Politechnika Wrocławska

Faculty of Computer Science and Management
Field of study: **COMPUTER SCIENCE**

Bachelor Thesis

Big Data techniques with Python

Oleksii Kyrylchuk

keywords:
big data analysis
data cleansing
data visualization

The aim of the work is to demonstrate how to operate big data in Python programming environment as well as the tools that help to do it.

Supervisor	dr. hab. inż. Dariusz Król	.....	.....
	Title/ degree/ name and surname	*grade*	*signature*
The final evaluation of the thesis			
Przewodniczący Komisji egzaminu dyplomowego	.....	.....	.....
	Title/ degree/ name and surname	*grade*	*signature*

For the purposes of archival thesis qualified to: *

a) Category A (perpetual files)
b) Category BE 50 (subject to expertise after 50 years)

* Delete as appropriate

stamp of the faculty

Wrocław 2018

The writing of the thesis has been requested.

Contents

1 INTRODUCTION **6**

1.1 Goals of the Research 6

1.1.1 Time series data 6

1.1.2 Graph data 6

1.1.3 Geographical data 7

1.2 Thesis Overview 7

1.2.1 Time series data 7

1.2.2 Graph data 8

1.2.3 Geographical data 8

2 TIME SERIES DATA ANALYSIS **9**

2.1 Data gathering (pandas-datareader) 9

2.1.1 Downloading the data 9

2.1.2 Data transformation 12

2.1.3 Summary 15

2.2 Traditional stock visualizations (matplotlib) 16

2.2.1 Preparaion 16

2.2.2 Basic plot 17

2.2.3 Moving average plot 18

2.2.4 Candlestick plot 20

2.3 Stock correlation matrix visualization (matplotlib) 23

2.3.1 Stock correlation 23

2.3.2 Naive plot 24

2.3.3 Correct plot 26

2.3.4 Summary and necessity of plotting the matrix 28

2.4 Predictive analysis with a LSTM neural network (keras) 29

2.4.1 Long short-term memory networks 29

2.4.2 Data preparation 29

2.4.3 Network creation 32

2.4.4 Network training 34

2.4.5 Results overview 35
2.5 Summary 36

3 GRAPH DATA ANALYSIS 37
3.1 Preprocessing, igraph creation and community detection 37
3.1.1 Packages used 37
3.1.2 Pandas dataframe 37
3.1.3 Igraph creation 38
3.1.4 Community detection 40
3.2 Plotting large graphs (datashader) 41
3.2.1 Simple plot 41
3.2.2 Regular plot 43
3.2.3 Plotting communities 48
3.3 Plotting smaller graphs (igraph) 51
3.3.1 Cairo library 51
3.3.2 Regular plot 52
3.3.3 Community plot 58
3.3.4 Weighted community plot 63
3.4 Summary 65

4 GEOGRAPHICAL DATA ANALYSIS 66
4.1 Country level 66
4.1.1 Data loading 66
4.1.2 Country level visualization 67
4.1.3 Results 69
4.2 City level visualization 71
4.2.1 Data loading 71
4.2.2 Plotting with bokeh and datashader 73
4.2.3 Results 74
4.3 Summary 76

5 CONCLUSION 78

6 REFERENCES 80

Streszczenie

W dzisiejszych czasach znaczenie analizy dużej objętości danych staje się bardziej i bardziej oczywista. Coraz więcej firm decyduje się otworzyć dział analizy danych w celu poprawy własnej oferty; inne z analizy danach utworzyły ofertę dla innych firm. Praca inżynierska prezentuje wybrane metody analizy danych oferowane przez środowisko programowania Python, szeroko wykorzystanym w dziedzinie danologii. Szczególną uwagę poświęcono na pokazaniu czytelnikowi przykładów, jak można wykorzystać Python dla analizy danych z publicznie dostępnych źródeł przez prezentacje kodu z różnych projektów, jak i przykładowych wyników, które mogą zostać uzyskane.

Abstract

The importance of analyzing large volumes of data is becoming more and more apparent in the modern era. An increasing number of companies decide to open a data analysis department in order to improve their offerings, and some of them even base their product around analyzing data for other people. This work presents some of the data analysis methods using the Python programming language environment, which is widely used in the domain of data science. It focuses on showing the reader examples of how one can use Python to analyze data from publicly available sources by presenting the source code of different projects as well as the results that it generates.

1 INTRODUCTION

1.1 Goals of the Research

In the modern world, big data and machine learning are becoming more and more prominent as companies such as Facebook, Google and Amazon gather and analyze all sorts of data from their users. But which tools are they using to do it?

Right now, the two main languages in data science are Python [26] and R, while Java is the third by popularity and Matlab is also quite popular despite only being used in the academic environment.

The main goal of this work is to show how to use the Python 3 programming language in dealing with different kinds of data. It may be useful for long-term users of other languages that want to try Python out as well as users of Python 2, support for which will be stopped in 2020. You can see a more detailed breakdown below.

1.1.1 Time series data

- To show how to use python to obtain stock price data, and transform it for further use.
- To show how to use matplotlib to plot some of the more traditional single stock visualizations.
- To show how you can visualize a stock correlation matrix, as well as to discuss the necessity of such visualization.
- To show how to predict future stock prices using a long short term memory neural networks.

1.1.2 Graph data

- To show how to load a graph from an edge list file
- To show how to detect communities in a graph.
- To show how to run community detection algorithms in igraph
- To show how to plot regular large graphs with datashader
- To show how to plot large graphs while distinguishing communities in them

- To show how to create a regular graph's plot with igraph
- To show how to create a graph plot with marked communities on it, and compare it to the regular plot
- To show how to create a graph plot with marked communities that also shows the weight of its nodes, and compare this plot to the other two

1.1.3 Geographical data

- To show how to load geographical data from a sqlite3 database using pandas
- To show how to plot a country-level indicator plot using the loaded geographical data
- To show how to load location data from a file using dask
- To show how to transform the location data using dask
- To show how to plot the location data on the real-world map with datashader

1.2 Thesis Overview

This works takes a project-oriented approach to data science, showing the reader the insides and results of mini-projects conducted in three different areas, each using a different type of data. You can find the area overview below.

1.2.1 Time series data

The data source for this project is daily prices of 2557 stocks spanning across 17 years taken from NYSE[1].

This project will focus on using packages called numpy[31] and pandas[18] to process the data as well as packages like matplotlib[12] to plot it and keras[6] to predict future data.

In the first section I will show you how to download the data using pandas-datareader as well as how to merge the individual stock data files into one big file for easier processing.

In the second section, I will show how to create more traditional visualizations of singular stocks using matplotlib.

[1]New York Stock Exchange

In the third section, I will show how to create a stock correlation chart with matplotlib and discuss its usability.

In the fourth section, I will show how to create a long short-term memory neural network using keras, how to train it on the stock data and how to obtain save it and obtain predictions.

1.2.2 Graph data

The data source for this project is a graph with .

This project will focus on processing graph data using the igraph package for Python that is built on top of the C library with the same name [7] and visualizing it using datashader [13] and plotting functions that are built into igraph. The data source for this project is a graph of the YouTube social network from 2007, that was downloaded from The Koblenz Network Collection [19].

In the first section I will focus on reading the obtained files and importing it in the Python environment by creating an `igraph.Graph()` object. I will also show how to detect communities in an igraph by using such algorithms as louvain[4] and infomap[28].

In the second section, I will cover visualization of the whole graph as well as a large subsection of the graph using datashader.

In the third section, I will cover visualization of a smaller subsection of the graph using the plotting methods that are built into igraph.

1.2.3 Geographical data

This project will contain two data sources - one for country-level data, and the second one for city-level data.

In the first section, dedicated to the country-level data, I will show how to create a country indicator chart using matplotlib and the basemap package.

The data source for the country-level project is a database that contains a collection of development indicators for 227 countries obtained from the world bank[2].

In the second section, dedicated to the city-level data, I will show how to create a scatter plot that shows that locations of taxi pickups and dropoffs using datashader and dask.

The data source for the city-level project is a file that, after cleaning, contains 15.5 millions rows of taxi trip data obtained from the Chicago Data Portal [5].

2 TIME SERIES DATA ANALYSIS

2.1 Data gathering (pandas-datareader)

2.1.1 Downloading the data

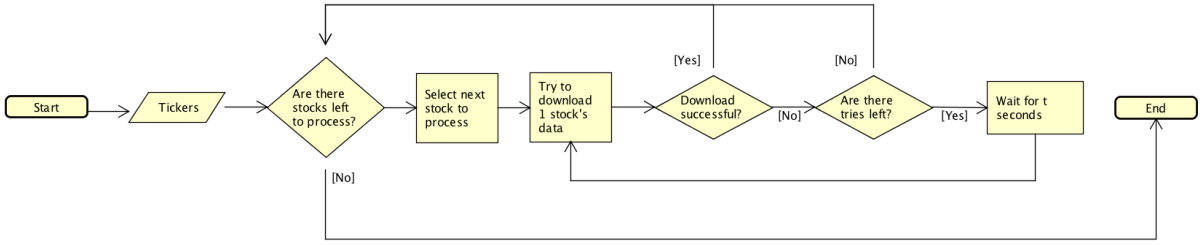

Figure 2.1: Stocks download flowchart

As you can see on the figure 2.1, the download process can only begins after you have specified the tickers, or short names of the companies that you want to download the data for.

There are a couple of methods to obtain a list of tickers that you want to analyze: you could go through the companies that interest you and look up the ticker for every one of them, you could use a package called BeautifulSoup to obtain the tickers by scraping a webpage, or, if you are dealing with all of the stocks traded on a particular market (like we are here with NYSE), the official webpage for that market usually provides such a list. In this case, I have downloaded a list of all tickers from NYSE from the official NASDAQ website [21] and saved the ticker column from the provided file manually.

It is possible to just go trough them one by one and call the download function for every ticker, but that approach may be detrimental due to the fact that sometimes Yahoo! Finance's API just outright refuses the pandas-datareader's connection attempts, probably due to a large number of requests from one IP address.

Due to this inconsistency, we have to incorporate some fail-saves into the code, like checking whether the download was completed successfully and repeating it if something went wrong. The timeout between tries is also introduced here to avoid stressing the Yahoo! servers too much, and to avoid being flagged as a malicious user.

After a stock is processed this way, we save the data if we have successfully downloaded it, and go to the next stock in the list. While this approach doesn't guarantee that 100% of the stocks will get downloaded, it protects us from being stuck in an infinite loop due to Yahoo!

Finance not having a data for a certain stock.

Used packages

Below you will find packages that will be used in this section of the project.

```
# Python's STL packages
import datetime as dt
import gc
import logging
import os
import sys
import time
# Other packages
import numpy as np  # Arrays
import pandas as pd  # DataFrames
import pandas_datareader as web  # Gets stock data from Yahoo
from tqdm import tqdm  # Progress bar
```

Listing 2.1: Imports

Tqdm

Please pay attention to the use of the tqdm package in the listing 2.2. It is a very useful package that allows you to create progress bars that work in both console and jupyter notebooks, and help with determining how much time is left in a long process, or how many things we have already processed. To use it, you can just change `for i in iterable` to `for i in tqdm(iterable)`. And if you want to use it in while loops, you can create a tqdm object yourself and manually update it, as is is shown in the listing.

```
def get_stock_data(start_dt, end_dt, max_tries=50, timeout=2, provider='yahoo',
                   ticker_folder=os.path.join('data', 'tickers'),
                   ticker_fname='tickers.csv',
                   dest_folder=os.path.join('data', 'stocks')):
    '''
    Gets stock data of S&P500 from yahoo and saves it in the {folder}/{tick}.csv

    Throws an exception is anything goes wrong.
    '''
    logging.debug('Obtaining the tickers...')
    tickers = pd.read_csv(os.path.join(ticker_folder, ticker_fname))
    logging.debug('Obtained the tickers...')
    logging.debug(tickers)

    # We have to check whether the dest folder exists
    dest_path = dest_folder
```

```
    if not os.path.exists(dest_path):
        logging.debug('Creating the destination folder')
        os.makedirs(dest_path)

    # Downloading the prices
    logging.debug('Starting processing tickers...')
    down_cnt, to_down_cnt = 0, len(tickers)
    for index, ticker in tqdm(tickers.itertuples(), desc='Tickers processed',
                              leave=False, file=sys.stderr, unit='company',
                              total=tickers.shape[0]):
        df = None
        logging.debug(f'Starting a new outer loop iteration for {ticker}')
        dest_fpath = os.path.join(dest_path, f'{ticker}.csv')
        # Check if we downloaded it before
        if not os.path.exists(dest_fpath):
            # Try to download the stock for max_tries tries, waiting
            # for timeout in between tries
            pbar = tqdm(range(max_tries), desc='Number of tries',
                        leave=False, file=sys.stderr, unit='try')
            tries = max_tries
            while tries > 0:
                pbar.update(1)
                try:
                    logging.debug(f'Trying to get {ticker} data'
                                  f' from {provider}...')
                    df = web.DataReader(ticker, provider, start_dt, end_dt)
                    tries = 0
                    down_cnt += 1
                except Exception as e:
                    tries -= 1
                    logging.debug(e)
                    logging.debug(f'{provider} has denied our request - '
                                  f'sleeping for {timeout} seconds')
                    time.sleep(timeout)
            pbar.close()

            if df is None:
                logging.debug(f'Couldn\'t get the {ticker} data. Continuing')
                continue
            logging.debug(f'Successfully got {ticker} data from {provider}. '
                          'Now saving it...')

            df.to_csv(dest_fpath)
            logging.debug(f'Saved the {ticker} data.')
        else:
            to_down_cnt -= 1
            logging.debug(f'Not downloading data for {ticker}, '
                          'since we already have it')

```

```
66      logging.info('Finished processing all tickers!')
67      logging.info(f'Downloaded: {down_cnt}/{to_down_cnt} items')
68      logging.info(f'You can find the results in the folder {dest_path}')
```

Listing 2.2: Downloading the stocks data

2.1.2 Data transformation

After downloading the stocks, we are left with a lot of files, where every file represents the data for a single stock. And this is a problem if we want to run some computations that involve every stock, since then we would need to loop through all the individual files in order to achieve that. That's why in this subsection I'll show how to merge all those files in one CSV file.

There are a couple of ways to accomplish this, but the one I find the most useful is to make numpy arrays out of the files, and then use the `np.concatenate` function to merge them. It is even possible to convert the resulting numpy array to a dataframe afterwards.

But there is one problem with this method. In order to concatenate two numpy arrays, they must have the same number of rows (or columns, depending on which axis you are using to concatenate the arrays). And even though we have specified that we want data in a time period between 2000 and 2017, not every stock will actually have those dates in their CSV file. Since if a company had their IPO[2] after 2000, their file will start with the date that the stock could be traded.

In order to merge the files despite this, it is necessary to first go through every file list and make sure that they have the same number of rows inside, even if some of those rows are null. In order to do this, we can handpick one file to serve as an example, and then assign that file's index to every other file. You can observe this process in listing 2.5.

```
69  def get_timearr(stock_dir, example_timefile='A.csv'):
70      time_df = pd.read_csv(os.path.join(stock_dir, example_timefile),
71                            index_col=0)
72      return time_df.index.values
```

Listing 2.3: Extracting a time array (index) from a file

You can see a usage of generators in the listing 2.4. A generator can usually be distinguished by the use of the `yield` keyword, and is essentially a function that is treated as an iterable. If a you have a function that generates large amounts of data, your program can benefit drastically

[2]Initial Public Offering - the first time that the company can be traded

from using generators, since everything is generated on the fly and you don't have to store the whole resulting list in memory.

```
73 def list_csv(path):
74     for f in os.listdir(path):
75         if f.endswith('.csv'):
76             yield f
```

Listing 2.4: Listing all csv files in a directory

```
77 def reindex_csv(stock_dir, save_dir, time_arr, reload_data=False):
78     # If there is no save directory, create it
79     if not os.path.isdir(save_dir):
80         os.makedirs(save_dir)
81 
82     for stock_fname in tqdm(list_csv(stock_dir)):
83         # Check for already reindexed files
84         if not reload_data and os.path.isfile(os.path.join(save_dir,
85                                                            stock_fname)):
86             continue
87         # Reindex the files
88         stock = pd.read_csv(os.path.join(stock_dir, stock_fname), index_col=0)
89         stock = stock.reindex(time_arr, fill_value=np.nan)
90         stock.to_csv(os.path.join(save_dir, stock_fname))
91     gc.collect()  # Use garbage collector to clean the unused objects
```

Listing 2.5: Reindexing the files for future concatenation

One other problem that arises with merging the files is how to deal with columns being called the same way in every file. In this example, I have decided to solve it by appending the company name to every column, so we can still access data from a specific company in the future.

```
92 def get_per_diff(old, new):
93     return abs(new - old) / old
94 
95 
96 def merge_dfs(stock_folder, save_folder, save_fname='stocks_all_merged.csv',
97               reload_data=False, add_per_oc=True, add_per_lohi=True,
98               add_volume=True):
99     '''
100    Merges the stock csv files into one big file with all adj. closes
101
102    Raises an exception if something goes wrong.
103    '''
104
105    if (os.path.isfile(os.path.join(save_folder, save_fname)) and
106            not reload_data):
```

```
        logging.warning('The target file is already present in the save_folder.'
                        ' Please use the reload_data argument to overwrite it.')
        return

    logging.debug('Started merging the stock data - getting the files')
    fnames = sorted(list(list_csv(stock_folder)))
    logging.debug('Number of csv files in the folder: {}'.format(len(fnames)))
    logging.debug(f'Filelist: {fnames}')

    time_arr = get_timearr(stock_folder, fnames[0])
    to_stack = []
    col_names = []

    logging.debug('Starting merging dataframes')
    for cur_fname in tqdm(fnames, desc='Files processed', file=sys.stdout,
                          leave=True, unit='file'):
        cur_fpath = os.path.join(stock_folder, cur_fname)
        cur_ticker = cur_fname[:-4]
        col_names.append(cur_ticker)

        logging.debug(f'Processing the file {cur_fname}')
        cur_df = pd.read_csv(cur_fpath, index_col=0)

        if add_volume:
            col_names.append(f'{cur_ticker}_Vol')
        else:
            cur_df.drop(['Volume'], inplace=True, axis=1)

        if add_per_oc:
            cur_df['PerOC'] = get_per_diff(cur_df['Open'], cur_df['Close'])
            col_names.append(f'{cur_ticker}_OC')

        if add_per_lohi:
            cur_df['PerLH'] = get_per_diff(cur_df['Low'], cur_df['High'])
            col_names.append(f'{cur_ticker}_LH')

        cur_df.drop(['Open', 'Close', 'High', 'Low'],
                    inplace=True, axis=1)
        to_stack.append(cur_df.as_matrix())

    # It's faster to just stack a list of numpy arrays than to try and merge dfs
    merged_df = pd.DataFrame(np.concatenate(to_stack, axis=1),
                             index=time_arr, columns=col_names)

    logging.debug('Finished merging dataframes')
    save_path = os.path.join(save_folder, save_fname)
    logging.debug(f'Saving the data to {save_path}')

    if not os.path.exists(save_folder):
```

```
156         os.makedirs(save_folder)
157
158     merged_df.to_csv(save_path)
159
160     return merged_df
```

Listing 2.6: Merging the singular stock files

A correlation matrix serves as a way to store relationships between stocks, and will be covered later in the section 2.3.

```
161 def make_corr_matrix(merge_folder, save_folder,
162                       merged_close_fname='stocks_close_merged.csv',
163                       save_fname='corr_matrix.csv', reload_data=False):
164     if (os.path.isfile(os.path.join(save_folder, save_fname)) and
165             not reload_data):
166         logging.warning('The target file is already present in the save_folder.'
167                         ' Please use the reload_data argument to overwrite it.')
168         return
169
170     merged_path = os.path.join(merge_folder, merged_close_fname)
171     save_path = os.path.join(save_folder, save_fname)
172
173     logging.debug(f'Opening the merged closes folder at {merged_path}')
174     merged_df = pd.read_csv(merged_path)
175     corr_df = merged_df.corr()
176
177     logging.debug(f'Saving the corr_df to {save_path}')
178     corr_df.to_csv(save_path)
179
180     return corr_df
```

Listing 2.7: Creating a correlation matrix

We can check that the transformation executed properly by comparing the number of stocks in the source folder with the number of columns in the merge dataframe.

```
1 # Check that all stocks have been merged
2 merged = merge_dfs(STOCK_FOLDER, MERGED_FOLDER, reload_data=True)
3 print(len(os.listdir('stocks/')) == merged.shape[1] / 4)  # Returns true
```

Listing 2.8: Verifying the transformation

2.1.3 Summary

After defining all those functions, the only thing left now is to call them with correct arguments. As you can see from the listing 2.9, I use a datetime package from python's standard library to

specify the date ranges that I want to retrieve the data for, and in this case it's from 2000 until 2017.

```
# We want to get the data from 2000 till 2017
START_DT = dt.datetime(2000, 1, 1)
END_DT = dt.datetime(2017, 1, 1)
# We want to place the merged file in data/merged
STOCK_FOLDER = os.path.join('data', 'stocks')
MERGED_FOLDER = os.path.join('data', 'merged')

# Downloading the data
get_stock_data(START_DT, END_DT, max_tries=5, ticker_fname='NYSE.csv',
               dest_folder=STOCK_FOLDER, timeout=0.1, provider='yahoo')

# Reindexing the csvs so we can np.concatenate them later
reindex_csv(STOCK_FOLDER, STOCK_FOLDER, get_timearr(STOCK_FOLDER, 'A.csv'),
            reload_data=True)

# Merging the dataframes
merge_dfs(STOCK_FOLDER, MERGED_FOLDER, reload_data=True)
merge_dfs(STOCK_FOLDER, MERGED_FOLDER, save_fname='stocks_close_merged.csv',
          reload_data=True, add_per_oc=False, add_per_lohi=False, add_volume=False)

# Making the correlation matrix
make_corr_matrix(MERGED_FOLDER, MERGED_FOLDER, reload_data=True)
```

Listing 2.9: Executing the functions

To sum up, while the pandas-datareader module requires a lot of extra checking to ensure that the data was actually downloaded, it simplifies the process of web scraping tremendously, since you can just call a function that accesses certain provider's API by itself. And despite it being limited by the number of providers you can access with it, I would say that the benefit of not having to parse the html or json yourself can play a significant role in programmers choosing this package over plain old web scraping.

