178156	
AS	11.736505	3.820850	18.615047	0.169028	
B6	20.297641	1.436562	15.223725	0.312820	
DL	30.858959	9.572160	16.836252	0.033901	
EV	36.329407	1.368504	15.794439	0.000000	
...	
NK	9.080825	0.732499	43.840041	0.035171	
OO	23.495640	4.633053	15.186582	0.100759	
UA	21.436417	2.568226	21.186162	0.016759	
VX	9.308965	3.342583	25.093917	0.041622	
WN	11.737663	2.176463	9.701419	0.031890	

unique_carrier	late_aircraft_delay
AA	18.742267
AS	19.189946
B6	29.282627

DL	21.964020
EV	31.725109
...	...
NK	12.137978
OO	31.316244
UA	24.592458
VX	24.715582
WN	25.658966

[12 rows x 5 columns]

0.2.5 Exercise 2.5

We have already seen how a new column can be generated. Since our `fl_date` is in datetime format, obtaining the day of the week is straightforward.

```
[44]: df['weekday'] = df['fl_date'].dt.dayofweek
df['weekday'].head()
```

```
[44]: 0    6
      1    6
      2    6
      3    6
      4    6
      Name: weekday, dtype: int64
```

Then, a simple element-wise difference between columns is enough for `delaydelta`.

```
[45]: df['deltadelay'] = df['arr_delay'] - df['dep_delay']
df['deltadelay'].head()
```

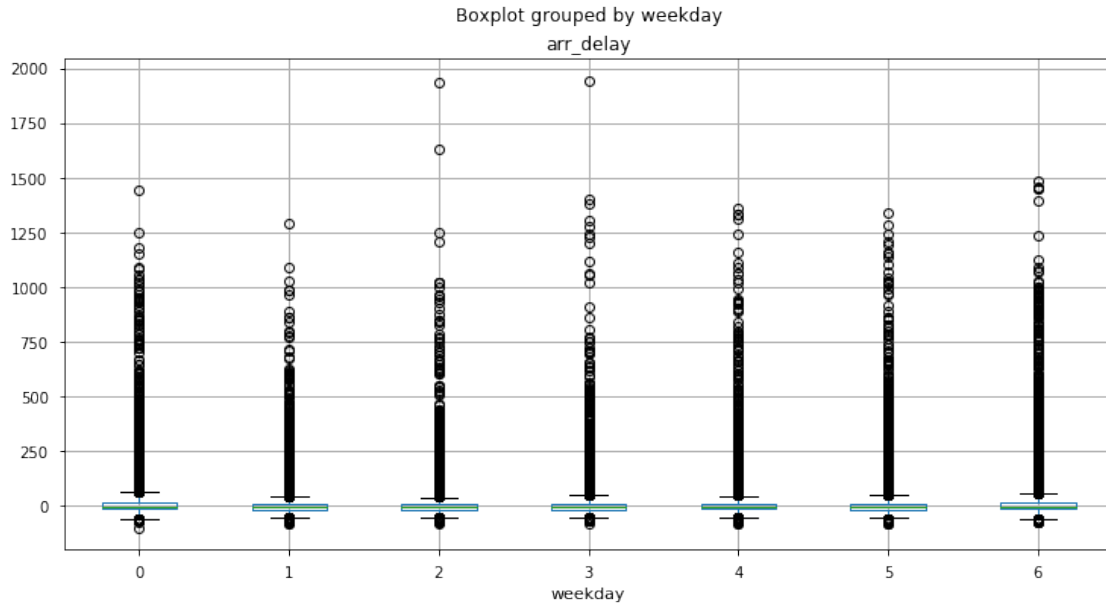
```
[45]: 0    -4.0
      1     8.0
      2    -9.0
      3    20.0
      4    42.0
      Name: delatdelay, dtype: float64
```

0.2.6 Exercise 2.6

To accomplish the task we can use a boxplot. The boxplot is a method for graphically depicting numerical sets thorough their quartiles.

```
[46]: df.boxplot(by='weekday', column='arr_delay')
```