2.2 Traditional stock visualizations (matplotlib)

2.2.1 Preparaion

Imports

The packages that will be used in this section are listed in the listing 2.10 below.

```
# STL
import os
```

```
# Other
import matplotlib
import matplotlib.pyplot as plt
import numpy as np  # Arrays
import pandas as pd  # DataFrames
from matplotlib.finance import candlestick_ohlc  # Candlestick graph
```

Listing 2.10: Imports

Data loading

We don't need to do any preparations in this section except for loading the data in memory. Please note the usage of `plt.style.use('seaborn')` in the listing 2.11 - it changes the default simple looking style of matplotlib plots into a one that looks more modern.

```
plt.style.use('seaborn')  # Fancier matplotlib plots

STOCKS_FOLDER = os.path.join('data', 'stocks')

print('Getting data...')

print('Single stock...')
df = pd.read_csv(os.path.join(STOCKS_FOLDER, 'A.csv'),
                 parse_dates=True,
                 index_col=0)

print('Finished getting data')
```

Listing 2.11: Reading the data into the memory

2.2.2 Basic plot

A basic plot of one stock can be plotted without having to resort to creating your own plot in matplotlib, since a pandas DataFrame has a built-in plot function. In this example, I created an empty matplotlib figure first, then plotted the stock data on top of it and displated the figure using the `plt.show()` function. You can also call `plt.savefig('filename.pdf')` instead, if you want to save your plot to a file, or just click the save icon in the interactive window that `plt.show()` generates.

```
# One stock
plt.figure()
df['Adj Close'].plot()
plt.show()
```

Listing 2.12: Very basic plot of a stock

Figure 2.2: Basic plot

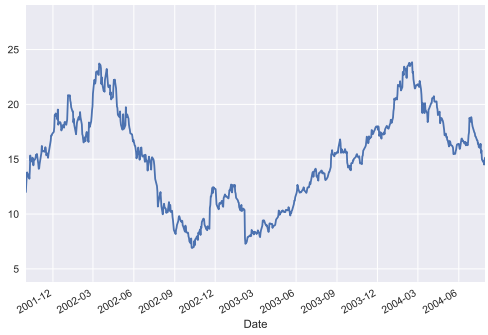

Figure 2.3: Basic plot - zoomed

2.2.3 Moving average plot

In this subsection, I will show how to create a moving average plot. Moving average plots are used by analysts because they help to bring forward long-term trends and serve as a measure to smooth out outliers.

In pandas, you can create a special column that keeps track of the moving average of last `n` days by using the command `df['column'].rolling(window=n).mean()`. Then, you can use this newly generated column as another input for the `plot` function, and plot the moving average and the close price on the same figure. You can also do this process for different values of n and plot multiple moving averages to compare them - if a MA with a smaller `n` rises and crosses the MA with a larger `n`, it means that the raise in the stock price is substantial and it might be worth it to invest in such a stock.

I would also like to mention the `subplot2grid((y, x), (posy, posx))` function that you can see in the example - it breaks the figure down into an y by x grid, and allows you to pick a point that you would like to position a new axis object in. `rowspan` and `colspan` are used to determine how many rows or columns respectively this part of the plot should take.

```
25 # Moving average
26 # Preparing the data
27 df_ma = df.copy()
28 # Creating the rolling avg column
29 df_ma['50MA'] = df_ma['Adj Close'].rolling(window=50, min_periods=0).mean()
30
31 # Plotting
32 plt.figure()
33
34 # Axis 1
35 ax1 = plt.subplot2grid((12, 1), (0, 0), rowspan=11, colspan=1)
```

```
ax1.xaxis_date()
ax1.plot(df_ma.index, df_ma['Adj Close'], color='#56648C')
ax1.plot(df_ma.index, df_ma['50MA'], color='#FF5320', linewidth=1)
plt.title('50 days moving average', fontsize=10, color='k')
ax1.yaxis.set_label_text('Stock price ($/share)')

# Axis 2
ax2 = plt.subplot2grid((12, 1), (11, 0), rowspan=1, colspan=1, sharex=ax1)
ax2.fill_between(df_ma.index, df_ma['Volume'],
                 0, color='#56648C', label='Volume')

# Plot general
plt.setp(ax2.xaxis.get_majorticklabels(), rotation=45)
plt.suptitle('Agilent Technologies, Inc.', fontsize=12, color='k')
plt.show()
```

Listing 2.13: Moving average plot

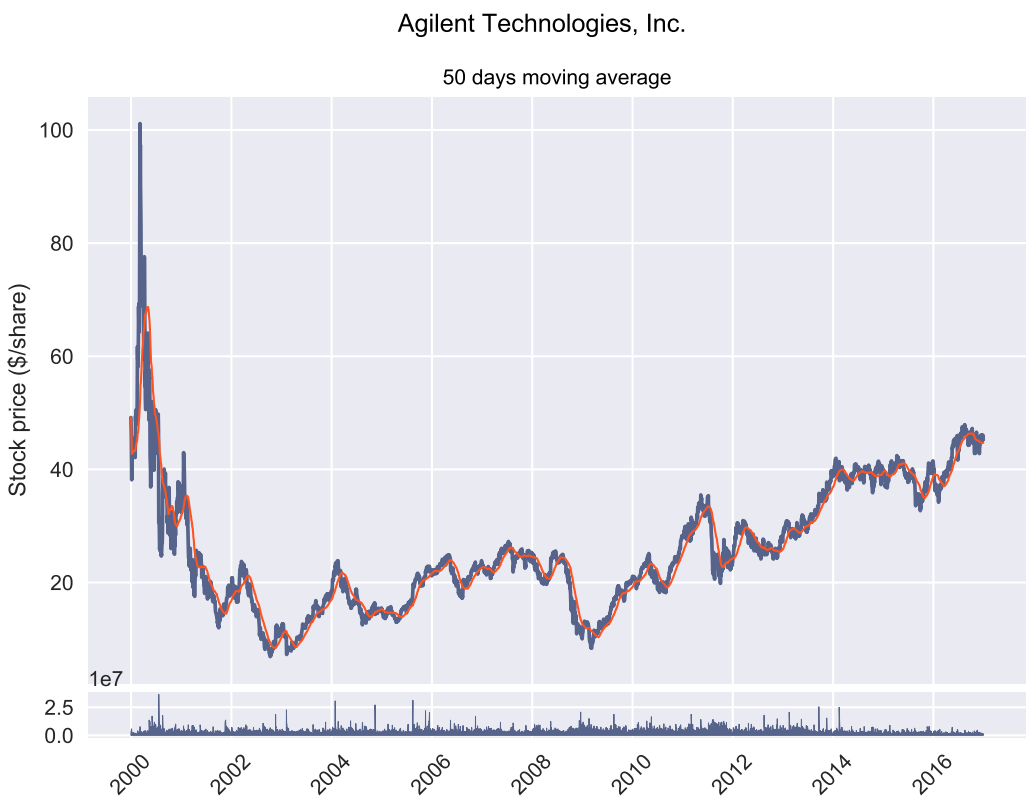

Figure 2.4: Moving average plot

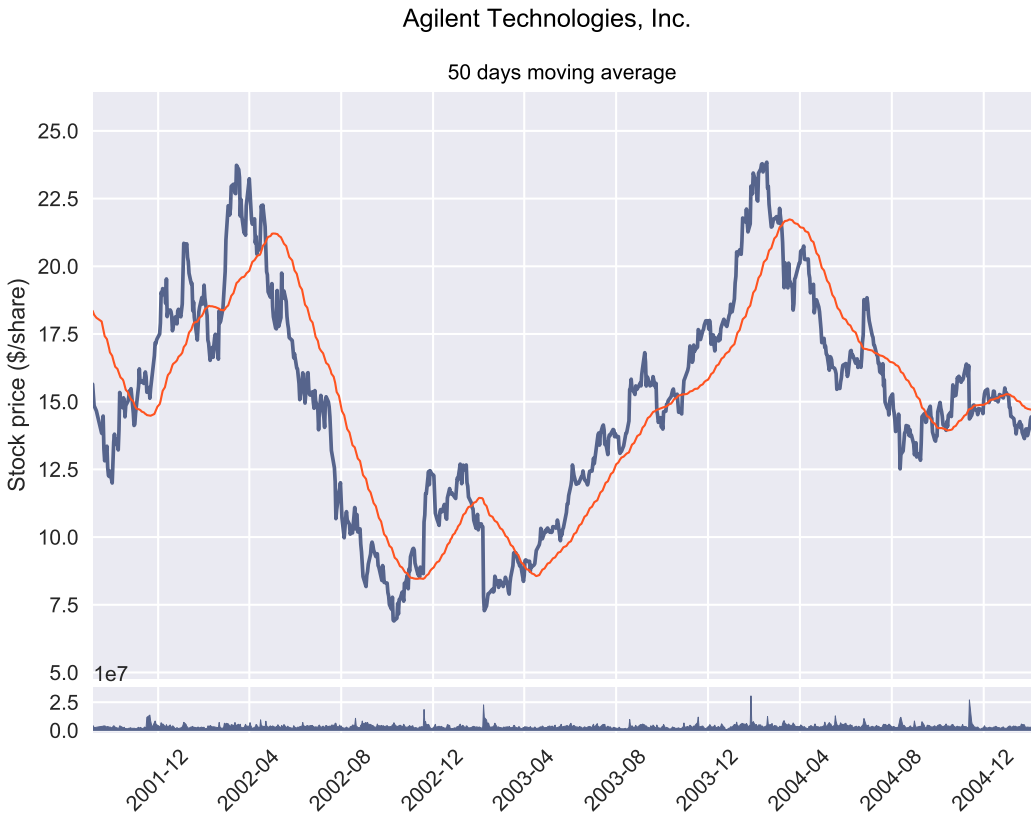

Figure 2.5: Moving average plot - zoomed

2.2.4 Candlestick plot

A candlestick plot is a way to visualize aggregated stock data that has originated in Japan and was popularized in the west by Steve Nison [22]. It employs a structure that is similar to box plots to show the stock's opening, close, high and low prices for a specific time period - usually a day or a week. Usually, a entry for one day (or week, month, etc.) consists of the following: (1) a box, showing opening and close prices and (2) lines, or "shadows", that show the high and low prices (you can see those on figure 2.6). This, in combination with changing the color of the box to show whether the stock has gone up or down in price to create this plotting technique. Candlestick plotting is also often combined with a volume plot that shows how much trading was done in that time period (on the bottom in the example).

Figure 2.6: Candlestick plot - zoomed

The pandas package has some support for the candlestick plots, which, as you can see in the listing 2.14, can be very useful when trying to create such plots by yourself. You can use functions like `df[column].resample('time_period')` to resample your data and then pair it with `df.ohlc()` to extract the open/high/low/close values from that aggregation. This, paired with `candlestick_ohlc` function from `matplotlib.finance` module allows you to create candlestick plots rather quickly.

```
51 # Candlestick
52 plt.figure(figsize=(8, 6))
53 date2num = matplotlib.dates.date2num
54
55 df_ohlc = df['Adj Close'].resample('10D').ohlc()
56 df_vol = df['Volume'].resample('10D').sum().to_frame()
57
58 df_ohlc.reset_index(inplace=True)
59 df_vol.reset_index(inplace=True)
60
61 # Converting the dates to the matplotlib's date format
62 df_ohlc['Date'] = df_ohlc['Date'].map(date2num)
```

```
63  df_vol['Date'] = df_vol['Date'].map(date2num)
64
65  # Axis 1
66  ax1 = plt.subplot2grid((12, 1), (0, 0), rowspan=11, colspan=1)
67  ax1.xaxis_date()
68  # Plotting the candlestick
69  matplotlib.finance.candlestick_ohlc(ax1, df_ohlc.values, colorup='g', width=2)
70  ax1.set_title('Candlestick plot', fontsize=10, color='k')
71  ax1.yaxis.set_label_text('Stock price ($)')
72
73  # Axis 2
74  ax2 = plt.subplot2grid((12, 1), (11, 0), rowspan=1, colspan=1, sharex=ax1)
75
76  # Fill between two curves
77  # (x, y_high, y_low)
78  ax2.fill_between(df_vol['Date'], df_vol['Volume'], 0, label='Volume')
79  ax2.xaxis.set_label_text('Date')
80
81  # General
82  plt.setp(ax2.xaxis.get_majorticklabels(), rotation=45)
83  plt.suptitle('Agilent Technologies, Inc.', fontsize=12, color='k')
84  plt.show()
```

Listing 2.14: Candlestick chart

As you can see in the figures 2.6 above and 2.7 below, the resulting plot is interactive and allows for the user to zoom in and only view the time period that they are interested in.

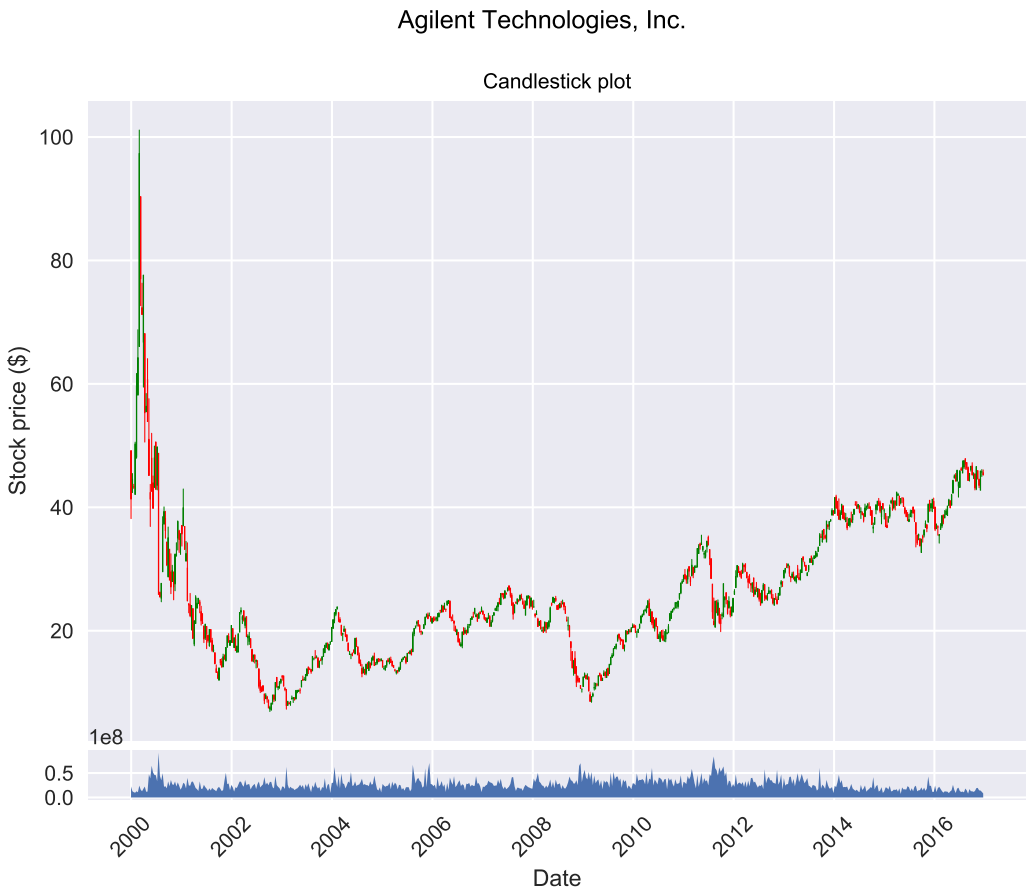

Figure 2.7: Candlestick plot

2.3 Stock correlation matrix visualization (matplotlib)

2.3.1 Stock correlation

A correlation between two stock prices shows, whether a rise in price of one stock correlates with a rise or fall in price of another stock, or whether two stocks are completely independent.

A logical, albeit a bit inefficient, way to store correlation between a certain number of stocks is to create a so-called correlation matrix, whose structure you can see in table 2.1.

Some research[16] suggests that correlation matrix may contain useful information about the stock market, which makes it into a metric that can be used by stock investors that would like to diversify their investment portfolio as much as possible, thus ensuring that if one of the stocks they have invested in drops in price, the other ones remain stable.

Once we have this matrix, a question arises - how can we visualize it? In this example visualization, I'll use a heat map chart, because of how the correlation numbers can be treated as colors.

Table 2.1: Correlation matrix structure

	Stock1	Stock2	Stock3
Stock1	1	corr(Stock1, Stock2)	corr(Stock1, Stock3)
Stock2	corr(Stock2, Stock1)	1	corr(Stock2, Stock3)
Stock3	corr(Stock3, Stock1)	corr(Stock3, Stock2)	1

```
# STL
import os
# Other
import matplotlib
import matplotlib.pyplot as plt  # Plots
import numpy as np  # Arrays
import pandas as pd  # DataFrames
```

Listing 2.15: Imports

```
plt.style.use('ggplot')
MERGED_FOLDER = os.path.join('data', 'merged')

print('Getting data...')

print('Merged correlation matrix...')
df_corr = pd.read_csv(os.path.join(
    MERGED_FOLDER, 'corr_matrix.csv'), index_col=0)

print('Finished getting data')
```

Listing 2.16: Reading the data into the memory

2.3.2 Naive plot

In this plot, I have used the `pcolormesh` function in order to create a heat map of the values from the df_corr dataframe. Data didn't require any extra preprocessing, since the correlations are already in a range `[-1:1]`, and that means that the colors will render properly.

Unfortunately, if you try to plot a heat map this large, some problems starts to occur with matplotlib. In this case, if just thought that the last cell to the right was a stock with a name starting with "A", even though it should be something from the other part of the alphabet, like "Z". And even if you try to zoom into the plot, those labels don't update and continue to be wrong. I will cover how to deal with this problem by inheriting from a formatter class in the section 2.3.3.

```
# PROBLEM - LABELS ARE WRONG
# Making a correlation matrix
```

```
fig = plt.figure()

# Setting up the axis
ax1 = plt.subplot2grid((1, 1), (0, 0), rowspan=1, colspan=1)
ax1.invert_yaxis()

hmap = ax1.pcolormesh(df_corr.values, cmap=plt.cm.RdYlGn)
hmap.set_clim(-0.5, 0.5)  # Clipping limit
fig.colorbar(hmap)

ax1.set_xticklabels(df_corr.columns)
ax1.set_yticklabels(df_corr.columns)

# Plotting the heatmap
plt.title('Stock correlation heatmap', color='k')
plt.show()
```

Listing 2.17: Correlation matrix - labels are broken

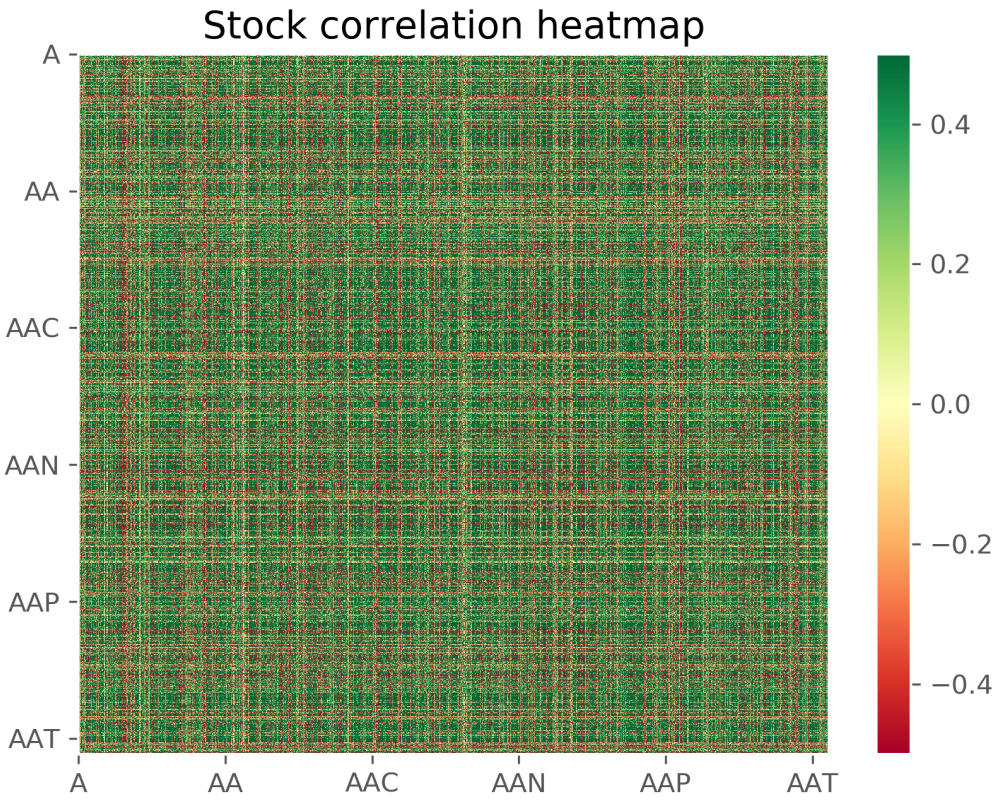

Figure 2.8: Correlation plot (incorrect labels)

2.3.3 Correct plot

The default matplotlib tick formatter can't dynamically update values. This means, that if I would want to set the ticks myself, I would have to display all of them on the screen at the same time, which just leads to all the labels overlapping with each other. But we can inherit from the matplotlib's tick formatter class in order to solve this issue. As you can see in the listing 2.18, you only have to override two functions from the parent class in order to achieve dynamic update of the labels, even if it does make the plot somewhat stutter when you use the interactive zoom feature.

I have also used the MaxNLocator, which is similar to the default matplotlib's behavior - it will choose to use between min_n_ticks and nbins + 1 locations, and position them in "nice locations", according to matplotlib's documentation. Both of those variables are keyword arguments to its __init__ function, and can easily be changed.

I would also like to point out, that while it can't be seen in the figures below, if you mouseover a cell in the heat map while being in an interactive mode, the program displays you the stocks that the chosen cell represents.

```
36 class StockFormatter(matplotlib.ticker.Formatter):
37     def __init__(self, cols):
38         self.cols = np.array(cols)
39 
40     def __call__(self, x, pos=None):
41         return self.cols[np.clip(x, 0, len(self.cols) - 1).astype('int')]
42 
43 
44 # Making a correlation matrix
45 fig = plt.figure()
46 
47 # Preparing the axis formatters
48 cols = list(df_corr.columns)
49 formatter_x = StockFormatter(cols)
50 formatter_y = StockFormatter(cols)
51 
52 locator_x = matplotlib.ticker.MaxNLocator(10)
53 locator_y = matplotlib.ticker.MaxNLocator(10)
54 
55 # Setting up the axis
56 ax = plt.subplot2grid((1, 1), (0, 0), rowspan=1, colspan=1)
57 ax.invert_yaxis()
58 
59 # Plotting the table itself
60 hmap = ax.pcolormesh(df_corr.values, cmap=plt.cm.RdYlGn)
```

```
hmap.set_clim(-1, 1)  # Clipping limit
fig.colorbar(hmap)

x = ax.get_xaxis()
y = ax.get_yaxis()

# Setting tickers for X axis
x.set_major_formatter(formatter_x)
x.set_major_locator(locator_x)

plt.xticks(rotation=45)

# Setting tickers for Y axis
y.set_major_formatter(formatter_y)
y.set_major_locator(locator_y)

# Plotting the heatmap
plt.title('Stock correlation heatmap - fixed', color='k')
plt.show()
```

Listing 2.18: Fixing the labelling

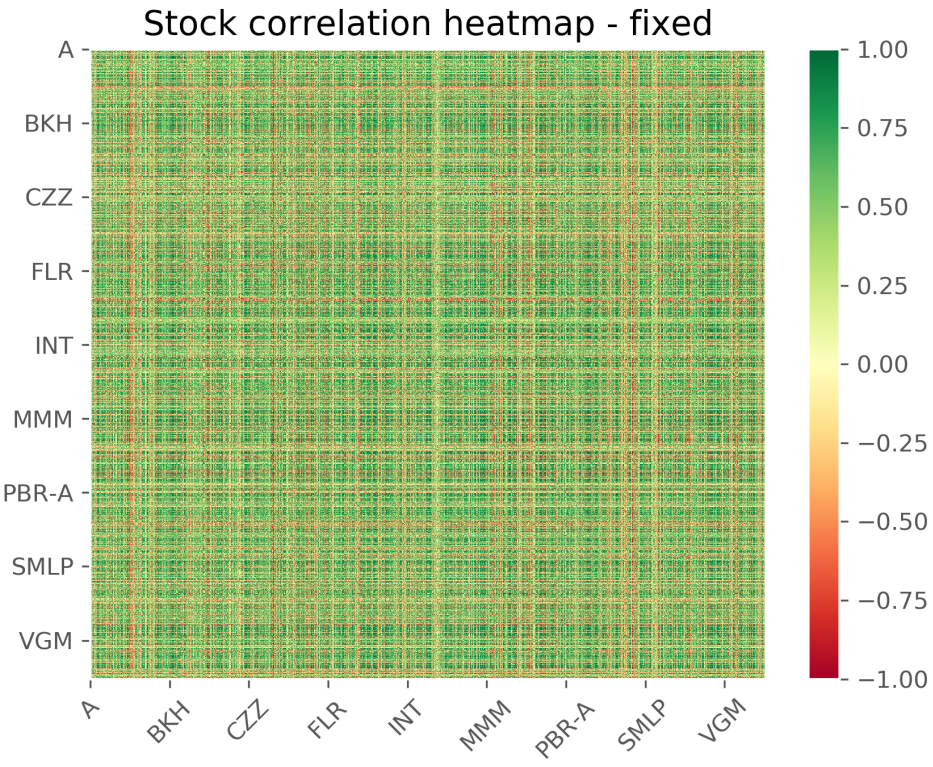

Figure 2.9: Correlation plot (fixed)

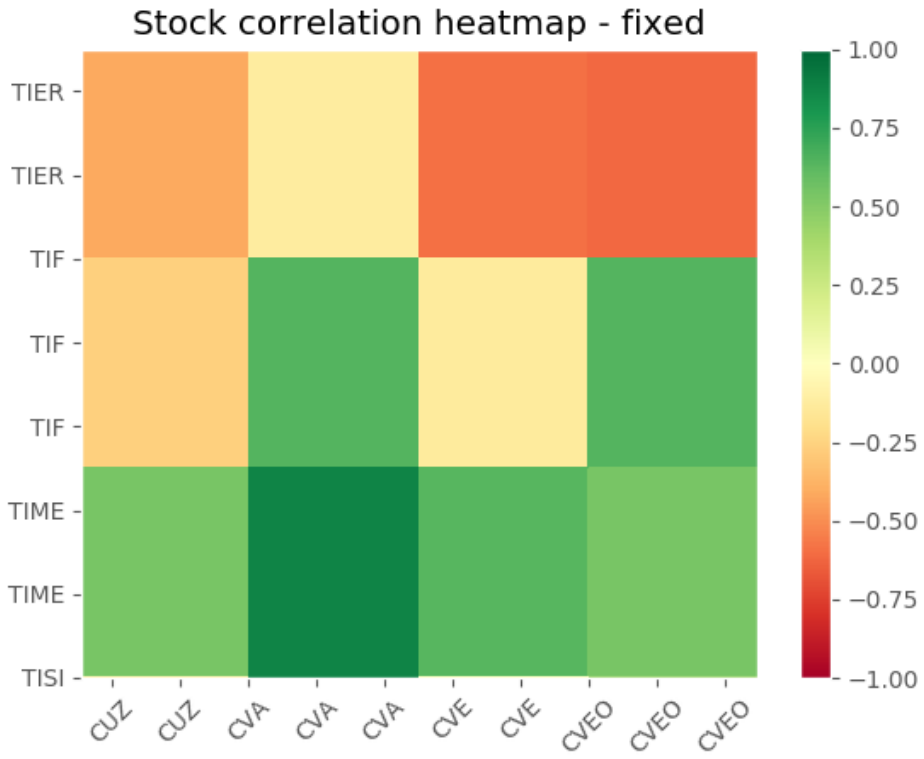

Figure 2.10: Correlation plot (fixed) - zoomed

2.3.4 Summary and necessity of plotting the matrix

In retrospective, I would say that there are problems with this plot simply because it's too cluttered and hard to understand. You have to zoom in whenever you want to actually see the correlation between some stocks and not just some vague green or red lines. However, once zoomed in it becomes quite easy to check the correlation between two plots. I think that it could be improved by either focusing more on the neutrally correlated stocks, and making the highly positive and highly negative areas transparent, or by making the number of stocks that we have to show smaller. Since the current version displays all stocks from the market and is very hard to read due to the number of rows and columns being too high, I think that a version with 5, 10 or maybe even 20 stocks could be better since then you would actually see all of the date from a picture without having to zoom.

2.4 Predictive analysis with a LSTM neural network (keras)

2.4.1 Long short-term memory networks

A long short-term memory neural network is a type of a recursive neural network, that was invented in 1997 by Hochreiter and Schmidhuber[11]. It replaces the standard neuron from a NN with a memory cell that not only learns how to predict something, but also how to forget something in order to be able to indentify long-term dependancies.

In this project, I will feed the neural network with the data for an entire NYSE stock market using the keras[6] with tensorflow[1] backend, and try to predict the price of one stock based on that.

Used packages

```
# STL
import os
import logging
import time
# Other
import pickle
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
# Keras
import keras
from keras.callbacks import EarlyStopping
from keras.models import Sequential, load_model
from keras.layers.core import Dense, Activation, Dropout
from keras.layers.recurrent import LSTM
from keras.callbacks import EarlyStopping
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler
# Plotting the LSTM-NN
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
```

Listing 2.19: Imports

2.4.2 Data preparation

Loading the data in the memory is the same as in the plotting section, but now we will need to preprocess it into a format that is acceptable by keras.

```
22 MERGED_FOLDER = os.path.join('data', 'merged')
23
24 MERGED_FNAME = 'stocks_all_merged.csv'
25 MERGED_PATH = os.path.join(MERGED_FOLDER, MERGED_FNAME)
26
27 # # We need to use index_col=0 to set it as a main
28 logging.info('Reading the merged dataset...')
29 df_merged = pd.read_csv(MERGED_PATH, index_col=0)
30 df_merged.fillna(0, inplace=True)
31 df_merged.reset_index(drop=True, inplace=True)
32 logging.info('Finished processing the merged dataset')
```

Listing 2.20: Loading data into the memory

Preprocessing

A LSTM NN requires us to feed it input in sequential blocks of data and not just block by block like a regular NN - this utilizes the memory function of the lstm neurons, and allows for these types of networks to predict next entry in a dataset (movie frames, stock prices, human speech, etc.). For the purpose of this example, I have chosen for the block size to be 30 days. It means, that we have to split out current dataset into 30-day continuous chunks. You can see me do it in the lines 59-67 of the listing 2.21. I start with an empty arrays X and y, and then add slices of 30 days to them until I can no longer add any.

You can also see how it is possible to scale features to a range between 0 and 1 (which improve your neural network's results [20] [10]) by using scikit-learn's[25] built in MinMaxScaler as well as how to execute a Principal Component Analysis[32] with another scikit-learn object. PCA reduces the number of features in your dataset, and while it may be detrimental to the results, a smaller number of features makes your neural network train a lot faster.

```
33 def convert_series(df_merged, window=30, pred_column='A', do_pca=False, pca=None, feature_len=None,
    ↪   do_scale=True):
34     X = []
35     y = []
36     col_number = df_merged.columns.get_loc('A')
37     scaler = None
38     data = df_merged.as_matrix()
39
40     # Scaling the data if requested
41     if do_scale:
42         logging.info('Starting scaling...')
43         scaler = MinMaxScaler(feature_range=(0, 1))
44         data = scaler.fit_transform(df_merged.as_matrix())
45         logging.info('Finished scaling')
```

```
46
47      data_y = data[:, col_number]
48      data_X = data
49
50      # Doing the princomp analysis to reduce dims if requested
51      if do_pca:
52          if pca is None:
53              logging.info('Executing PCA...')
54              pca = PCA(feature_len)
55              pca.fit(data_X)
56          data_X = pca.transform(data_X)
57      logging.info(f'Number of features: {data_X.shape[1]}')
58
59      # Processing X, y
60      logging.info('Making X, y datasets')
61      range_index = range(data_X.shape[0] - window - 1)
62      for i in range_index:
63          X.append(data_X[i: i + window])
64          y.extend(data_y[i + window: i + window + 1])
65
66      # Merging the disjointed arrays into one
67      X = np.stack(X)
68      y = np.array(y)
69      return X, y, pca, scaler
70
71
72  def split_x_y(df, pred_column='A', window=30, test_rows=260, do_pca=False, pca=None, feature_len=None,
     ↪   do_scale=False):
73      # Converting the dataframe to something we can feed NN with
74      logging.info('Converting dataframe to X,y')
75      X, y, pca, scaler = convert_series(
76          df, window, pred_column, do_pca, pca, feature_len, do_scale)
77
78      # Splitting the datasets
79      logging.info('Splitting X,y into train and test datasets')
80      X_train = X[:-test_rows]
81      y_train = y[:-test_rows]
82      X_test = X[-test_rows:]
83      y_test = y[-test_rows:]
84      return X_train, y_train, X_test, y_test, pca, scaler
```

Listing 2.21: Declaring the data preprocessing functions

The split_x_y function is mostly a wrapper for the convert_series function that converts the input dataframe into those window-sized chunks and splits the dataset into a train and a test set.

Please note the usage of the pickle package in the listing 2.22 - it can be very useful, since if you save your objects to the drive by pickling them you don't have to generate them again.

```
pca = None
with open('pca.pickle', 'rb') as f:
    pca = pickle.load(f)

logging.info('Splitting the data into train and test '
             'datasets as well as converting it to the correct format')
X_train, y_train, X_test, y_test, pca, scaler = split_x_y(
    df_merged, do_pca=False, pca=pca)
logging.info('Finished data preprocessing')

# Uncomment if you want to save a newly generated PCA object
# logging.info('Saving the PCA object...')
# with open('pca.pickle', 'wb') as f:
#     pickle.dump(pca, f)
# logging.info('Finished saving the pca object')
```

Listing 2.22: Executing the data preprocessing functions

2.4.3 Network creation

Netork creation is rather simple using keras - you just have to initialize an instance of the `Sequential` class to serve as your model's base, and then you can start adding layers to it using the `add` function. While it's rather obvious that `LSTM` represents a long short-term memory layer, a `Dense` layer is a layer filled with regular neurons, and a dropout keyword argument shows how many neurons will be randomly reset on each run. While it may sound counterintuitive, it sometimes is beneficial and helps to avoid overfitting. The loss chosen in this example is a mean squared error loss, or "mse", if we use keras' terms. The optimizer chosen for this example is Adam, which has been shown to converge faster than other similar algorithms in the study that introduced it[15].

```
def make_one_layer_lstm(num_neurons=2000):
    # input shape = (nb_samples,timesteps =30, input_dim= cols)
    logging.info('Making a 1-layer model with {num_neurons} lstm cells')
    model = Sequential()
    # Input layer
    logging.info('Processing input layer and hidden layer #1...')
    model.add(LSTM(
        num_neurons,
        return_sequences=False,
        input_shape=(X_train.shape[1], X_train.shape[2]),  # X_train.shape[1]
        dropout=0.1
    ))
    # Output layer
    logging.info('Processing output layer...')
```

```
    model.add(Dense(
        1,
        activation='linear'
    ))
    logging.info('Finished making the model')
    # Compiling the model
    logging.info('Started compilation...')
    start_time = time.time()
    model.compile(loss='mse', optimizer='adam')
    logging.info('Compiled in {}s'.format(time.time() - start_time))
    return model
```

Listing 2.23: Creating a neural network with one layer

```
def make_two_layer_lstm(first_layer=1000, second_layer=200):
    # input shape = (nb_samples,timesteps =30, input_dim= cols)
    logging.info(f'Making a 2-layer model, l1={first_layer}, l2={second_layer}')
    model = Sequential()
    # Input layer
    logging.info('Processing input layer and hidden layer #1...')
    model.add(LSTM(
        first_layer,
        return_sequences=True,
        input_shape=(X_train.shape[1], X_train.shape[2]),  # X_train.shape[1]
        dropout=0.1
    ))
    # Hidden layer #1
    logging.info('Processing hidden layer #2...')
    model.add(LSTM(
        second_layer,
        return_sequences=False,
        dropout=0.2
    ))
    # Output layer
    logging.info('Processing output layer...')
    model.add(Dense(
        1,
        activation='linear'
    ))
    logging.info('Finished making the model')
    # Compiling the model
    logging.info('Started compilation...')
    start_time = time.time()
    model.compile(loss='mse', optimizer='adam')
    logging.info('Compiled in {}s'.format(time.time() - start_time))
    return model
```

Listing 2.24: Creating a neural network with two layers

2.4.4 Network training

The `model.fit()` function in keras is used to train the network. It takes parameters like batch_size, that indicates how many entries we want to process at once, epoch_size, that shows how many times we want to go through the input dataset as well as validation_split that shows what fraction of the train data should be left for validation.

But I would like to focus your attention on the callbacks parameter, since it is a very interesting feauture of keras that allows you to specify “monitors"", that will watch the training and do something if their condition is satisfied. For example, you can look at the listing 2.26 and see how I defined an early stopping monitor that monitors `val_loss` with the patience of 5 on line 168. It means, that if our val_loss doesn’t improve for 5 epochs in a row (which shows that adam has found a local minimum), it will stop the training, thus saving us a lot of time. That said, it won’t be beneficial every time, since sometimes the found minimum isn’t ideal and the algorithm can “climb out"" of it.

```
157 def plot_pred(model, X_test, y_test):
158     fig = plt.figure()  # Create a new figure
159     plt.plot(y_test, label='True data')  # Plot the real data
160     y_pred = model.predict(X_test)  # Predict the data using the X_test
161     plt.plot(y_pred, label='Prediction')  # Plot the prediction
162     plt.legend(fancybox=True, shadow=False)  # Add a legend to the plot
163     return fig
```

Listing 2.25: Plotting the results

```
164 for model_fun in [make_one_layer_lstm, make_two_layer_lstm]:
165     model = model_fun()
166     # Stop if we haven't improved for X epochs
167     early_stopping_monitor = EarlyStopping(monitor='val_loss', patience=5)
168     # Fit the model
169     model.fit(
170         X_train,
171         y_train,
172         batch_size=30,
173         epochs=200,  # We can make it so large due to the early stopping monitor
174         validation_split=0.1,
175         shuffle=True,
176         callbacks=[early_stopping_monitor]
177     )
178     # Print the loss on the test set
179     model.evaluate(X_test, y_test)  # Will use the default scalar loss metric
180     # Plot the prediction for the test set
181     plot_pred(model, X_test, y_test)
```

```
182     plt.show()
183     # Save the model to a HDF5 file
184     model.save(f'{model_fun.__name__}_e{epoch_size}.h5')
```

Listing 2.26: Training the network

2.4.5 Results overview

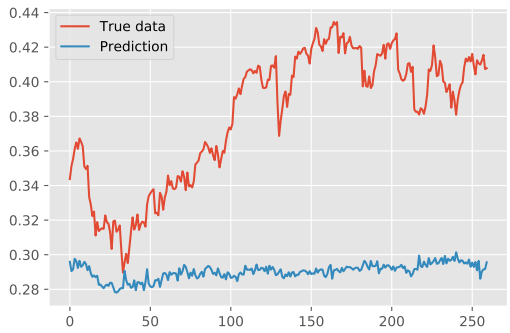

Figure 2.11: One layer network prediction

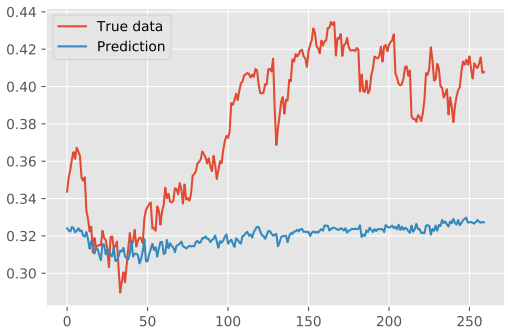

Figure 2.12: Two layer network prediction

Table 2.2: LSTM networks losses

One layer	Two layer
0.00934559645561	0.00476981351654

Looking at the results, it is clear that the a network with two layers has proven to be more accurate than a network with one layers. It is probably due to the large amounts of data that we tried to give to the networks - having two LSTM layers has helped in detecting more intricate dependencies. Of course, the obtained results could have been improved by testing more nodes/layers combinations as well as using different preprocessing techniques. I would also recommend using PCA in the data preprocessing step to reduce the number of input dimensions if your computer can not handle this amount of columns.

Also, please note how to load the model once you have it saved in the listing 2.27. Once you have saved a keras model to an HDF5 file using `model.save()`, you can load it back in and be ready to make predictions by using `load_model()`.

```
185 # Get all model files in a directory
186 models = [i for i in os.listdir() if i.endswith('.h5')]
187 for i in models:
188     temp_model = load_model(i)
```

```
189     # Plotting the model's structure
190     plot_model(temp_model, to_file=f'{i}.pdf', show_shapes=True)
191     # Will use the default scalar loss metric
192     print(i, temp_model.evaluate(X_test, y_test))
193     plot_pred(temp_model, X_test, y_test)  # Plotting the model's prediction
194     plt.savefig(f'prediction_{i}.pdf')  # Saving the prediction plot
```

Listing 2.27: Obtaining the final plots and losses

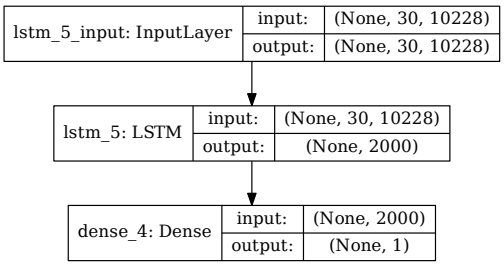

Figure 2.13: One layer network structure

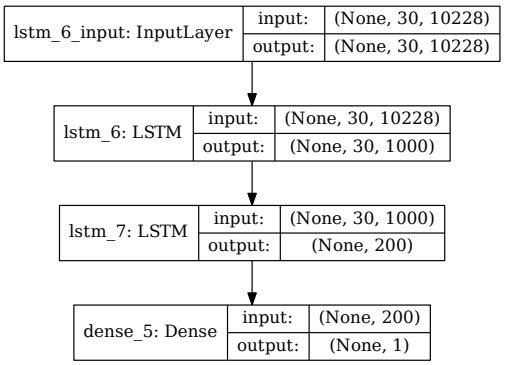

Figure 2.14: Two layer network structure

2.5 Summary

The time series analysis project has accomplished its goals that were focused on showing the reader how to operate and plot the stock data. The data gathering part will allow for the readers to obtain their own data and process it in their own ways. It also showcased the pandas' and numpy's ability to process tabular data, which is a very useful asset that one can utilize. Traditional stock visualizations have also turned out well, and allow one to see the price of a stock over time. As for the correlation matrix, while visualizing it has produced some quite unclear plots, it also showed the reader how to create a heatmap. And I hope that the data prediction section will make more people use neural networks, and especially long short-term memory neural networks in their work, since they are often either the best or in the competition for the title of the best data forecasting tool.