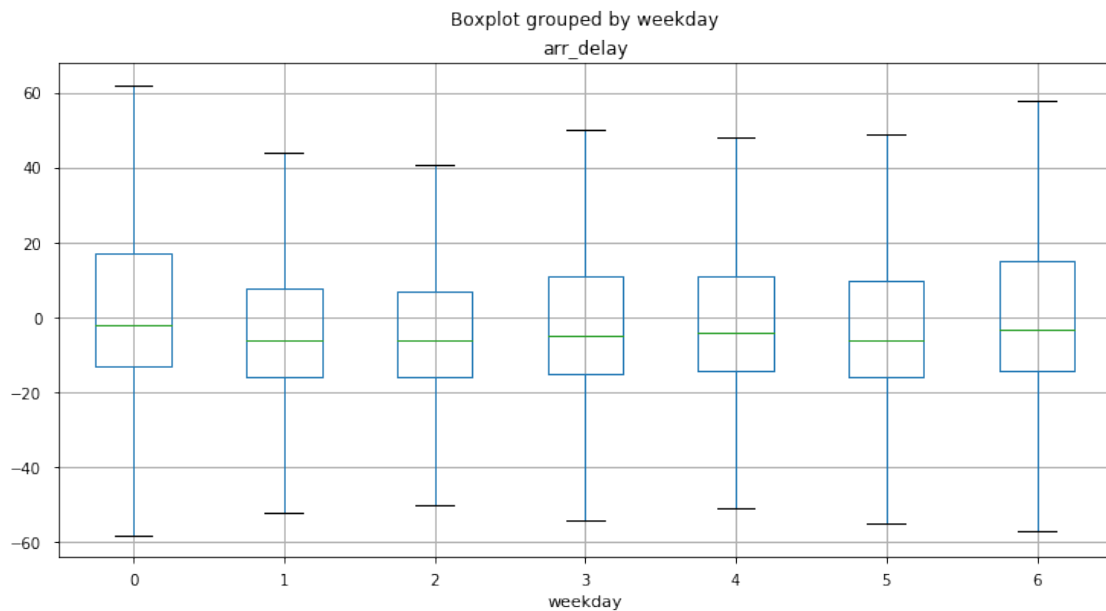
```
[46]: <matplotlib.axes._subplots.AxesSubplot at 0x1a2b260e80>
```



Several outliers squeeze down each plot. Let's exclude them.

```
[47]: df.boxplot(by='weekday', column='arr_delay', showfliers=False)
```

```
[47]: <matplotlib.axes._subplots.AxesSubplot at 0x11cd91320>
```



In both the cases, there is no clear correlation between the arrival delay and the day of the week.