3 GRAPH DATA ANALYSIS

3.1 Preprocessing, igraph creation and community detection

3.1.1 Packages used

Below you can find the imports that will be used in the entire project

```
# Datashader
import datashader as ds
import datashader.layout as ds_l
import datashader.bundling as ds_b
import datashader.transfer_functions as ds_tf

# MPL
from matplotlib import pyplot as plt

# igraph
import igraph as ig
import cairocffi as cairo
import louvain

# Misc
from tqdm import tqdm
import datetime as dt
import random
import pickle
import sys
import os

# Data
import pandas as pd
import numpy as np
```

Listing 3.1: Imports

3.1.2 Pandas dataframe

The source file for this project is a file called “out"" with no extentions. But if you read it using a text editor, you will see that it is just a regular delimited file that uses spaces as a separator and is, essentially, an edge matrix.

Please note how to skip rows (there were comments in the file) and define a custom separator when reading a DataFrame using keyword arguments.

```
FILE_PATH = os.path.join('data', 'youtube-u-growth', 'out')

# Pandas
df = pd.read_csv(FILE_PATH, skiprows=2, sep=' ',
                 names=['source', 'target', 'weight', 'date'])
df['date'] = pd.to_datetime(df['date'], unit='s')  # Converting date to datetime
df.describe(include='all')  # Checking the df. The weight is always 1
df.drop('weight', axis=1, inplace=True)  # Dropping weight because it's const
```

Listing 3.2: Using pandas to load the initial data source

3.1.3 Igraph creation

igraph utilizes its own system of numbering vertices and edges - they must be sequential and start from zero. But the source file starts numbering vertices from one, and has breaks in the numbering, so, for example, vertice 17 can be followed by vertice 25 instead of vertice 18. This means, that we have to rename the vertices and edges ourselves, while maintaining the proper edge information.

In the listing 3.3, I start with creating an empty `igraph.Graph` object. Then I use a `np.ravel()` to flatten out the edge matrix, and then select unique vertice names from it using `pd.unique()` function. After obtaining this list of vertice names, I can use its length to create a continuous sequence of numbers starting with zero using `np.arange(start, end)` that will replace the vertice names. I proceed by adding that sequence to a graph by using the `add_vertices` function.

I have to map vertice numbers from the edge matrix to the new continuous list to be able to import it into igraph. First step in doing that is to use builtin `dict` and `zip` functions to create a dictionary where a vertice real name is the key and the sequential number is the value. As you can see from the listing, `zip(it1, it2)` function is very useful - it takes two iterators as an input and then outputs a iterator that returns tuples `(it1[i], it2[i])` when called an ith time. In here, I have used the `dict` function to instantly convert that tuple iterator to a disctionary. Then, I use the `np.vectorize(function)` function, that converts a function from a one that takes one element of an array as an argument to a one that takes the entire array as an argument, and executes the old function on every element of that array. I have also used a lambda expression here. A lambda expression in Python is a short-hand way to define a function that doesn't require using `def` and proper spacing. But, since it is only a short-hand after all, you can't execute more than one command using the lambda expression. Thankfully, this time we

only have to look up the old vertice name in the dictionary and get the new name out of it, and a dictionary lookup only takes one command. After executing the vectorized function on the edge matrix, the result are ready to be imported into igraph, which I do by using the `add_edges()` function.

```
# Converting dataframe to the igraph's graph - Preparations
# Creating a blank graph
g = ig.Graph()

# Creating arrays for edges, vertices and edge attributes
edge_df = df.loc[:, 'source':'target']
edge_matrix = edge_df.as_matrix()
vertices_unique = pd.unique(edge_matrix.ravel('K'))

# The ids are starting from zero
vertice_array = np.arange(0, vertices_unique.shape[0])
print('Created a blank graph and element arrays')

# Adding vertices
g.add_vertices(vertice_array.tolist())
print('Added vertices')

# Changing the edges from names to ids
vertice_ids = dict(zip(vertices_unique, vertice_array))
rename_edges = np.vectorize(lambda x: vertice_ids[x])
edge_matrix = rename_edges(edge_matrix)
print('Renamed edges in the edge_matrix to their ids')

# Adding edges
g.add_edges(edge_matrix)
print('Added edges')

# Adding edge attributes
g.es['date'] = df['date'].as_matrix()
print('Added edge attributes')
print('Displaying summary')
ig.summary(g)
```

Listing 3.3: Creating an igraph

We can also check whether the data imported correctly using the code from listing 3.4. I would also recommend looking into more complicated solutions like pytest if you are working on a project that involves multiple people, since it makes it harder to make mistakes when introducing new code.

```
def test_edge_renaming():
    # Convert the source dataframe to a numpy
```

```
    orig_edge_matrix = df.loc[:, 'source':'target'].as_matrix()
    for orig_edge in orig_edge_matrix:
        try:
            test = g.get_eid(vertice_ids[orig_edge[0]],
                             vertice_ids[orig_edge[1]])
        except:
            print(f'Edges renaming test failed - edge {orig_edge} is missing')
            return False
    print('Edges renaming test passed!')
    return True


print(f'Edge count is correct: {len(df) == len(g.es)}')
print(f'Vertice count is correct: {len(vertices_unique) == len(g.vs)}')
print(test_edge_renaming())
```

Listing 3.4: Verifying the transformation

```
Edge count is correct: True
Vertice count is correct: True
Edges renaming test passed!
```

Figure 3.1: Check outputs

3.1.4 Community detection

You can see the community detection code in the listing 3.5 below. It uses the louvain algorithm to partition the initial graph, and then switches to infomap to partition the smaller one. This is done this way, because the louvain algorithm was specifically developed and optimized for large graphs, while infomap requires us to perform some computationally expensive operations like walking the graph tree to find the communities.

```
# Finding communities
# Partitioning the large graph using louvain
partition = louvain.find_partition(g, louvain.ModularityVertexPartition)
louvain_graph = partition.cluster_graph()
ig.summary(louvain_graph)

# Partitioning the louvain graph using infomap
imap = louvain_graph.community_infomap()
ig.summary(imap)

# Saving the new graphs to pickle objects
with open('louvain_graph.pickle', 'wb') as f:
    pickle.dump(louvain_graph, f)
```

```
71 with open('imap.pickle', 'wb') as f:
72     pickle.dump(imap, f)
73
74 # Following lines can be used to retrieve the saved files on the following runs
75 # with open('louvain_graph.pickle', 'rb') as f:
76 #     louvain_graph=pickle.load(f)
77 # with open('imap.pickle', 'rb') as f:
78 #     imap = pickle.load(f)
```

Listing 3.5: Detecting communities

3.2 Plotting large graphs (datashader)

3.2.1 Simple plot

Datashader introduction

Datashader is a package that allows the user to visualize large amounts of data. It uses a lot of different aggregation techniques in order to generate plots that, while showing a lot of data, are possible to process and show.

Description

In this subsection, I have decided to try and plot the whole graph of the YouTube friends netwrok using datashader. I started with defining the following plot functions: plots_edges, plot_vertices and plot_full that you can see in the listing 3.6. All of them start out by creating a `ds.Canvas` object, that will be used as a matplotlib figure by datashader - all plots, points, lines will be plotted on that canvas.

As for what we will put on that canvas, Datashader works as follows: there are two types of graph functions you can call - a bundling function or a layout function. A bundling function is responsible for the edge placement, and the layout - for vertices. The bundling function is called that way, because datashader allows to aggregate edges into bundles, and we will cover that in the next subsection.

For the edge plot, I expect to receive that output of the bundling function that I will then just have to pass to the `canvas.line` function in order to plot the edges. And `ds_tf.shade` is used as a compile function - once you shade your points, they are converted to an RGBA image.

For vertices it's the opposite - I expect to receive the output of the layout function that I can later pass to a `canvas.points` function. The spread function in here helps to make vertices more visible by extending every pixel occupied by a vertice by a predefined value.

```
def plot_edges(edges, name=None, plot_height=500, plot_width=500):
    canvas = ds.Canvas(plot_height=plot_height, plot_width=plot_width)
    return ds_tf.shade(canvas.line(edges, 'x', 'y', agg=ds.count()), name=name)


def plot_vertices(verts, name=None, cat=None, plot_height=500, plot_width=500, spread=0):
    canvas = ds.Canvas(plot_height=plot_height, plot_width=plot_width)
    aggregator = None if cat is None else ds.count_cat(cat)
    agg = canvas.points(verts, 'x', 'y', aggregator)
    # Vertices will be plotted in red
    return ds_tf.spread(ds_tf.shade(agg, name=name, cmap=['#ff0000']), px=spread)


def plot_full(verts, edges, name='', cat=None,
              plot_height=500, plot_width=500, spread=0):
    # Determining plot
    xr = verts.x.min(), verts.x.max()
    yr = verts.y.min(), verts.y.max()
    canvas = ds.Canvas(plot_height=plot_height,
                       plot_width=plot_width, x_range=xr, y_range=yr)
    # Aggregating by gategory is possible
    aggregator = None if cat is None else ds.count_cat(cat)
    agg = canvas.points(verts, 'x', 'y', aggregator)
    # Plotting vertices
    np = ds_tf.spread(ds_tf.shade(agg, name=name + ' verts',
                                  cmap=['#ff0000']), px=spread)
    # Protting edges
    ep = ds_tf.shade(canvas.line(
        edges, 'x', 'y', agg=ds.count()), name=name + ' edges')

    return ds_tf.stack(ep, np, how="over", name=name)
```

Listing 3.6: Datashader plotting functions

In the full graph example I am using the random layout and directly connected edges bundling (in another words, no bundling at all) because when I have tried to run the bundling and a more complicated layout calculations on my data science server, it ran for three days and then just crashed. I assume that this graph was a tad too big for datashader to process.

```
# Creating vertex/edge dataframes
g_df_vert = pd.DataFrame(g.vs.indices, columns=['name'])
g_df_edge = pd.DataFrame(edge_matrix, columns=['source', 'target'])

# Trying to plot the entire graph. Using the random layout because the fg one crashed the server
g_layout_fg = ds.layout.random_layout(g_df_vert, g_df_edge, seed=55)
g_bundling_fg = ds_b.directly_connect_edges(g_layout_fg, g_df_edge)
```

Listing 3.7: Creating datashader layouts

To plot these images in this case I've used a simple `ds_tf.Image function`, that just displays an image. I will cover how to save them to a file in a next subsection.

```
39 # Edge plot
40 g_plot = plot_edges(g_layout_fg, g_bundling_fg)
41 ds_tf.Image(g_plot)
42
43 # Full plot
44 g_plot = plot_full(g_layout_fg, g_bundling_fg)
45 ds_tf.Image(g_plot)
```

Listing 3.8: Plotting the whole plot using datashader

Results

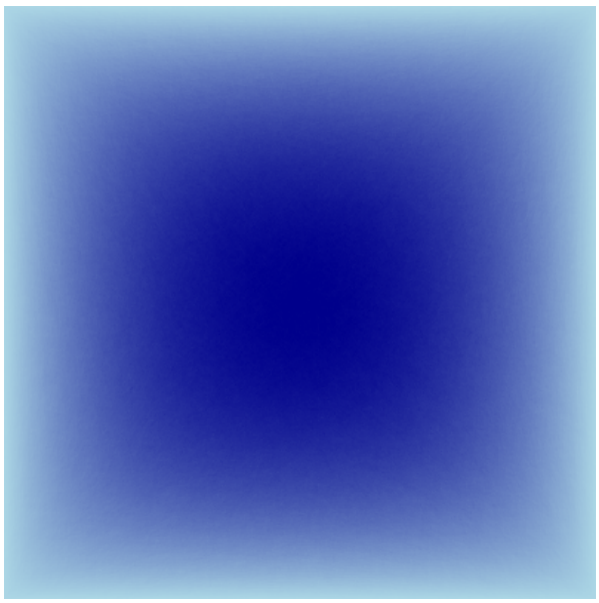

Figure 3.2: Edge plot of the full graph

Figure 3.3: Full plot of the full graph

As you can see, no indormation can be obtained from the above figures. And while datashader is certainly capable of plotting a huge amounts of data points, sometimes using that capability is a bad idea and only leads to dissappointing results.

3.2.2 Regular plot

Description

In this subsection, I will show you how to plot a quite large subgraph (26000 vertices) of the YouTube graph using datashader. I am using the communities obtained by the louvain algorithm as vertices. I will also compare different layout and bundling combinations.

Creating a plot list

While the last subsection's results were quite dissappointing, the plotting functions used there are correct. Since in this subsection I will be comparing a lot of different layouts and bundlings, I have decided to create a list of these plots, and then just save them using a for loop. As you can see, I started with creating a one of every layout and then created a one of every bundling for those layouts. Then, I have created a list that includes every layout/bundling combination by just calling the plot functions from inside of the list declaration.

```
# Creating the vertice/edge dfs
lv_df_vert = pd.DataFrame(louvain_graph.vs.indices, columns=['name'])
lv_df_edge = pd.DataFrame([(e.source, e.target)
                           for e in louvain_graph.es], columns=['source', 'target'])

# Force directed layout
lv_layout_f = ds.layout.forceatlas2_layout(lv_df_vert, lv_df_edge)
lv_bundling_f = ds_b.hammer_bundle(lv_layout_f, lv_df_edge)
lv_bundling_dc_f = ds_b.directly_connect_edges(lv_layout_f, lv_df_edge)

# Random layout
lv_layout_r = ds.layout.random_layout(lv_df_vert, lv_df_edge)
lv_bundling_r = ds_b.hammer_bundle(lv_layout_r, lv_df_edge)
lv_bundling_dc_r = ds_b.directly_connect_edges(lv_layout_dc_r, lv_df_edge)

# Circlular layout
lv_layout_c = ds.layout.circular_layout(lv_df_vert, lv_df_edge)
lv_bundling_c = ds_b.hammer_bundle(lv_layout_c, lv_df_edge)
lv_bundling_dc_c = ds_b.directly_connect_edges(lv_layout_c, lv_df_edge)
```

Listing 3.9: Creating layouts

```
# Making the plots
lv_plots = [
    # Force-directed layout
    # Directly connected
    plot_vertices(
        lv_layout_f, 'Vertices - Force-directed - Directly connected'),
    plot_edges(lv_bundling_dc_f, 'Edges - Force-directed - Directly connected'),
    plot_full(lv_layout_f, lv_bundling_dc_f,
              'Full - Force-directed - Directly connected'),
    # Bundled
    plot_vertices(lv_layout_f, 'Vertices - Force-directed - Bundled'),
    plot_edges(lv_bundling_f, 'Edges - Force-directed - Bundled'),
    plot_full(lv_layout_f, lv_bundling_f, 'Full - Force-directed - Bundled'),
    # Random layout
    # Directly connected
    plot_vertices(lv_layout_r, 'Vertices - Random - Directly connected'),
```

```
    plot_edges(lv_bundling_dc_r, 'Edges - Random - Directly connected'),
    plot_full(lv_layout_r, lv_bundling_dc_r,
              'Full - Random - Directly connected'),
    # Bundled
    plot_vertices(lv_layout_r, 'Vertices - Random - Bundled'),
    plot_edges(lv_bundling_r, 'Edges - Random - Bundled'),
    plot_full(lv_layout_r, lv_bundling_r, 'Full - Random - Bundled'),
    # Circular layout
    # Directly connected
    plot_vertices(lv_layout_c, 'Vertices - Circular - Directly connected'),
    plot_edges(lv_bundling_dc_c, 'Edges - Circular - Directly connected'),
    plot_full(lv_layout_c, lv_bundling_dc_c,
              'Full - Circular - Directly connected'),
    # Bundled
    plot_vertices(lv_layout_c, 'Vertices - Circular - Bundled'),
    plot_edges(lv_bundling_c, 'Edges - Circular - Bundled'),
    plot_full(lv_layout_c, lv_bundling_c, 'Full - Circular - Bundled')
]
```

Listing 3.10: Creating a list with layout combinations

Saving the plot

In order to save the plots, I have to iterate through the previously created list of layouts and us the `export_image` function from datashaders.utils package to save the plots. Please not, how to use it, since it is much more convenient to save images using a specialized function than by saving an image from your jupyter notebook or just a regular python output. I have also used tqdm to show the progress of saving the images.

```
# Saving the images
for plot in tqdm(lv_plots, file=sys.stdout, desc='Plots saves:', unit='plot'):
    _ = ds.utils.export_image(plot, f'{plot.name}', fmt='.png')
```

Listing 3.11: Saving the plots

Results

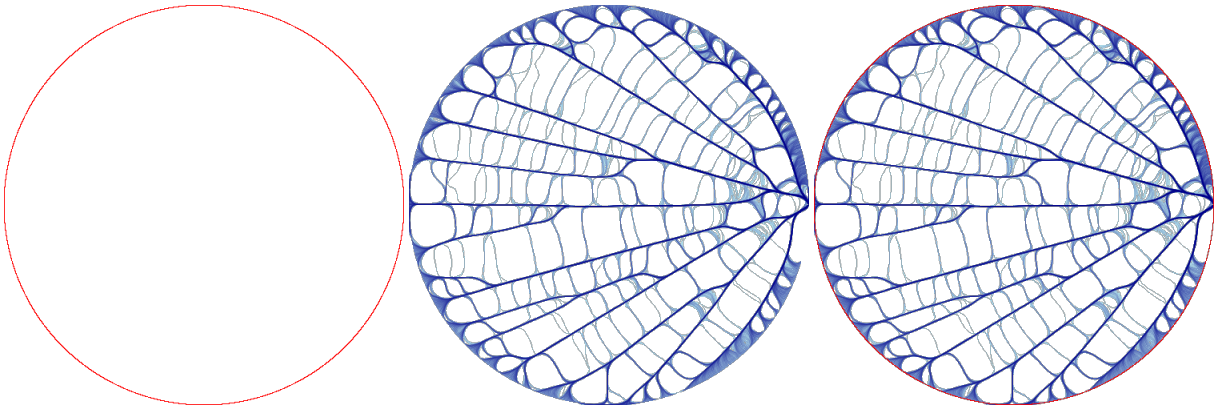

Figure 3.4: Circular layout - Bundled

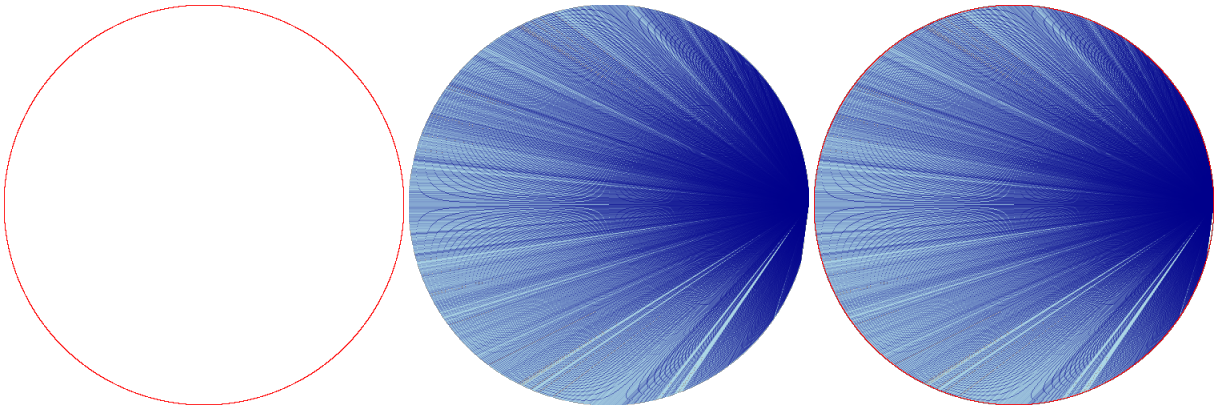

Figure 3.5: Circular layout - Directly connected

I would say that the circular layout overall is a quite bad way to represent large graphs, since you can't really see the connections between nodes once they are in a circle. While it may work with very small graphs, in here they are just pictures of a filled circle.

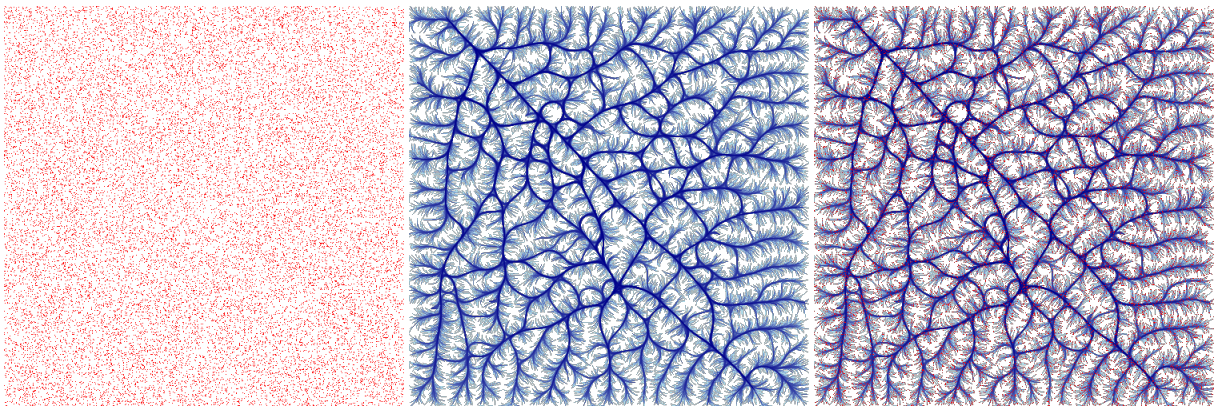

Figure 3.6: Random layout - Bundled

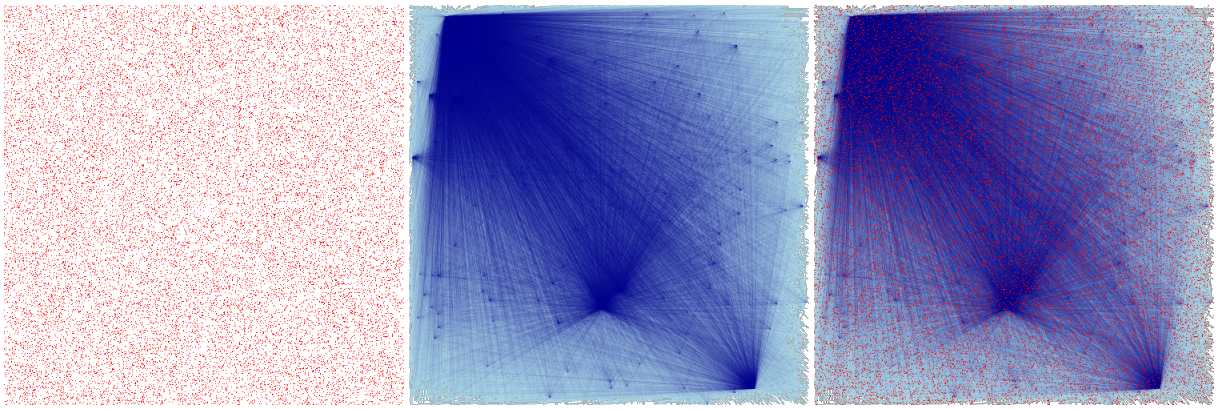

Figure 3.7: Random layout - Directly connected

The random layout looks a bit better than the circle layout, and you can even see some of the more highly-connected vertices in the directly connected bundling. I am actually quite satisfied with the directly connected version just for this fact. And I actually think that bundling is detrimental to this plot, since all it does is hides those highly connected points of interests.

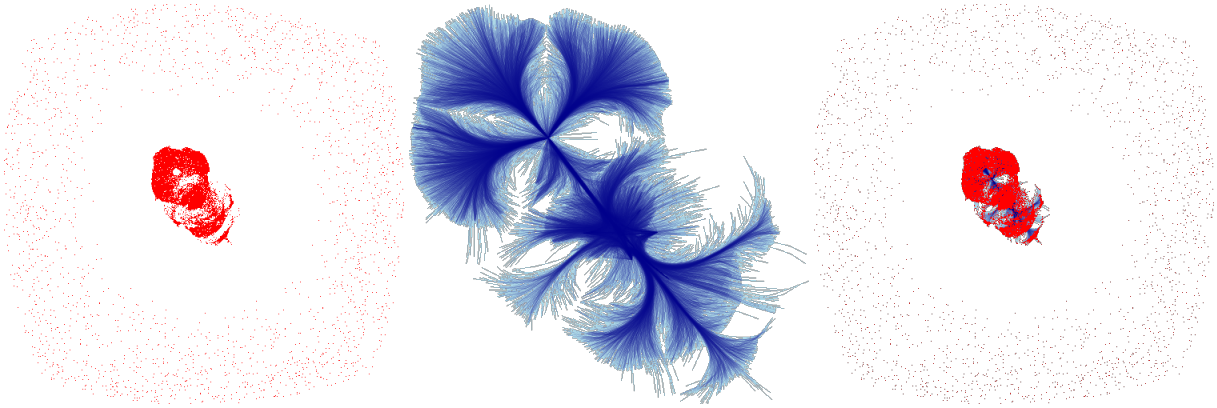

Figure 3.8: Force-directed layout - Bundled

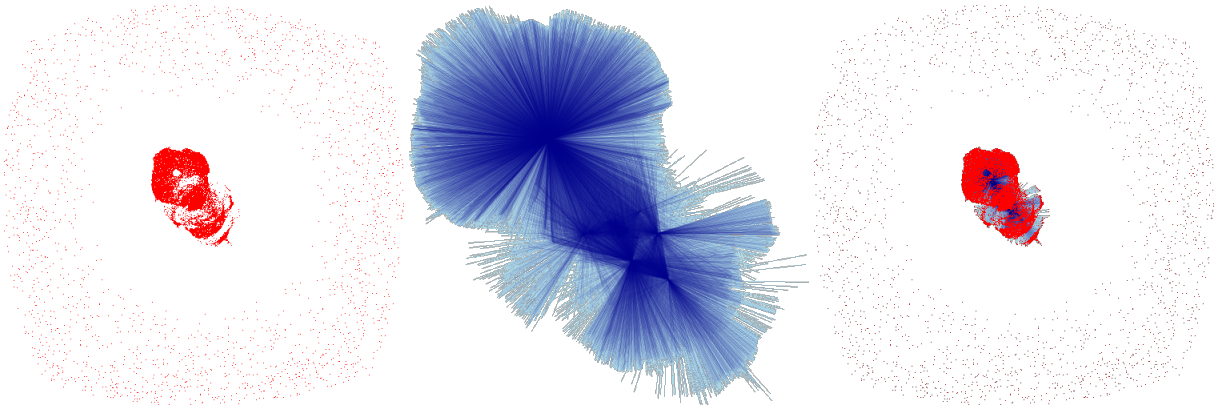

Figure 3.9: Force-directed layout - Directly connected

The force-directed layout (which is also known as Forced Atlas 2 [3]) looks the best to me

in this comparison, since it clearly shows the points with the most edges directed to/from them. The only problem that I can see is that in the full plot it pushes the unconnected vertices too far away, which makes it very hard to see the graph itself.

3.2.3 Plotting communities

```
lv_df_comm = pd.DataFrame({'name': louvain_graph.vs.indices,
                           'comm': imap.membership})
lv_df_comm.comm = lv_df_comm.comm.astype('category')
# ~2000 communities only have 1 member
print(lv_df_comm.comm.value_counts()[:15])

# Preparing the necessary lists
# Keep all communities with more than one member
comms_to_keep = [ind for ind, val
                 in enumerate(lv_df_comm.comm.value_counts())
                 if val > 1]

# len(...) is one more than the max value, since comms_to_keep starts with 0
# Create a new community for uncategorized
new_community_id = len(comms_to_keep)

# A list that will be used to replace the current community column
new_comms = []

for comm in lv_df_comm.comm:
    if comm in comms_to_keep:
        new_comms.append(comm)
    else:
        new_comms.append(new_community_id)

# Assigning it to to the df
lv_df_comm.comm = pd.Series(new_comms, dtype='category')

# Plotting the new category distribution
lv_df_comm.comm.value_counts().plot('bar')
plt.show()
```

Listing 3.12: Merging uncategorized vertices

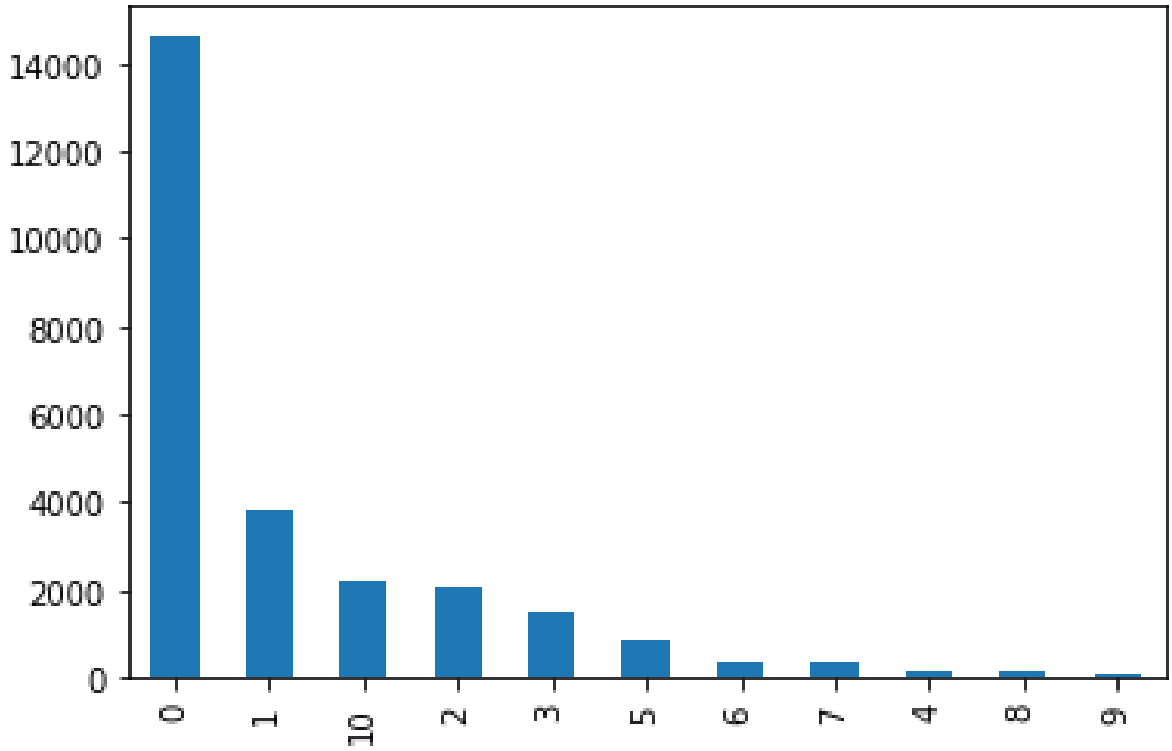

Figure 3.10: Communities after the merge

```
32 # Preparing the layouts
33 lv_layout_comm = ds.layout.forceatlas2_layout(
34     lv_df_comm, lv_df_edge, linlog=True)
35 lv_bundling_comm = ds_b.hammer_bundle(lv_layout_comm, lv_df_edge)
36
37 # Spread
38 ds_tf.Image(plot_full(lv_layout_comm, lv_bundling_comm, 'Full - Force-directed - Categorized',
39                       cat='cat', spread=1, plot_height=1000, plot_width=1000))
40
41 # No spread
42 ds_tf.Image(plot_full(lv_layout_comm, lv_bundling_comm, 'Full - Force-directed - Categorized',
43                       cat='cat', spread=1, plot_height=1000, plot_width=1000))
```

Listing 3.13: Plotting the communities

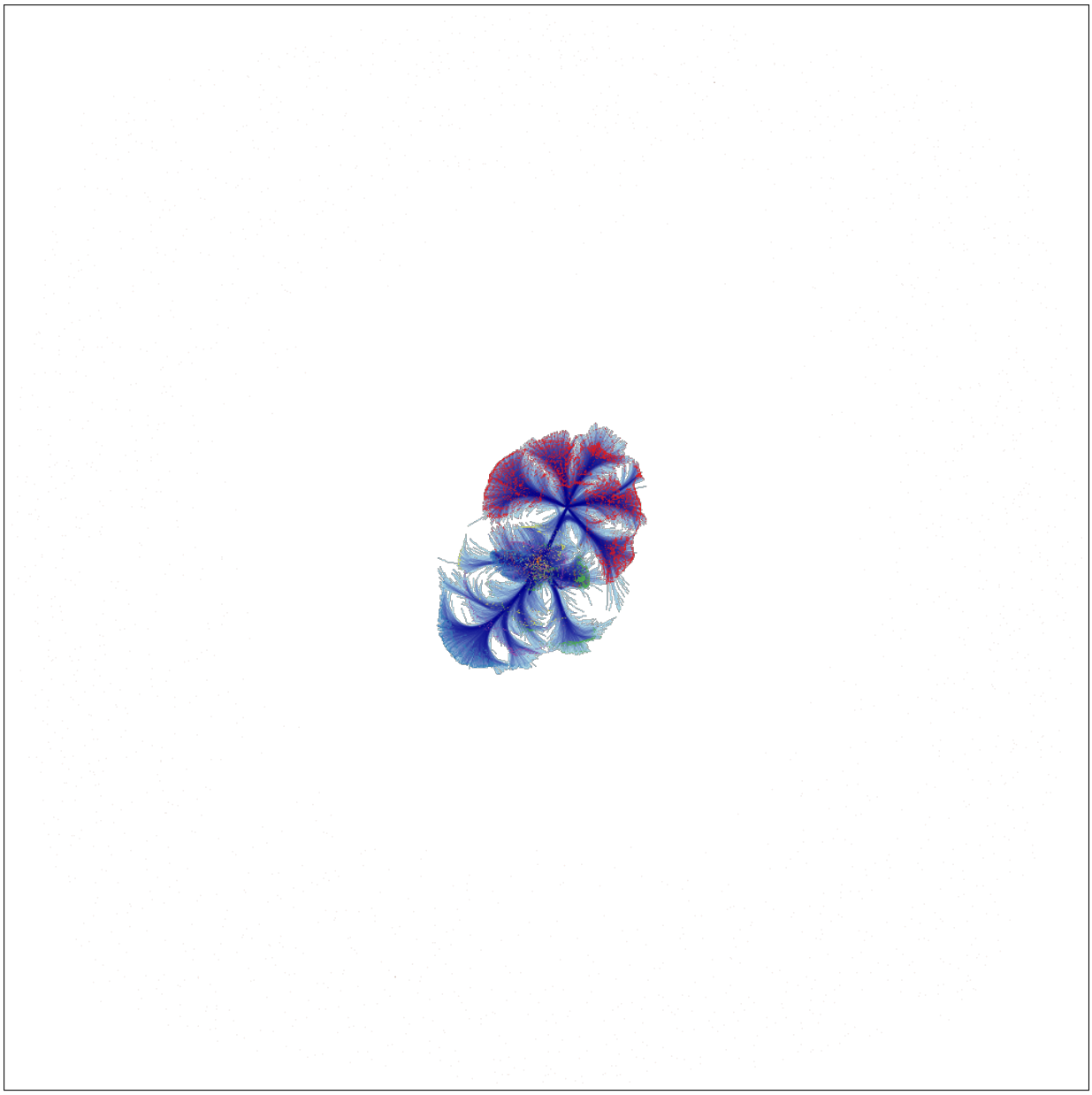

Figure 3.11: Plot without pixel spreading

In the plot with pixel spreading, one can see the nodes without any edges that encircle the main graph.

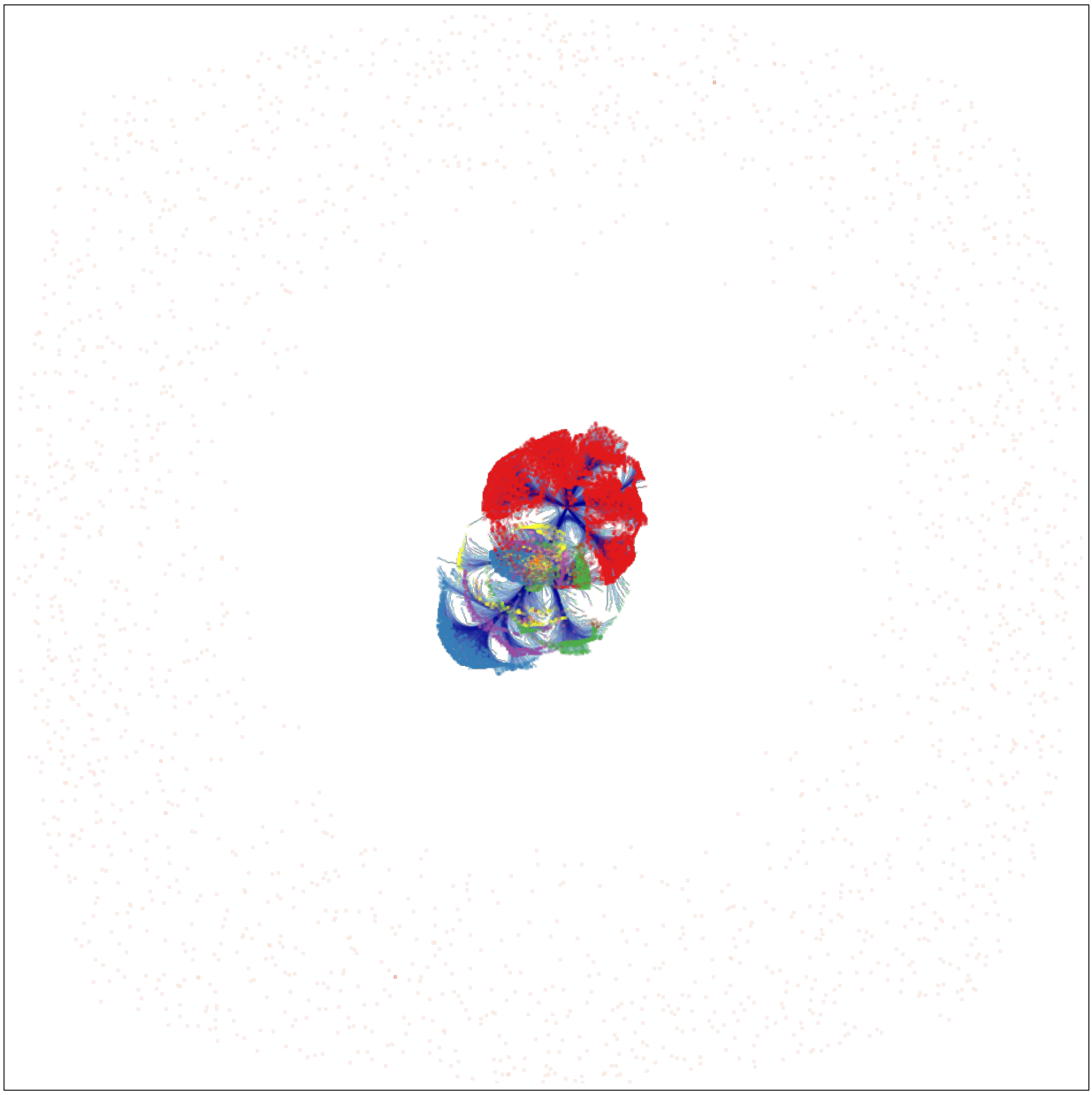

Figure 3.12: Plot with pixel spreading

3.3 Plotting smaller graphs (igraph)

3.3.1 Cairo library

In this section, I will show you how to plot a smaller (around 1000 nodes) portion of our graph using tools that are supported by igraph, in particular using its bindings to a C library called cairo. Since it is originally written in C and only provides an API that other languages can use, we will also have to use a package that will be able to connect python code to the C library itself. In this example, I'm using cairocffi as such package, hence the inclusion of `import cairocffi as cairo` in the starting import list. We have to import it as cairo, because otherwise igraph will

attempt to use a different binding package, `pycairo`, which is quite outdated and can outright refuse to save vector images.

3.3.2 Regular plot

Selecting vertices

First thing that we have to do in this section is to select a subgraph for plotting. I have decided to settle on the subgraph of vertices with high degrees, because that meant that every node would be connected to many other nodes, and that can be a good example of how to deal with clutter on the plot.

In order to select this subgraph, you can use the `graph.vs.select(*args, **kwargs)` method in your graph. This method works differently based on what you pass it:

1. If you pass a list of integers into it `select([1,2,3])`, or skip the list and call `select(1,2,3)`, it will return vertices at those indexes
2. If you pass a special keyword argument to it, it will select all nodes with a property that match that argument
3. If you pass a function to it, it will call that function on every vertex and return all vertices that the function has returned `True` for
4. If you don't pass anything into the function, it returns an empty list

The select method has 8 special keyword arguments:

- `eq` - equal to
- `ne` - not equal to
- `lt` - less than
- `gt` - greater than
- `le` - less than or equal to
- `ge` - greater than or equal to
- `in` - value is in the given list
- `notin` - value is not in the given list

Please note, that you have to include the name of your property before the special keyword: `graph.vs.select(age_in=[19,20,21])`. You can see how I have used `gt` in the following example. The `_degree` you see in front of it is a semi-private variable (since no variable is truly private in python) that keeps track of the degree of the vertex.

```python
# Selecting high degree nodes
hdg_vertices = g.vs.select(_degree_ge=800)  # Select vertices with a high degree
hdg_subgraph = hdg_vertices.subgraph()  # Create a new subgraph
hdg_vcount = hdg_subgraph.vcount()  # Will be used later in the layout calculation
# Check the number of vertices
print(f'Number of vertices in the subgraph: {hdg_vcount}')
```

Listing 3.14: Selecting nodes for the subgraph

Styling the resulting plot

There are many different options to choose from if you want to change how the graph appears on the screen. They are originally available as keyword arguments that you can pass to the `ig.plot()` function, but I advocate for putting all of them in a separate dictionary, since it takes away from the dissaray of having to include many options into one function call.

In this paper I will mainly focus on the layout option, as well as gloss over a couple of settings connected to the size of the vertices and edges, but if after reading this you will want to learn more about them, you can call `help(ig.plot)` while in a python interpreter to see the function's docstring.

One of the most important arguments provided to us is the layout one, since it changes how nodes and edges are positioned in the resulting plot. It is possible to choose from the following layouts[3]:

- Circle layout
- Star layout
- Grid layout
- Fruchterman Reingold layout [9]
- Fruchterman Reingold grid layout
- DrL layout [17]
- Graphopt layout [29]
- Kamada Kawai layout [14]
- Sugiyama layout [30]
- Random layout
- Large Graph layout
- Reingold Tilford layout (for trees) [27]
- Bipartite layour (for 2-layer graphs)

As you can see in the listing, other options are responsible for things like edge width, vertex size and shape, where to save the file, et cetera.

[3]Visual comparison is available on the pages 55-57

The `bbox` argument is responsible for how big your final plot is going to be (in another words, it is declaring a limiting box on a figure that no vertex can cross).

The `target` keyword argument is also very useful - it specifies the name of the file that your final chart will be stored in, and supports multiple file extentions (PDF, SVG, PNG). In the next section I will also explain you how to add a color palette to the dictionary in order to distinguish the communities that we will detect.

```
# Setting up the plot
hdg_style = {}  # Creating a style dictionary
hdg_style['layout'] = \
    hdg_subgraph.layout_fruchterman_reingold(maxiter=1000, area=hdg_vcount**3)
hdg_style['vertex_size'] = 0.001
hdg_style['bbox'] = (1024, 1024)
hdg_style['vertex_shape'] = 'circle'
hdg_style['edge_width'] = 0.1
hdg_style['target'] = 'plot_naive.svg'
```

Listing 3.15: Creating a style dictionary

Plotting and saving the resulting plot to a file

```
# Plotting
ig.plot(hdg_subgraph, **hdg_style)
```

Listing 3.16: Plotting the image

Since we have already constructed the keyword argument dictionary for the plot function, the rest becomes very clean and easy - we just have to pass the dictionary to the function while remembering to unroll it to convert it to key-value pairs.