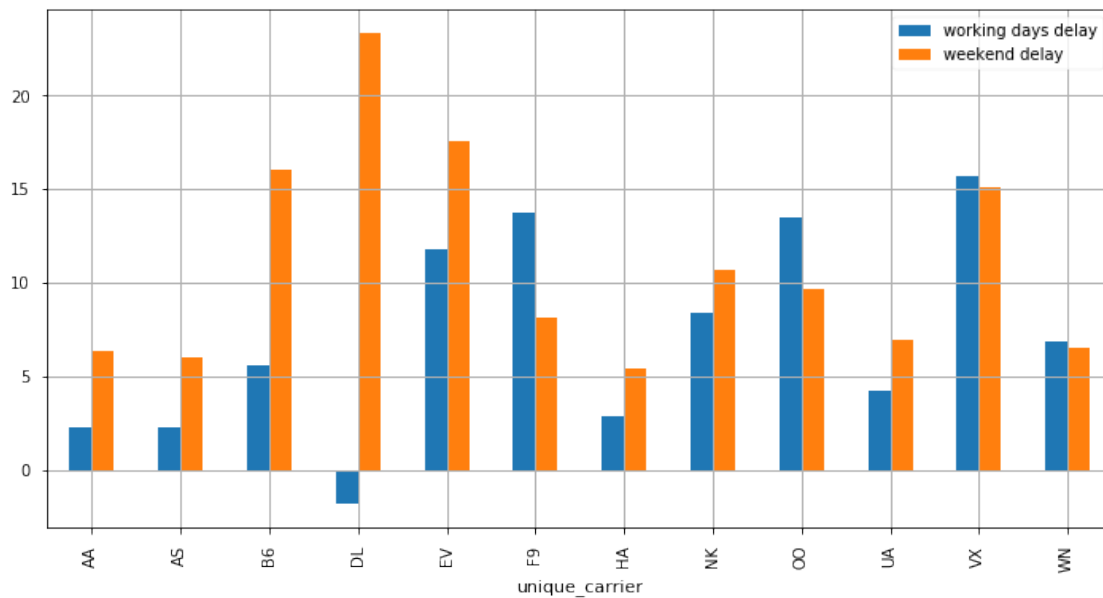
0.2.7 Exercise 2.7

We can now exploit a bar chart.

```
[48]: we_delay = df.loc[df.weekday > 4].groupby('unique_carrier').arr_delay.mean()
      wd_delay = df.loc[df.weekday <= 4].groupby('unique_carrier').arr_delay.mean()

      we_delay.name = "weekend delay"
      wd_delay.name = "working days delay"
```

```
[49]: ax = pd.concat([wd_delay, we_delay], axis=1).plot.bar()
      ax.grid(True)
```



The most uneven behavior is obtained by Delta Airlines. Considering the working days, it has a negative mean delay, meaning that, on average, it landed its airplanes earlier than the scheduled time. Conversely, during weekends, it has the highest mean arrival delay.

0.2.8 Exercise 2.8

The creation of the multi-index on an existing DataFrame is achieved by setting multiple columns to the index itself.

```
[50]: multi_df = df.set_index(['unique_carrier', 'origin', 'dest', 'fl_date']).
      ↪sort_index()
      multi_df[multi_df.columns[:4]].head()
```

```
[50]:
```

unique_carrier	origin	dest	fl_date	airline_id	tail_num	fl_num	\
AA	ABQ	DFW	2017-01-01	19805	N4XSAA	1282	
			2017-01-01	19805	N3NRAA	2611	
			2017-01-01	19805	N4WNAA	2402	
			2017-01-02	19805	N4XKAA	2611	
			2017-01-02	19805	N4XBAA	2402	


```
origin_airport_id
```

unique_carrier	origin	dest	fl_date	origin_airport_id
AA	ABQ	DFW	2017-01-01	10140
			2017-01-01	10140
			2017-01-01	10140
			2017-01-02	10140
			2017-01-02	10140

0.2.9 Exercise 2.9

Working with multi-level indices is extremely useful sometimes. To access specific rows, you can specify an indexing procedure for each level.

```
[51]: multi_df.loc(['AA', 'DL'], ['LAX']), ['dep_time', 'dep_delay']
```

```
[51]:
```

unique_carrier	origin	dest	fl_date	dep_time	dep_delay
AA	LAX	ATL	2017-01-01	1051.0	16.0
			2017-01-01	1747.0	137.0
			2017-01-02	1548.0	18.0
			2017-01-02	2230.0	40.0
			2017-01-02	1055.0	20.0
...			
DL	LAX	TPA	2017-01-26	1146.0	1.0
			2017-01-27	1137.0	-3.0
			2017-01-29	1149.0	4.0
			2017-01-30	1142.0	-3.0
			2017-01-31	1206.0	21.0

```
[5437 rows x 2 columns]
```

Note that the first element passed to `.loc` is a tuple, specifying a fancy indexing access on the first two levels of the index.

0.2.10 Exercise 2.10

Let's break down the problem. We first detect all the interested records. To do so, we can exploit the datetime type to filter on dates.

```
[52]: # fw_df = multi_df.loc[:, :, 'LAX', '2017-01-01':'2017-01-08'], :] # not allowed
fw_df = multi_df.loc[(slice(None), slice(None), 'LAX', slice('2017-01-01', '2017-01-08')), :]
```

The first line of the previous cell is commented out, since the character `:` cannot be used to access index levels. If you want to use it anyway, pandas provides the `IndexSlice` object that handles the translation for you. The previous result would be obtained via:

```
[53]: fw_df = multi_df.loc[pd.IndexSlice[:, :, 'LAX', '2017-01-01':'2017-01-08'], :]
```

We can now complete the task using again the `groupby` operator.

```
[54]: fw_df.groupby('fl_num')['arr_delay'].mean()
```

```
[54]: fl_num
1      0.000000
2     60.000000
4     11.625000
5      2.875000
7     16.750000
...
6344  47.000000
6354  60.000000
6522  49.666667
6563  -8.000000
6710  10.000000
Name: arr_delay, Length: 1105, dtype: float64
```

0.2.11 Exercise 2.11

It is time to explore the use of pandas pivot table. We want to count the flights departed each day of the week, for each carrier. Thus, the aggregation function would be `count`.

```
[55]: cfd = pd.pivot_table(df, values='fl_num', index='unique_carrier', columns='weekday', aggfunc='count')
cfd
```

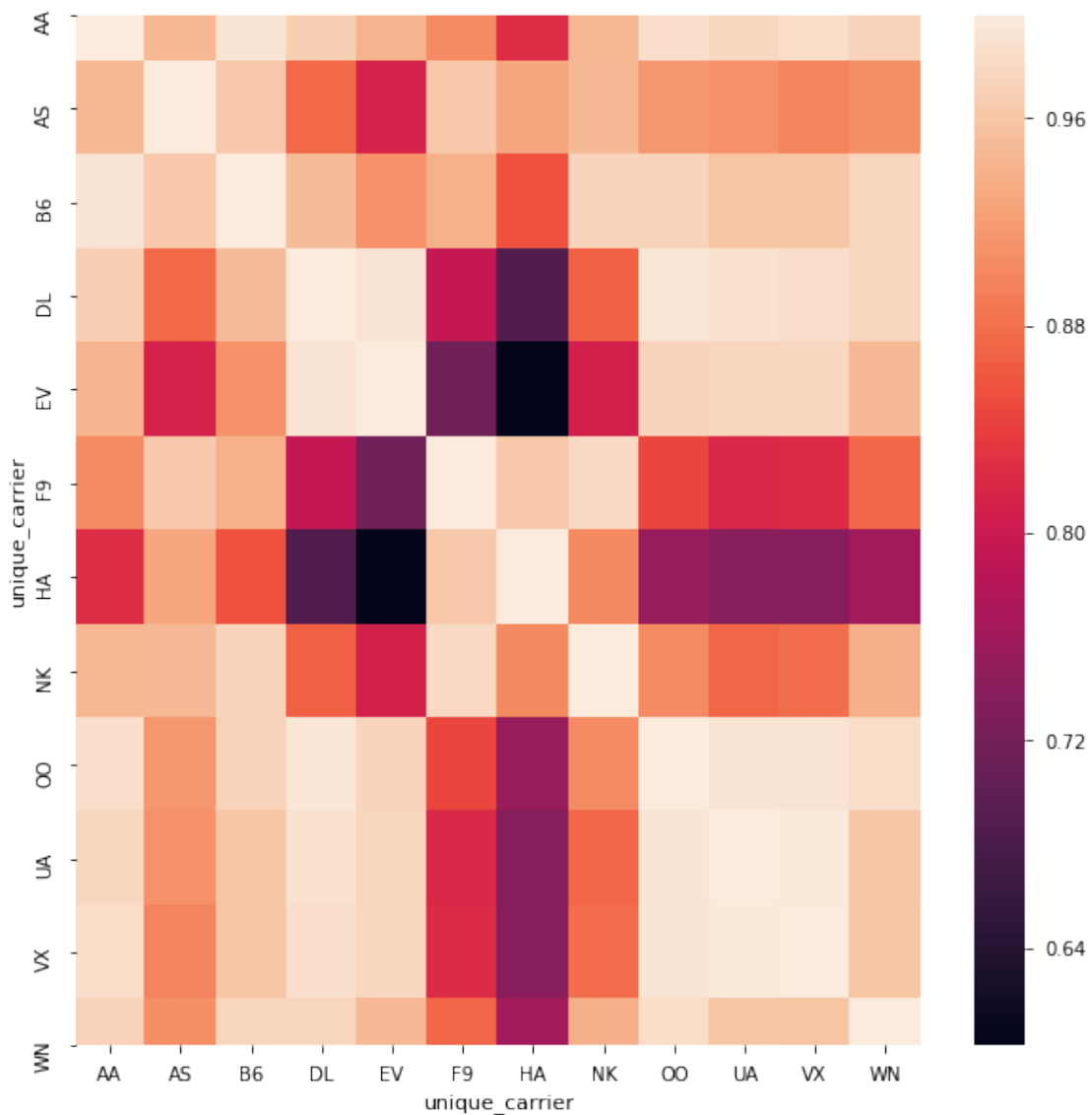
```
[55]: weekday      0      1      2      3      4      5      6
unique_carrier
AA      12035  11457  9651  9854  9820  7905  11430
AS       2440   2261  1806  1935  1953  1762   2297
B6       4063   3942  3143  3258  3169  2643  3859
DL      12157  11512  9248  9695  9521  6492  10406
EV       5935   5649  4627  4923  4774  2905  5065
...
NK       1954   1970  1616  1610  1563  1470  1946
```