Layout comparison and revision of the results

All of the plots generated in this section use the aforementioned style dictionary, while only changing the `['layout']` part of it.

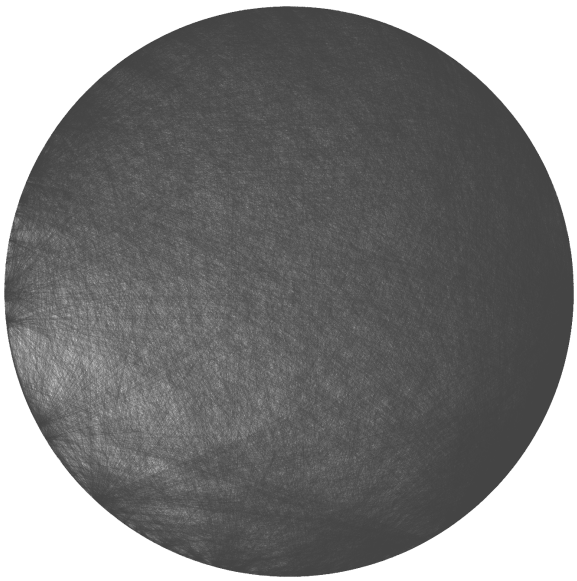

Figure 3.13: Circular layout

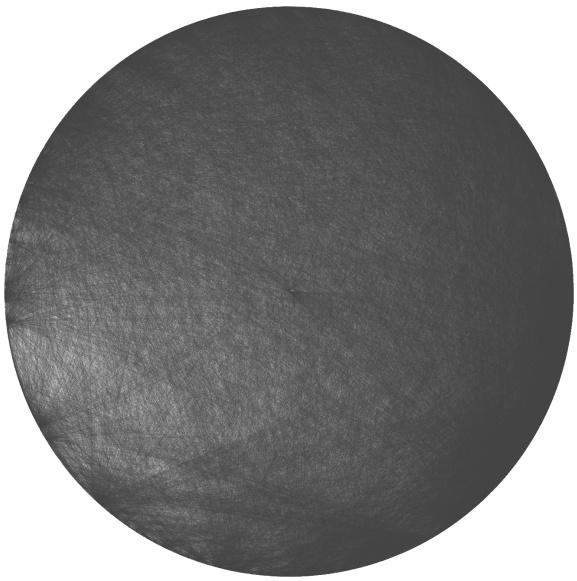

Figure 3.14: Star layout

Circular layout puts all vertices on a circle and then draws the edges between them, while the star layout puts one of the vertices in the middle an tries to center the plot around it. As you can see from the above figures, in this case they are almost non-distinguishable, and don't show any insights about the graph.

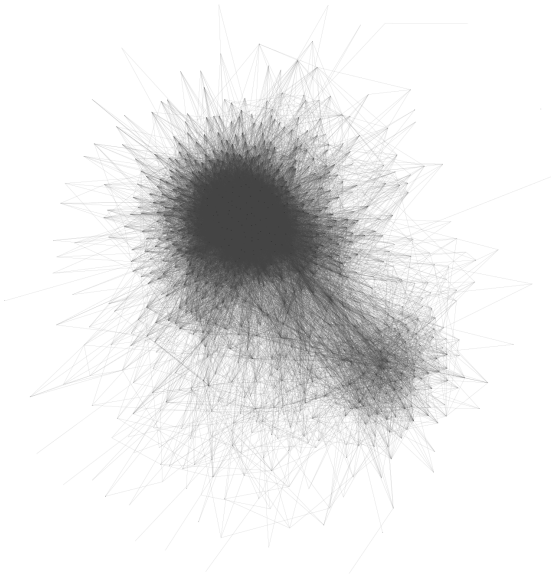

Figure 3.15: Fruchterman Reingold layout

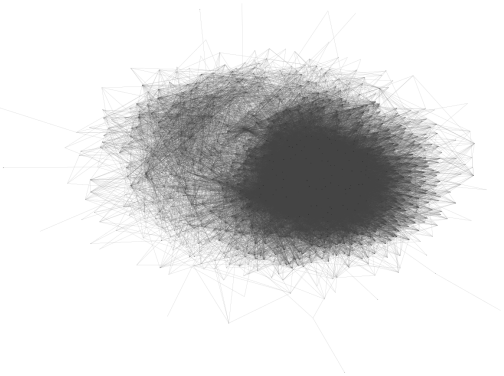

Figure 3.16: Kamada Kawai layout

Figure 3.17: DrL layout

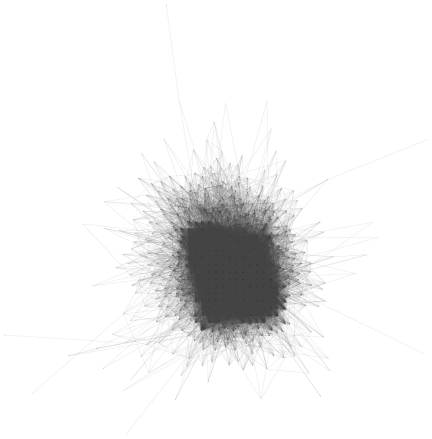

Figure 3.18: Graphopt layout

Force-directed algorithms like Fruchterman Reingold, Kamada Kawai, Graphopt and DrL all use physics simulations to generate a plot. In this case, it led to Kamada Kawai and Graphopt making a plot that is mostly centered around one point, while Fruchterman Reingold and DrL layouts drew the graph by using two centers, thus highlighting two major communities within it. Unfortunately, DrL layout also spread the vertices too far away from each other, making the whole plot seem like a regular line.

Grid layouts put the vertices on an imaginary mesh, and then draw the connections between them. They can be useful for smaller graphs, but as you can see from the figures below, one can hardly infer any information from these if the amount of points is large enough.

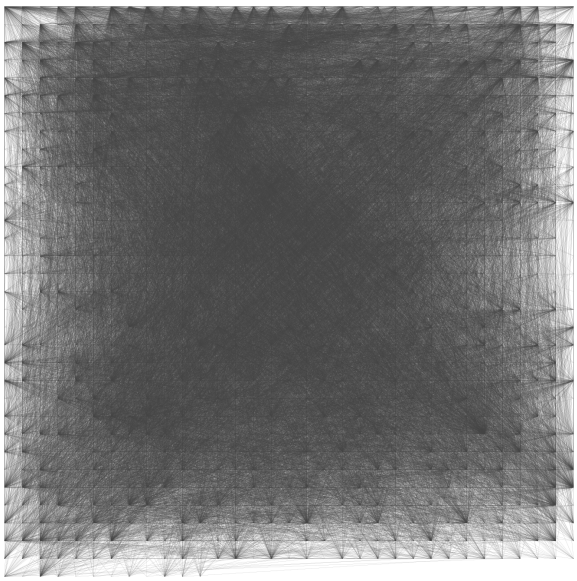

Figure 3.19: Grid layout

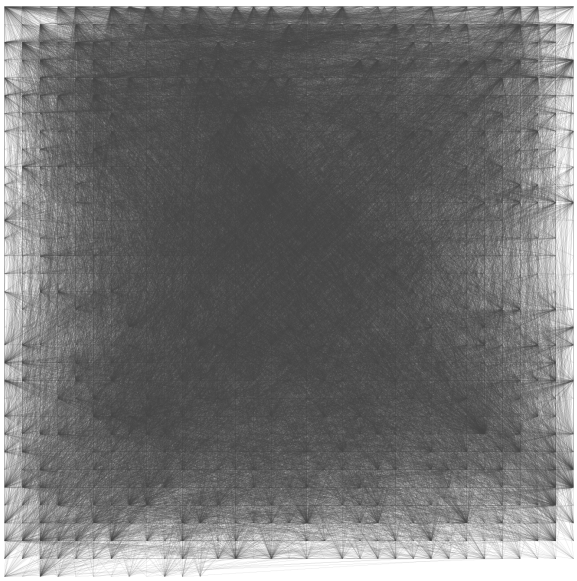

Figure 3.20: Fruchterman Reingold grid layout

Random layout, as its name suggests, places vertices on random points in the plot and then proceeds to draw the edges between them. Sugiyama layout tries to put vertices on different rows of the plot while directing their edges downwards, and is most suited for trees and not a social network graph like the one in this example.

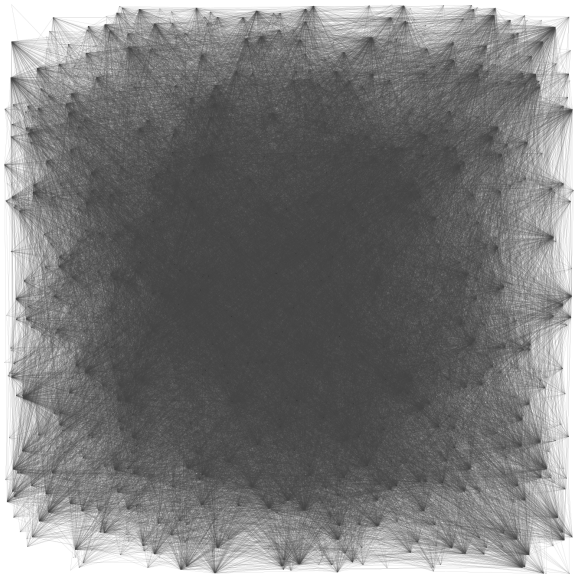

Figure 3.21: Random layout

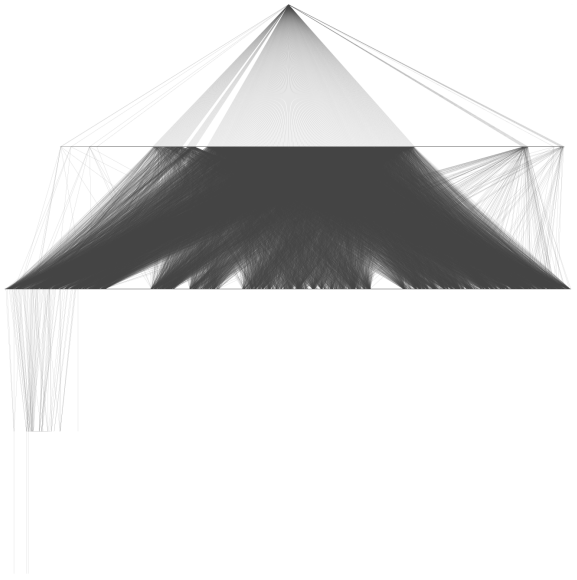

Figure 3.22: Sugiyama layout

While igraph makes it quite easy to make regular plots like these, they don't provide one with much insight into the data itself. Even though you could look at the picture made using the Fruchterman Reingold layout, and detect two major communities in the network with a naked eye, other layouts aren't as simple to read. Still, I find that force-directed layouts like Fruchterman Reingold represented the social network graph better than other layouts.

3.3.3 Community plot

Clustering the subgraph

The size of the subgraph we're clustering is smaller than the one used in the datashader example, so using infomap right away without needing to cluster it with the louvain algorithm in advance is reasonable. In here I'm also storing the node's membership list into a variable, because we will need it later to color edges.

```
1 # Clustering the high degree subgraph
2 hdg_imap = hdg_subgraph.community_infomap()
3 hdg_membership = hdg_imap.membership
```

Listing 3.17: Using infomap to cluster the subgraph

Making communities visible

There are a couple of ways to make communities in your graph more visible on the resulting plot. You could (1) use color to distinguish between them, (2) draw vertices from one community close to each other, (3) separate communities by drawing their boundaries, or (4) label each vertice with their community label. Some of those techniques are only effective when applied to very small graphs (like labeling), while other are a better fit for a medium-sized graph like the one used in this example.

In this example, I have decided to color all vertices within a community and edges between them using one color and to assign very heavy weights to them, and used a more neutral color for edges between vertices that belong to two different communities as well as assigning a very light weight to them. This guarantees that when I will run the fruchterman reingold layout algorithm, the communities will be pulled apart from each other, while vertices in a community will remain together.

As for colors, since the number of communities changed whenever I ran the infomap algorithm, I have decided to go with a randomized approach and just generate a random color for each community using a list comprehension and the `random.randint(min, max)` function

that would give me a color-representing number that then would be converted to hex using the `06x` string format.

```
# Selecting random colors for groups using a list comprehension with an f-string
community_colors = [f'#{random.randint(0, 0xFFFFFF):06x}' for comm in hdg_imap]
# Creating a list that will be used to assign color to every vertice
vert_colors = list(range(hdg_subgraph.vcount()))
# Initializing lists that will hold the edge attributes
edge_colors = []
edge_weights = []
```

Listing 3.18: Initializing color and weight lists

To assign color to a vertice in igraph, you can just add an attribute "color"to the vertex, and igraph with cairo will fill that vetice with that color if it is in the pallette that you have to assign in the keyword argument (or style) dictionary. The enumerate function adds an id to the every member of an iterable that you pass into it, so I used it to get access to the community ID numbers, since the `comm` variable is just a list of vertices. Then I proceeded to iterate through every vertice and assign a matching color to them.

```
# Assigning the vertice color based on their community
for comm_id, comm in enumerate(hdg_imap):
    for vert in comm:
        vert_colors[vert] = community_colors[comm_id]
```

Listing 3.19: Assigning color to vertices

I did a very similar thing to the edges, except this time I have checked whether both of their ends are in one community and assigned different color and weights based on that.

```
# Assigning the edge color and weight based on the vertices it connects
# Adding weights will make the group separation more visible
for edge in hdg_subgraph.es:
    if hdg_membership[edge.source] == hdg_membership[edge.target]:
        edge_colors.append(vert_colors[edge.source])
        edge_weights.append(3 * hdg_vcount)
    else:
        edge_colors.append('#dbdbdb')
        edge_weights.append(0.1)
```

Listing 3.20: Assigning color to edges

Styling the resulting plot

In order for igraph to be able to use a color, it has to be in it's palette. You can either use standard colors from the standard palette or create your own PrecalculatedPallette, passing the color list into the function (color can be specified using hex, rgb or their english names like "red"). So, while the edges' and vertices' colors are decided based on their color attribute, the palette that those colors will be drawn from have to be specified in the style dictionary or as a regular keyword argument. You can also use the `mark_groups` option if you either don't want to color the vertices or edges yourself or just want to make sure that every group is clearly delineated and is very visible[4].

One more new thing in this code listing is the `edge_order_by` argument, and it is quite important for this example, since it determines the order that the edges are drawn in. So, if we order them by weight that means that the the gray edges between communities that have lighter weights will be drawn first, and the heavier and more colorful edges inside of the communities will be drawn on top of them.

```
24 # Styling the plot - graph properties
25 hdg_subgraph.vs['color'] = vert_colors  # Adding color as a vertice property
26 hdg_subgraph.es['color'] = edge_colors  # Adding color as an edge property
27 hdg_subgraph.es['weight'] = edge_weights  # Adding weight as an edge property
```

Listing 3.21: Styling using graph properties

```
28 # Styling the plot - style dictionary
29 hdg_comm_style = {}
30 hdg_comm_style['layout'] = \
31     hdg_subgraph.layout_fruchterman_reingold(maxiter=1000,
32                                              weights=hdg_subgraph.es['weight'],
33                                              area=hdg_vcount**3,
34                                              repulserad=hdg_vcount**3)
35 hdg_comm_style['bbox'] = (1024, 1024)
36 hdg_comm_style['vertex_size'] = 0.5
37 hdg_comm_style['vertex_shape'] = 'circle'
38 hdg_comm_style['edge_width'] = 0.1
39 hdg_comm_style['target'] = 'plot_infomap.svg'
40 hdg_comm_style['palette'] = ig.PrecalculatedPalette(community_colors)
41 hdg_comm_style['edge_order_by'] = 'weight'
42 # hdg_comm_style['mark_groups'] = True # Uncomment if you want to delineate the groups
```

Listing 3.22: Styling using a style dict

[4]You can see how it looks on the pages 61-62

Plotting and reviewing the results

As you can see from the listing below, the plotting code didn't change from the last section, because all variable arguments have been placed in the style dictionary.

```
43 # Plotting
44 ig.plot(hdg_imap, **hdg_comm_style)
```

Listing 3.23: Saving the plot to a file

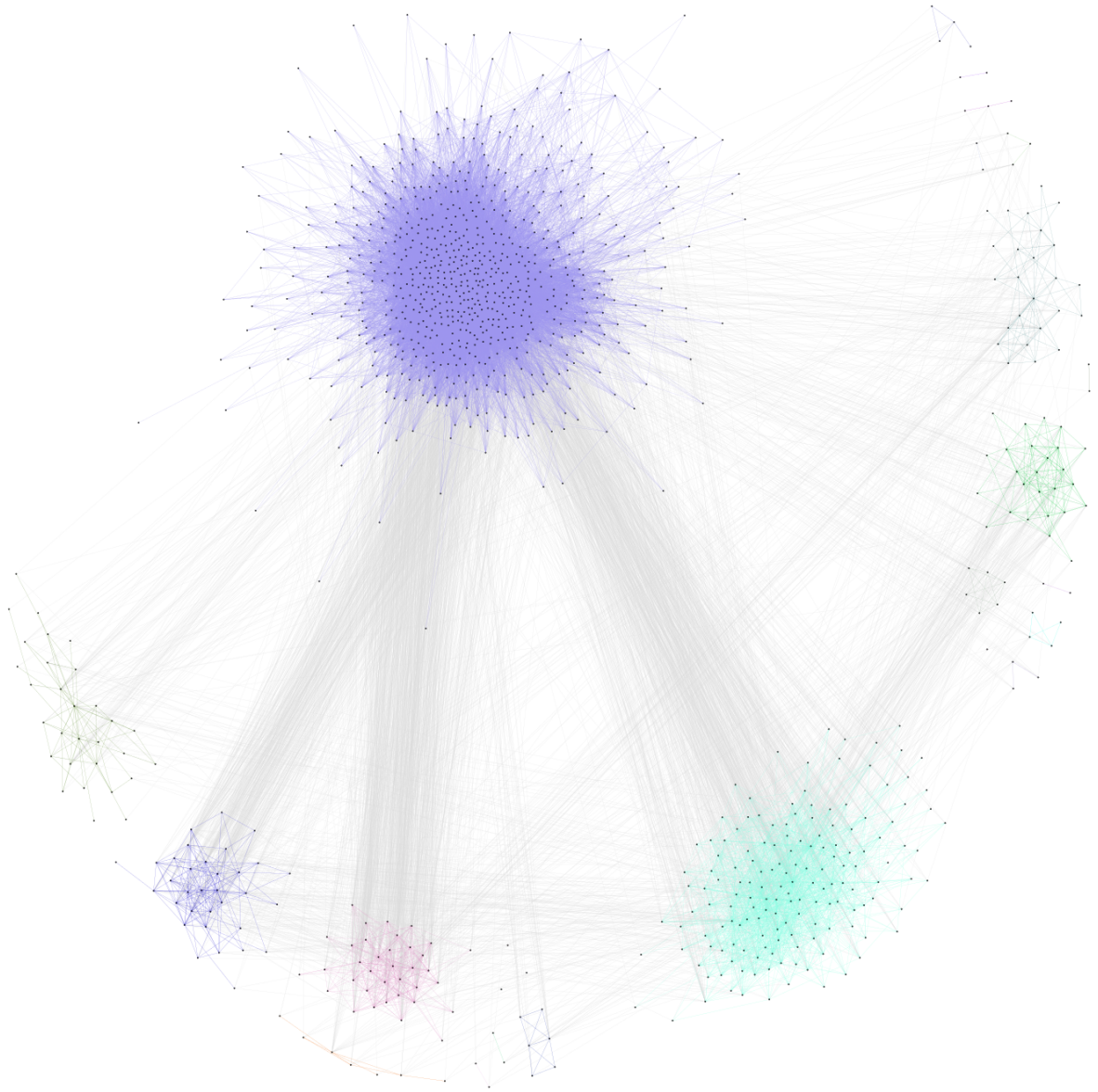

Figure 3.23: Community plot - Fruchterman Reingold layout

Looking at this plot, I find that the clear separation between groups and their coloring makes them easier to distinguish from each other, even if the groups could be made more prominent by changing their colors or increasing the edge thickness.

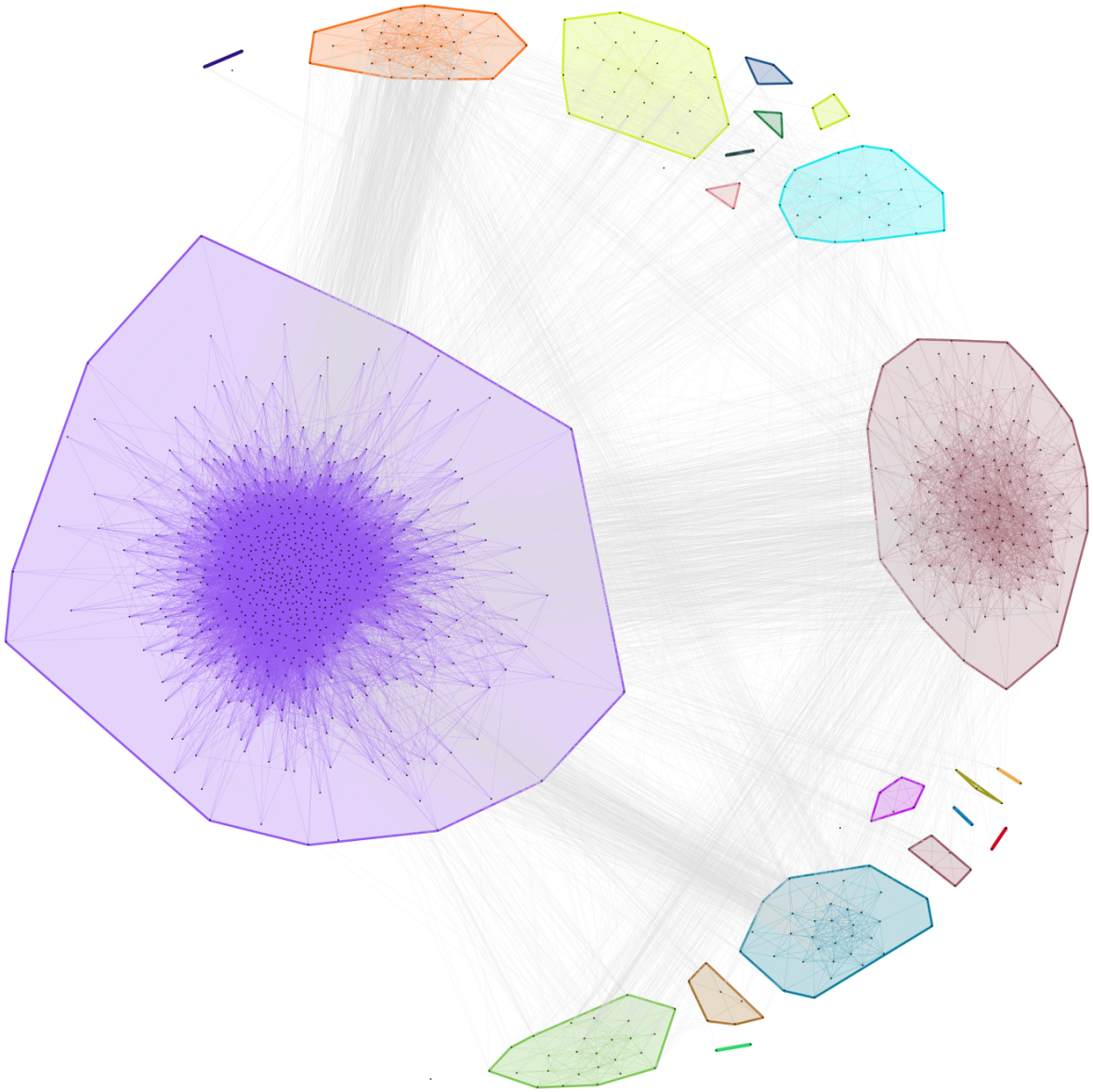

Figure 3.24: Community plot - delineated groups - Fruchterman Reingold layout

The delineated plot separates the communities even more than the regular coloring does, and adds boundaries that make even smallest groups of 2-3 vertices visible due to the bold lines connecting them. The problem with this plot is that the boundaries that separate the groups are drawn first, which makes them being crossed over by the edges that connect different communities. While that could be changed, the way of doing that would have to be very roundabout due to the delineation itself being just a true or false check in the plot function. I think that it may have been done intentially, just so you could actually see the edges properly.

To sum up, adding some kind of order with the groups and colors is, in my mind, a major improvement compared to the plots that were generated in the last section.

3.3.4 Weighted community plot

Pagerank application

Pagerank[24] is an algorithm developed by, among others, Larry Page and Sergey Brin that was used as a basis for the Google search engine. It uses a number of links to a specific vertex (originally, a webpage) to evaluate their importance. This metric is used as the weight of the vertices in this example. The pagerank implementation in the igraph package already auto assigns the weight, so you don't have to do that yourself.

```
1  hdg_pgrank = hdg_subgraph.pagerank()
```

Listing 3.24: Using pagerank to assign weights to vertices

Style dictionary

Style dictionary for the weighted graph is very similar to the one that was used to create the community plot, with an additions of weights as a keyword argument to pass to the fruchterman reingold layout and vertice size now depending on it's weight instead of being constant. As you can see from the code listing, the `vertex_size` requires either a constant number or a iterable of all vertices' weights.

```
2  hdg_comm_style = {}
3  hdg_comm_style['layout'] = \
4      hdg_subgraph.layout_fruchterman_reingold(maxiter=1000,
5                                               weights=hdg_subgraph.es['weight'],
6                                               area=hdg_vcount**3,
7                                               repulserad=hdg_vcount**3)
8  hdg_comm_style['bbox'] = (1024, 1024)
9  hdg_comm_style['vertex_size'] = hdg_pgrank_arr * 3000
10 hdg_comm_style['vertex_shape'] = 'circle'
11 hdg_comm_style['edge_width'] = 0.1
12 hdg_comm_style['target'] = 'plot_pagerank_fg.svg'
13 hdg_comm_style['edge_order_by'] = 'weight'
14 hdg_comm_style['palette'] = ig.PrecalculatedPalette(community_colors)
```

Listing 3.25: Styling using a style dict

Plotting and reviewing results

The plot function call in this case is the same as in two previous ones - `ig.plot(hdg_subgraph, **hdg_style)`.

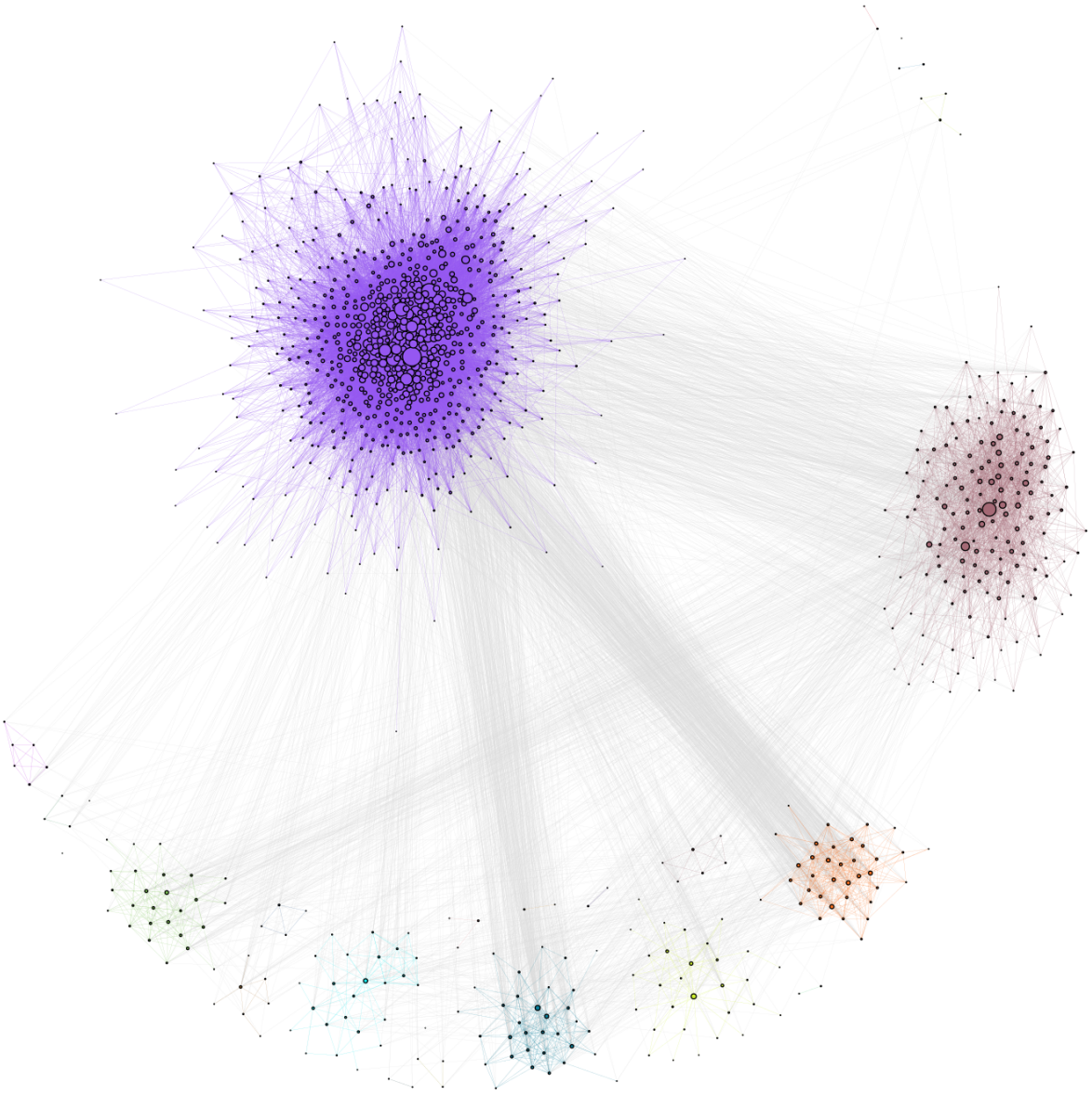

Figure 3.25: Weighted plot - Fruchterman Reingold layout

Comparing this plot to the previous two, this one clearly presents the most information out of the three. If in the first section we had to only deal with the vertices and edges themselves, now we not only have communities of vertices that should be grouped, we also have different sizes of vertices inside those communities. While pagerank is an algorithm that always values nodes with more connection higher, thus making the violet community the most prominent group, if this chart was made using a different data set it could reveal something like the least populated community being the most important or something among those lines.

3.4 Summary

This project's goals were focused on showing the reader how to use igraph and how to plot the graph data using datashader and igraph/cairo. It accomplishes these goals, and I hope that more people will start using igraph after reading this work. I would say that the plots creating with the use of igraph/cairo tools are overall more informative and the tools themselves have a larger amount of possible options than datashader, but it balances out by the fact that igraph/cairo can only process relatively smaller graphs (around a thousand of nodes), while datashader has processed the full graph which had more than 3 million nodes. I would also say that the igraph is rather easy to use compared to datashader. As for the community detection section, most community detection algorithms have their own packages (like louvain) or are built into a package, so that is a benefit of using a Python programming environment - it allows us to not reinvent the wheel and just use the already made algorithm.

4 GEOGRAPHICAL DATA ANALYSIS

4.1 Country level

4.1.1 Data loading

Packages used

The listing 4.1 includes all packages that you will find the packages used in the project.

```
# STL
import os
# Data manipulation
import sqlite3
import numpy as np
import pandas as pd
# Matplotlib
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
from matplotlib import afm, cbook, ft2font, rcParams, get_cachedir
# Map
from mpl_toolkits.basemap import Basemap
```

Listing 4.1: Imports

Using a Sqlite database

Sqlite [23] is a relational database management system, that, unlike other database systems, doesn't utilize a client-server model. A Sqlite database is usually stored in a single file and can be transmitted to other people to share data. It has a downside though - it can only support one active connection at a time, so while it is popular among developers, they are often forced to switch to a more "heavy" database when they publish their products. Thankfully, in this case it is going to be enough for this project, since it doesn't require multiple connections.

In this case, I've made a connection cursor using a `sqlite3.connect(file_path)` function, that allows me to execute SQL queries. But, instead of executing the query myself I had decided to use pandas in order to get a DataFrame as a result right away without having to convert it to one.

```
con = sqlite3.connect(os.path.join('wdi', 'database.sqlite'))
df = pd.read_sql_query(
    'SELECT * FROM Country c '
```

```
18      'INNER JOIN Indicators i ON c.CountryCode = i.CountryCode '
19      'WHERE i.IndicatorCode="IT.CEL.SETS.P2"', con)
```

Listing 4.2: Loading the data

```
20  print(df.columns)  # Has duplicate columns due to joining
21  df = df.T.drop_duplicates().T  # Removing duplicate columns
22  df = df.infer_objects()  # Properly parse object types
```

Listing 4.3: Removing duplicate columns

4.1.2 Country level visualization

This project is going to use the basemap package from matplotlib toolkits. To use it, you firstly have to create a Basemap object while specifying a projection that you are going to use as well as the resolution of your map (the c in the listing 4.4 stands for “crude", or the less detailed plot). Then you have to obtain a special shape file (you can download it from a website like Natural Earth Data [8]) that contains country borders and use the `readshapefile` method in order to process it. This example is focused on country level data, so we have to use the `drawcountries` method in order to plot the country borders as well as the `drawcoastlines` to show the continent borders.

Then I start a loop, in which I retrieve a value of the indicator for every country, and either color it with the corresponding shade of the color scheme or just leave it gray in case the data set didn’t include this country.

The legend has to be plotted on a separate axis in order to not overlap with the map, and the loop that you see after that is for showing more precise limits of the color shades. Please not the use of semi-private `_ticker()` function in order to obtain the coordinates of the tickers.

```
23  def draw_plot(df_plot, bins, bins_real, year, title, legend_title):
24      # Preparation
25      cm = plt.get_cmap('Greens')
26      scheme = [cm(i / len(bins)) for i in range(len(bins))]
27      cmap = matplotlib.colors.ListedColormap(scheme)
28      shapefile = 'shape/ne_10m_admin_0_countries'
29
30      # Creating figure and axis. We need 1 axis for the plot and 1 for legend
31      fig = plt.figure(figsize=(18, 18))
32      ax = plt.subplot2grid((2, 2), (0, 0), rowspan=2, colspan=2)
33
34      # Creating a map and reading it from a shape file
35      m = Basemap(lon_0=0, projection='robin', resolution='c')
```

```
    m.readshapefile(shapefile, 'units', color='#444444', linewidth=.2)

    # Drawing the country and continent borders
    m.drawcoastlines()
    m.drawcountries()

    # Iterating through the map units because it has the country info
    # df_plot contains kpis and we are using a country as a key
    for info, shape in zip(m.units_info, m.units):
        iso3 = info['ADM0_A3']
        if iso3 in df_plot.index:
            color = scheme[df_plot.loc[iso3]['bin'] - 1]
        else:
            color = '#dddddd'
        # Adding a polygon for each country
        ax.add_patch(Polygon(np.array(shape), facecolor=color))

    # Creating a legend
    ax_legend = fig.add_axes([0.35, 0.25, 0.3, 0.03], zorder=3)
    cb = matplotlib.colorbar.ColorbarBase(
        ax_legend, ticks=bins, boundaries=bins, cmap=cmap, orientation='horizontal')
    ax_legend.set_xticklabels([f'{i:.0%}' for i in bins])

    # Ticks inside legend's color boxes
    if show_values_inside:
        for i in range(len(bins_real) - 1):
            text = f'{bins_real[i]:.1f}-{bins_real[i+1]:.1f}'
            x_indent = cb._ticker()[0][i] + 0.09 - len(text) / 165
            ax_legend.annotate(text, xy=(x_indent, 0.4))

    # Title and legend title
    ax.set_title(title, fontsize=17)
    ax_legend.set_title(legend_title)

    return fig, ax
```

Listing 4.4: Drawing a map plot

The figure 4.5 shows the data preparation that executes before we call the aforementioned plot function.

```
def plot(year, title, legend_title, ind='IT.CEL.SETS.P2', show_values_inside=False):
    # Customizing the data connection to be able to retrieve different indicators
    con = sqlite3.connect(os.path.join(get_root(), 'wdi', 'database.sqlite'))

    df = pd.read_sql_query(
        'SELECT * FROM Country c '
        'INNER JOIN Indicators i ON c.CountryCode = i.CountryCode '
        f'WHERE i.IndicatorCode="{ind}"', con)
```

```
79
80      df = df.T.drop_duplicates().T  # Removing duplicate columns
81      df = df.infer_objects()  # Correct object types
82
83      df_plot = df[(df['Year'] == year) & (df['Region'] != '')]
84      df_plot = df_plot.iloc[:, [0, 1, -1]]
85
86      if (df_plot['Value'].max() <= 0):
87          raise Exception('No data to plot')
88
89      # Normalizing the data to show percentages
90      norm_min = df_plot['Value'].min()
91      norm_max = df_plot['Value'].max()
92      df_plot['Value'] = (df_plot['Value'] - norm_min) / (norm_max - norm_min)
93
94      # Using countrycode as an index to use it in draw_plot
95      df_plot.set_index('CountryCode', inplace=True)
96      df_plot.dropna(inplace=True)
97
98      # Calculating the bin distribution
99      _, bins = np.histogram(df_plot['Value'], bins=5)
100     df_plot['bin'] = np.digitize(df_plot['Value'], bins)
101     bins_real = norm_min + bins * (norm_max - norm_min)
102
103     return df, draw_plot(df_plot, bins, bins_real, year, title, legend_title, show_values_inside)
```

Listing 4.5: Preparing data and plotting

4.1.3 Results

The resulting plot (figure 4.1) looks quite informative and presents data well. I'm quite surprives that in some countries there are two mobile celluear subscriptions per person, but I guess that is because of people having to use one phone for work and the second one for communicating with close friends as family.

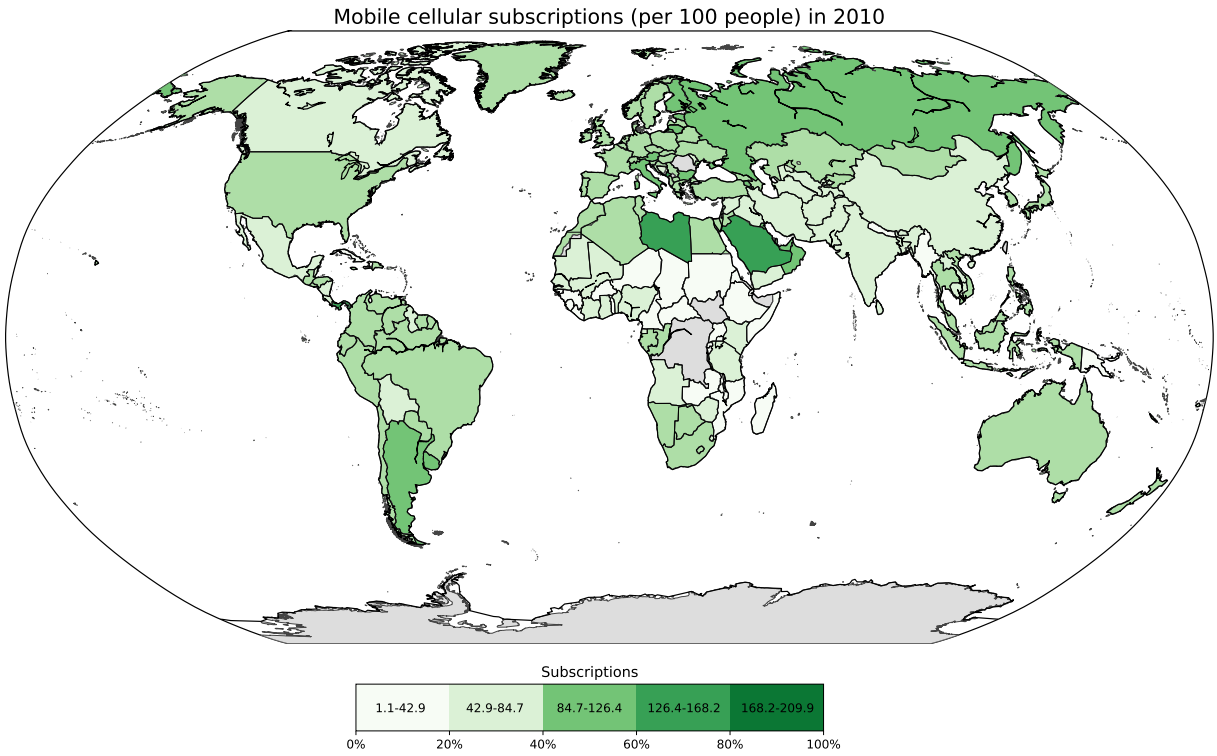

Figure 4.1: Country level vizualization - Cellular subscriptions

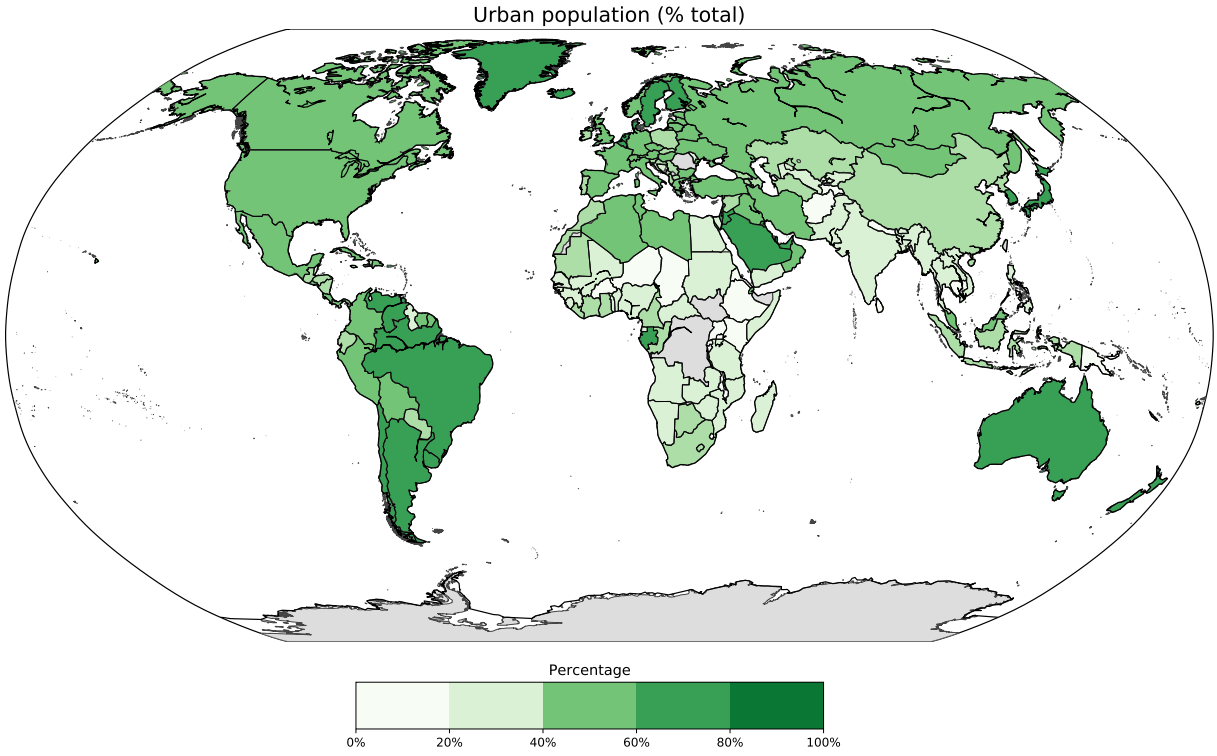

Figure 4.2: Country level vizualization - Urban population

4.2 City level visualization

4.2.1 Data loading

Packages used

The listing 4.6 includes all packages that you will find the packages used in the project.

```
# STL
from functools import partial
import os

# Data
import numpy as np
import pandas as pd
import dask.dataframe as dd
from datashader.utils import lnglat_to_meters

# Plotting
import datashader as ds
import datashader.transfer_functions as ds_tf
import bokeh.plotting as bp
from bokeh.models.tiles import WMTSTileSource
from datashader.bokeh_ext import InteractiveImage
```