OO	8321	7912	6413	6631	6667	4913	7409
UA	7498	6883	5384	5956	5920	3964	6566
VX	986	914	741	791	792	522	899
WN	17913	17855	14260	14220	14012	11168	16044

[12 rows x 7 columns]

Then, to compute the pairwise correlation between the carriers on different days we can count again on the pandas `corr()` method. However, it is applied to columns, hence we first need to transpose the DataFrame. The result is directly passed to seaborn for the heatmap visualization.

```
[56]: plt.figure(figsize=(10,10))
      _ = sns.heatmap(cfd.T.corr())
```



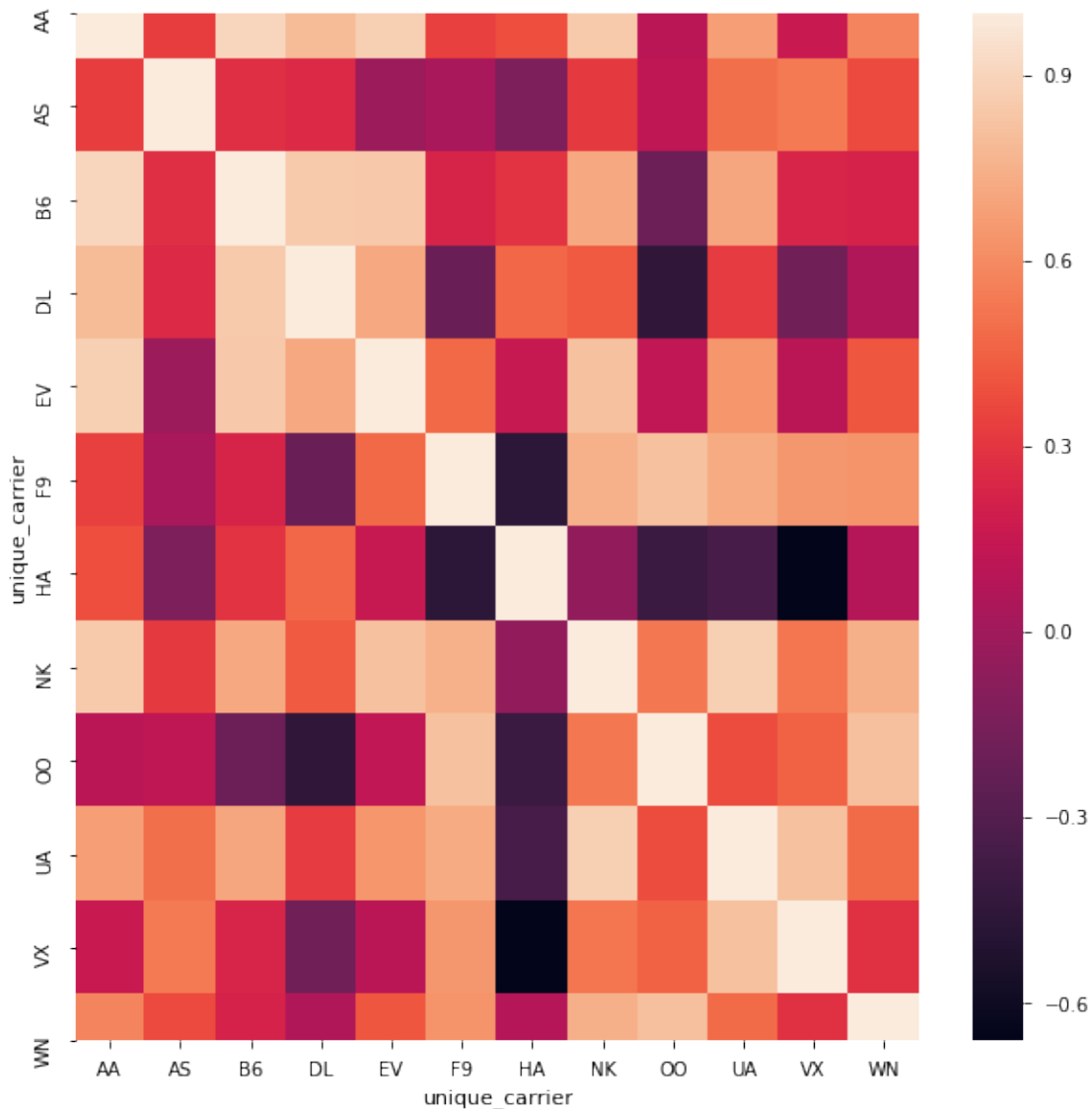
The matrix represents a degree of correlation along the week (the [Pearson correlation](#) is used by default): correlated carriers have operated the same number of flights in the same day of the week.

It seems that Hawaiian Airlines (HA) has a different flight schedule compared to most of the other companies. Conversely, Delta Airlines instead shares the plan with many companies.

0.2.12 Exercise 2.12

Just as before, we need a pandas pivot table.

```
[57]: plt.figure(figsize=(10,10))
_ = sns.heatmap(pd.pivot_table(df, values='arr_delay', index='unique_carrier',
↪columns='weekday', aggfunc='mean').T.corr())
```