Listing 4.6: Imports

Using Dask

Dask is a package that tries to replicate the behavior of pandas's DataFrame while parallelizing the actual computations. Since the data we are dealing with in this project contains more than 15 million rows after discarding the null rows, it should provide us with significant processing speed improvements. It uses a partition-based system that stores partitions of your main file and operates on them. The partitions can be computed and merged together by using the `df.compute()` method. As you can see from the listing 4.7, it replicates the read_csv function from pandas quite perfectly.

```
df = dd.read_csv(os.path.join(get_root(), 'taxi', 'Taxi_Trips.csv'), usecols=[
    'Trip Start Timestamp',
    'Trip End Timestamp',
    'Trip Seconds',
    'Trip Miles',
    'Trip Total',
    'Payment Type',
    'Pickup Centroid Location',
```

```
    'Dropoff Centroid  Location'
], assume_missing=True).dropna().reset_index(drop=True)
df = df.rename(columns={'Dropoff Centroid  Location': 'Dropoff',
                        'Pickup Centroid Location': 'Pickup'})
```

Listing 4.7: Loading the data using dask

Transformation is one point where dask diverges from the pandas package. Since the execution has to be parallelized, you have to create a special function that you will then pass in `df.map_partitions(func)`, which will schedule your function for execution in every partition of the dask dataframe. In this transformation, I have used simple string parsing functions like `split` to parse a column that originally had a form `POINT (-87.6262149064 41.8925077809)`. And since the plot that we will create later will use Marcator coordinate system, I have converted the parsed latitudes and longitudes to Mercator system as well.

```
def convert_points(arr):
    return_list = []
    for point in arr:
        try:
            # If we can't parse the float, the row is empty, fill it with 0s
            return_list.append([float(var) for var in point[7:-1].split(' ')])
        except:
            return_list.append([0.0, 0.0])
    return return_list


def transform(df):
    # Pickup
    # Converting the pickup column into numbers
    pickup_latlng = pd.DataFrame(convert_points(df['Pickup']),
                                 columns=['pickup_x', 'pickup_y'])
    # Converting lat/lng to Mercator coords
    df['pickup_x'], df['pickup_y'] = lnglat_to_meters(
        pickup_latlng['pickup_x'], pickup_latlng['pickup_y'])

    # Dropoff
    dropoff_latlng = pd.DataFrame(convert_points(df['Dropoff']),
                                  columns=['dropoff_x', 'dropoff_y'])
    df['dropoff_x'], df['dropoff_y'] = lnglat_to_meters(
        dropoff_latlng['dropoff_x'], dropoff_latlng['dropoff_y'])

    return df
```

Listing 4.8: Data transformation functions

As you can see, when you have finished your transformation function, actually executing the transformation is quite easy - you just have to map the function to the partitions and compute the result.

```
56 df = df.map_partitions(transform)
57 df_comp = df.compute()
58
59 print(len(df) == len(df_comp))  # Prints True
```

Listing 4.9: Executing the transformation

4.2.2 Plotting with bokeh and datashader

In order for our locations to be displayed on a satellite map, we have to use a special service that would provide us with those images. In the listing 4.10, I have used arcgisonline as this service. I also define the limits of our plot in there, which are approximate limits of Chicago.

```
60 geourl = 'https://server.arcgisonline.com/ArcGIS/rest/services/World_Imagery/MapServer/tile/{Z}/{Y}/{X}.jpg'
61 sw = lnglat_to_meters(-87.92, 41.67)  # lon = x, lat = y
62 ne = lnglat_to_meters(-87.53, 42.03)
63 x_range, y_range = zip(sw, ne)
```

Listing 4.10: Preparing to make the plots

After having defined the limits, we have to create the plots themselves. In this example, I will be using a bokeh plot to show the map base and a datashader plot to show the data points. While merging two plots from two different libraries is quite a challenging task, it is possible by using the `InteractiveImage` function from the datashader package, which was designed specifically to be compatible with bokeh and will provide us with an image that one can zoom in/out using a mouse, etc. In order to create this InteractiveImage, we need two components - a bokeh plot and a function that will return a datashader Figure when called. In the listing 4.11, I have defined the formulas that are used to create those two components, and in the listing 4.12 I have showed how to call them and create the plot. Please note the use of `partial` function from the `functools` package from the Python's standard library, it allows you to create a function with some of the arguments already set to some values. As you can see, in here it can be used in order to define whether we want to diplay taxi pickups or dropoffs, and possibly to change the color pallette of the plot.

```
64 def base_plot(width=1000, height=1000):
65     p = bp.figure(plot_width=width, plot_height=height,
```

```
                    x_range=x_range, y_range=y_range, outline_line_color=None,
                    min_border=0, min_border_left=0, min_border_right=0,
                    min_border_top=0, min_border_bottom=0)
    # Making everything invisible to just show the map
    p.axis.visible = False
    p.xgrid.grid_line_color = None
    p.ygrid.grid_line_color = None

    return p


def image_callback(x_range, y_range, width, height, cmap=ds.colors.Hot, which='dropoff'):
    # Create a canvas
    cvs = ds.Canvas(width, height, x_range, y_range)
    # Aggreagate points
    count = cvs.points(df_comp, f'{which}_x', f'{which}_y')
    # Draw points
    image = ds_tf.shade(count, cmap=cmap)
    # Dynspread dynamically adjusts the pixel size
    return ds_tf.dynspread(image, threshold=0.75, max_px=8)
```

Listing 4.11: Creating the plot functions

```
# Create a bokeh plot
p = base_plot()
# Use the map as the background
tile_renderer = p.add_tile(WMTSTileSource(url=geourl))
tile_renderer.alpha = 0.7
# Plot the datashader image on top of the bokeh one
InteractiveImage(p, image_callback)  # Dropoffs
InteractiveImage(p, partial(image_callback, which='pickup'))  # Pickups
```

Listing 4.12: Plotting the data

4.2.3 Results

You can see the results on the figures 4.3 and 4.4 below. The whiter a point is, the more pickups/dropoffs were made at that location. It is clearly visible, that the center of the city is the most active area in the taxi movement, which is to be expected.

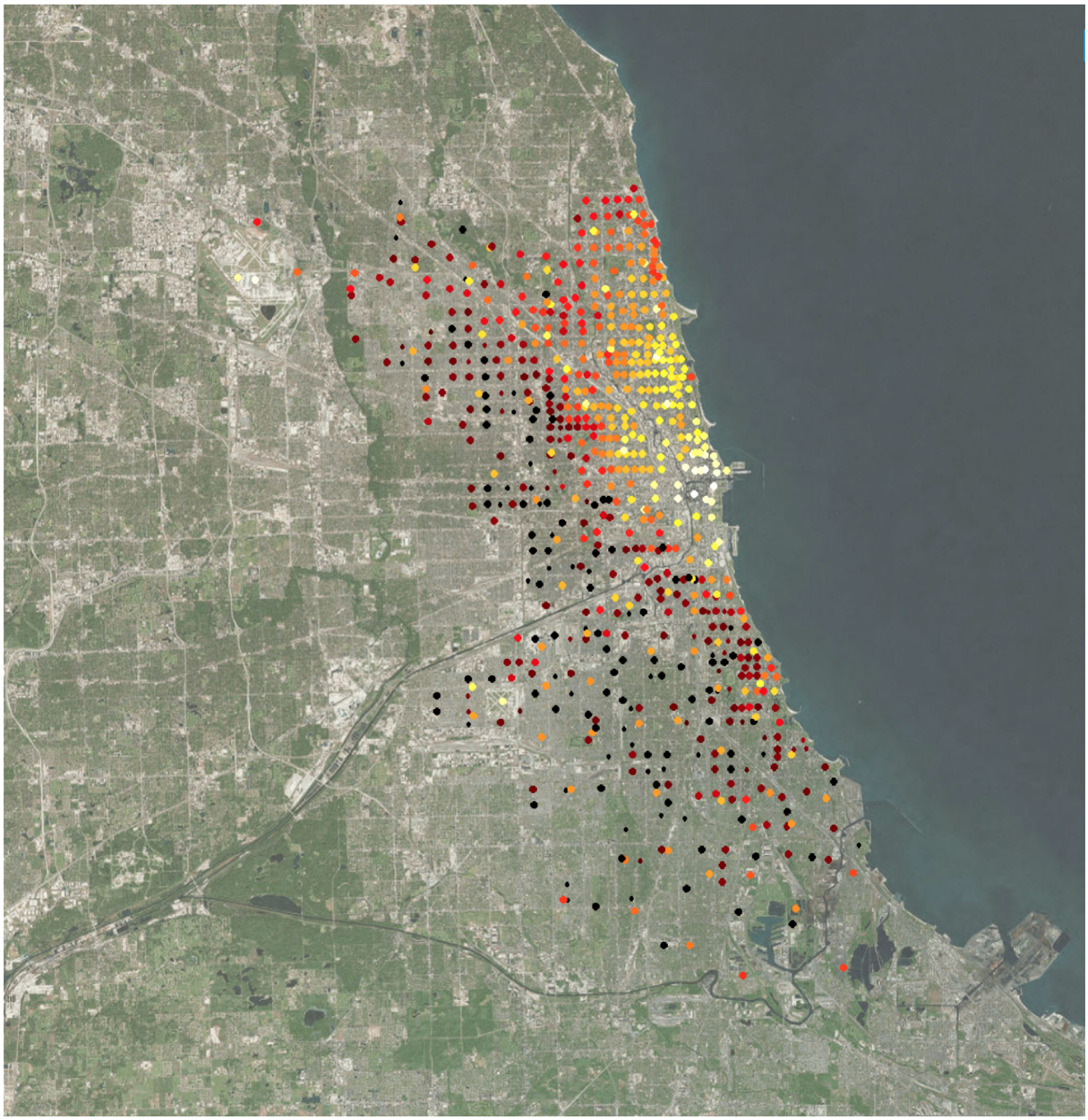

Figure 4.3: Pickup plot

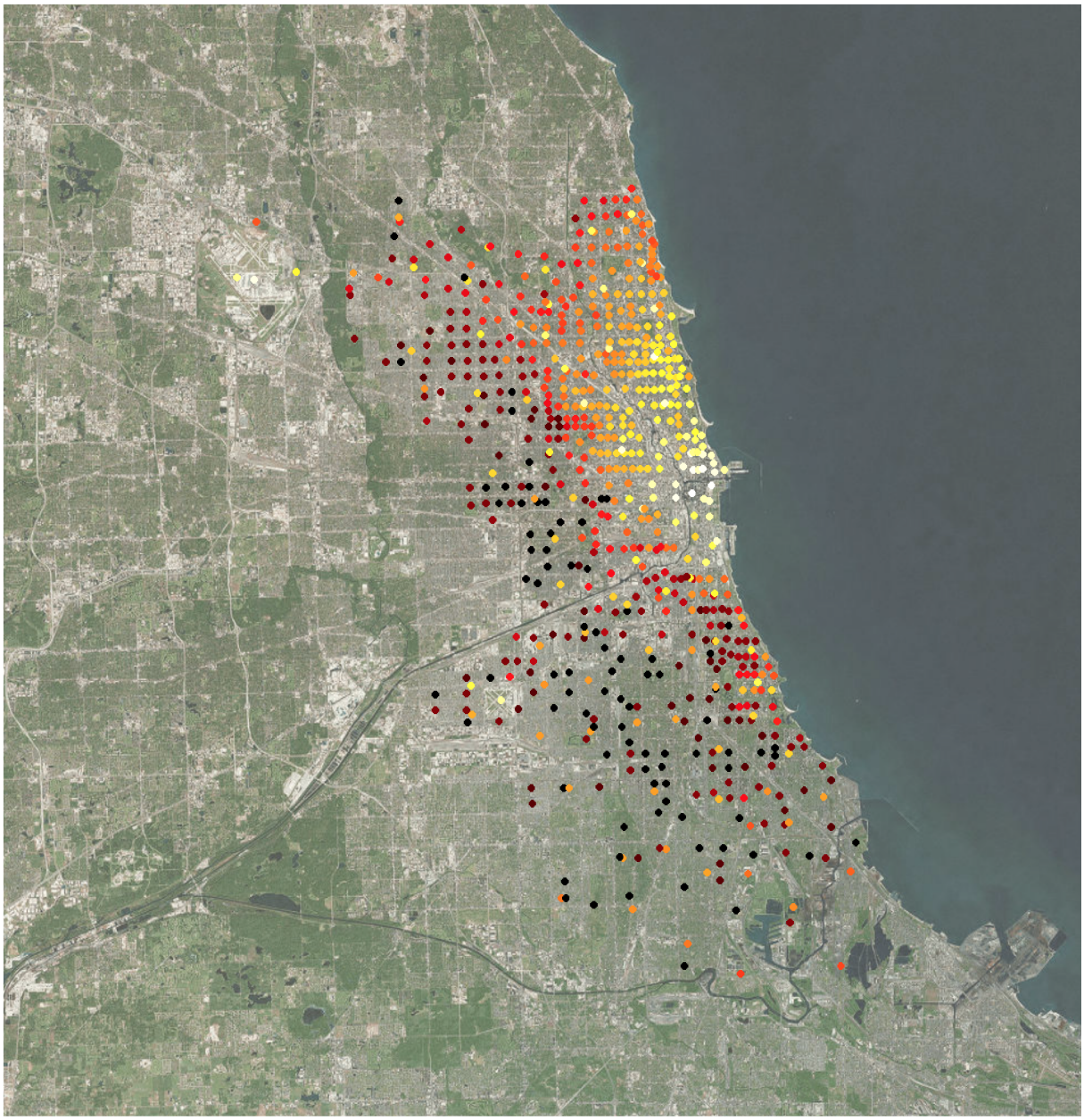

Figure 4.4: Dropoff plot

4.3 Summary

This project was focused on showing the reader how to process geographical data and how to plot it, and it accomplished those. I want to mention the matplotlib package in this section, since it has been used in two out of the three projects and has been very useful every time. I think that the country level visualization that was created using it is quite well made, and the relative ease of use of the package is apparent once you get through the initial exploration phase. I would also like to say that while the dask package is rather new, it has some promising possible usages, and I hope that it will be used and developed more. And while I have managed to plot the city-level

chart using bokeh and datashader, their integration was quite clunky and I would recommend trying to find a different technology stack to my readers.

5 CONCLUSION

While data analysis might seem like a scary and complex area of science, you can analize data quite easily by using easy to use tools created by other people. I hope that this paper will motivate other researchers to try out their strength in data science.

After finishing all of the projects, I would like to mention the packages that I found to be the most useful.

Firstly, numpy and pandas define the data analysis landscape in Python, and every data science project should consider using them if they are using Python as their language of choice.

Secondly, while matplotlib is usually regarded as a rather old-fashioned package due to the lack of the possibility to create interactive plots, it has a very vast array of possible usages, and provides a seemingly limitless number of tools that onle can use to create almost any kind of visualization, albeit with a bit of work that comes from learning how to properly utilize the package.

Thirdly, I can't stress how relatively easy it is to create neural networks with keras enough. It allows for a data scientist to focus on tuning all the possible parameters and inputs to the networks instead of worrying about something going wrong in its internals. It also utilizes tensorflow as a backend, which guarantees a certain speed of processing and a possibility to use the library in a CUDA GPU environment that speeds up the processing tremendously.

Fourtly, while the datashader package is very new, it can masterfully aggregate different types of data and plot it, even if the data is very big. I have used it in two use-cases here, one involving plotting of the graph and the second one involving plotting of the locational data, and in both cases it didn't require a lot of setup and aggregated the data rather well. You can see the results yourselves.

As for the projects, all of them have accomplished their respective goals, and were summarized in their respective sections, so while I will not go too much in detail about them here, I will shortly describe them.

The time series data analysis project showed the full stack of analyzing and predicting stocks, from downloading the data and individual stock visualizations to visualizations involving the entire data and predicting new data.

The graph data analysis project showed how to process a large graph in the igraph environment and had shown how to visualize the graph itself as well as different portions of the graph in

different ways, and compared those visualizations.

The geographical data analysis project had utilized two different datasets to show both the country-level data processing, where a location is represented by an iso country label and the city-level data processing, where a location is represented by a latitude and longitude. The section showed how to deal with the data in both cases and the project was able to generate plots based on it.

This work can be continued in many ways - this thesis omits how to deal with biological data, medical data, and motion tracking or augmented reality data. I would also like for someone to cover different packages that can be used to explore and analyze the same data types with different tools.

6 REFERENCES

[1] Abadi, M. et al. 'Tensorflow: Large-scale machine learning on heterogeneous distributed systems'. In: *arXiv preprint arXiv:1603.04467* (2016).

[2] Bank, W. *World Development Indicators*. 2017. URL: https://web.archive.org/web/20180122162054/https://www.kaggle.com/worldbank/world-development-indicators (visited on 22/01/2018).

[3] Bastian, M., Heymann, S., Jacomy, M. et al. 'Gephi: an open source software for exploring and manipulating networks.' In: *Icwsm* 8 (2009), pp. 361–362.

[4] Blondel, V. D. et al. 'The Louvain method for community detection in large networks'. In: *J of Statistical Mechanics: Theory and Experiment* 10 (2011), P10008.

[5] Chicago. *Chicago data portal - Taxi Trips*. 2017. URL: https://web.archive.org/web/20180124053222/https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew (visited on 24/01/2018).

[6] Chollet, F. et al. *Keras*. https://github.com/keras-team/keras. 2015.

[7] Csardi, G. and Nepusz, T. 'The igraph software package for complex network research'. In: *InterJournal, Complex Systems* 1695.5 (2006), pp. 1–9.

[8] Earth, N. *Natural earth data*. 2018. URL: https://web.archive.org/web/20180122163900/http://www.naturalearthdata.com/ (visited on 22/01/2018).

[9] Fruchterman, T. M. and Reingold, E. M. 'Graph drawing by force-directed placement'. In: *Software: Practice and experience* 21.11 (1991), pp. 1129–1164.

[10] Grus, J. *Data science from scratch: First principles with Python*. " O'Reilly Media, Inc.", 2015.

[11] Hochreiter, S. and Schmidhuber, J. 'Long short-term memory'. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[12] Hunter, J. D. 'Matplotlib: A 2D graphics environment'. In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95.

[13] James A. Bednar, J. C. et al. *Datashader: Revealing the Structure of Genuinely Big Data*. 15th Python in Science Conference (SciPy 2016). 2016.

[14] Kamada, T. and Kawai, S. ‘An algorithm for drawing general undirected graphs’. In: *Information processing letters* 31.1 (1989), pp. 7–15.

[15] Kingma, D. and Ba, J. ‘Adam: A method for stochastic optimization’. In: *arXiv preprint arXiv:1412.6980* (2014).

[16] Kwapień, J., Drożdż, S., Oświe, P. et al. ‘The bulk of the stock market correlation matrix is not pure noise’. In: *Physica A: Statistical Mechanics and its applications* 359 (2006), pp. 589–606.

[17] Martin, S., Brown, W. M. and Wylie, B. N. *Dr. L: Distributed Recursive (Graph) Layout*. Tech. rep. Sandia National Laboratories, 2007.

[18] McKinney, W. et al. ‘Data structures for statistical computing in python’. In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. SciPy Austin, TX. 2010, pp. 51–56.

[19] Mislove, A. ‘Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems’. PhD thesis. Rice University, Department of Computer Science, May 2009.

[20] Müller, A. C. and Guido, S. *Introduction to machine learning with Python: a guide for data scientists*. 2016.

[21] NASDAQ. *Company List (NASDAQ, NYSE, & AMEX)*. URL: `https://web.archive.org/web/20180121224545/http://www.nasdaq.com/screening/company-list.aspx` (visited on 21/01/2017).

[22] Nison, S. *Japanese candlestick charting techniques: a contemporary guide to the ancient investment techniques of the Far East*. Penguin, 2001.

[23] Owens, M. and Allen, G. *SQLite*. Springer, 2010.

[24] Page, L. et al. *The PageRank citation ranking: Bringing order to the web.* Tech. rep. Stanford InfoLab, 1999.

[25] Pedregosa, F. et al. ‘Scikit-learn: Machine learning in Python’. In: *Journal of Machine Learning Research* 12.Oct (2011), pp. 2825–2830.

[26] Puget, J. F. *The Most Popular Language For Machine Learning and Data Science Is . . .* URL: `https://www.ibm.com/developerworks/community/blogs/jfp/entry/What_Language_Is_Best_For_Machine_Learning_And_Data_Science?lang=en` (visited on 01/11/2017).

[27] Reingold, E. M. and Tilford, J. S. ‘Tidier drawings of trees’. In: *IEEE Transactions on Software Engineering* 2 (1981), pp. 223–228.

[28] Rosvall, M. and Bergstrom, C. T. ‘Maps of random walks on complex networks reveal community structure’. In: *Proceedings of the National Academy of Sciences* 105.4 (2008), pp. 1118–1123.

[29] Schmuhl, M. *graphopt*. 2003. URL: `https://web.archive.org/web/20180116173939/http://www.schmuhl.org/graphopt/` (visited on 16/01/2018).

[30] Sugiyama, K., Tagawa, S. and Toda, M. ‘Methods for visual understanding of hierarchical system structures’. In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.2 (1981), pp. 109–125.

[31] Walt, S. v. d., Colbert, S. C. and Varoquaux, G. ‘The NumPy array: a structure for efficient numerical computation’. In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30.

[32] Wold, S., Esbensen, K. and Geladi, P. ‘Principal component analysis’. In: *Chemometrics and intelligent laboratory systems* 2.1-3 (1987), pp. 37–52.