Now, an high correlation means that the two carriers have had the same mean arrival delay comparing the same days of the week. AA and DL, and VX and HA have an high correlation, positive and negative respectively. The rest of the heatmap does not bring significant information.

0.2.13 Exercise 2.13

We need again a pivot table, but the input DataFrame needs to be filtered beforehand. To do so, we can make use of a boolean mask.

```
[58]: mask = df.unique_carrier.isin(["HA", "DL", "AA", "AS"])
dcw = pd.pivot_table(df.loc[mask], values='delatdelay', index='unique_carrier',
↳columns='weekday', aggfunc='mean')
dcw
```

```
[58]: weekday          0          1          2          3          4          5  \
unique_carrier
AA          -3.576209  -4.621619  -4.601184  -4.091436  -3.553686  -4.558771
AS          -1.690789  -1.625446  -1.889198  -2.130705  -2.624551  -3.453872
DL          -8.913563 -10.211625 -10.544913 -10.604603 -9.623199  -5.979988
HA           0.258359   0.307772   0.759189   0.390762   0.746528   1.330508

weekday          6
unique_carrier
AA          -3.747053
AS           0.542632
DL          -7.001644
HA           1.207066
```

Finally, we can use the pandas plot method. Calling it on the whole DataFrame, a line for each column is created, with the index values on the x-axis. Thus, we only need to transpose the DataFrame in advance.

```
[59]: dcw.T.plot()
```

```
[59]: <matplotlib.axes._subplots.AxesSubplot at 0x1a21ef0b38>
```

