Contents

Temporary Notes

Abstract

1 INTRODUCTION

1.1 State of the Art in Data Science

Use stuff from the presentation

1. Why is data analysis useful?
    1.1. Modern amounts of data explanation
    1.2. Data analysis explanation
2. What tools are used to deal with large amounts of data?
    2.1. Traditional - ETL and DW (microsoft, qlikview, ...)
    2.2. Explorational - spark, R, python, matlab
3. Why choose python?
    3.1. Advantages and disadvantages of using python
    3.2. Overview of chosen packages

Rewrite everything

1.2 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` and `R`, while `Matlab` is also quite popular despite only being used in the academic environment.

This work's objective is to show how to use the Python 3 programming language in dealing with different kinds of data, and to help clarify any problems that might come up. 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.

1.3 Thesis Overview

This work will be split into three parts, each working with a different dataset.

In the first part, I'll show you how to obtain, plot and predict stocks based on the last 17 years' worth of stock data from NYSE[1]. I'll also cover some common problems that might occur

when one is trying to deal with such amount of data.

In the second part, I'll cover scraping facebook's API, and plotting geolocation data of their events. I'll also discuss some problems that might occur while trying to download data, as well as how to use latest tools from Python (like the asyncio library) to speed up the data gathering part greatly. I'll also give you a brief overview of the current data visualization landscape, and show you which plotting packages are the best to use when dealing with geolocation data.

In the third part, I'll delve into the YouTube system, and will try to download and analyze their videos.

Lastly, the fourth part will contain conclusions.

[1]New York Stock Exchange

2 TIME SERIES DATA ANALYSIS

2.1 Project Overview

2.1.1 Goals

- To show how to use python to obtain stock price data
- To show how to visualize the aforementioned data in different ways
- To show how to predict future prices by using a LSTM neural network

2.1.2 Description

This project will focus on dealing with time series data, which will be represented by the stock price data. It will contain four sections.

In the first one, I will show how to use the pandas-datareader package to download stocks traded on the New York Stock Exchange (NYSE) from the yahoo! finance website and merge them into one comma separated file.

In the second section, I will cover simple visualizations of singular stocks using the matplotlib package, which will include moving average and candlestick plots.

In the third section, I will cover visualization of a stock correlation matrix, which will, as opposed to the simple plots, include all of the stocks that were downloaded and not only one of them.

In the fourth and final section, I will demonstrate how to make a LSTM neural network for predicting the stock prices using the keras package and how to plot the predictions it generates, as well as cover how to use the pickle package to serialize python objects.

2.2 Data gathering (pandas-datareader)

Goal

The goal of this section is to show how to use python to obtain stock price data, and transform it for further use.

Process description

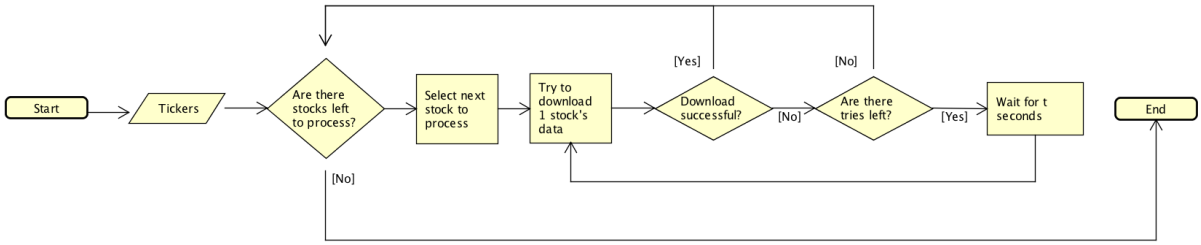


Figure 2.1: Stocks download flowchart

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 [7] 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, reload_tickers=False,
                   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...')
    if reload_tickers:
        tickers = save_tickers()
    else:
```

```
        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')
        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:
```

```
64              to_down_cnt -= 1
65              logging.debug(f'Not downloading data for {ticker}, '
66                            'since we already have it')
67      logging.info('Finished processing all tickers!')
68      logging.info(f'Downloaded: {down_cnt}/{to_down_cnt} items')
69      logging.info(f'You can find the results in the folder {dest_path}')
```

Listing 2.2: Downloading the stocks data

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.

```
70  def get_timearr(stock_dir, example_timefile='A.csv'):
71      time_df = pd.read_csv(os.path.join(
72          stock_dir, example_timefile), index_col=0)
73      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

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

you have a function that generates large amounts of data, your program can benefit drastically from using generators, since everything is generated on the fly and you don't have to store the whole resulting list in memory.

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

Listing 2.4: Listing all csv files in a directory

```
78 def reindex_csv(stock_dir, save_dir, time_arr, reload_data=False):
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         if not reload_data and os.path.isfile(os.path.join(save_dir,
84                                                            stock_fname)):
85             continue
86         stock = pd.read_csv(os.path.join(stock_dir, stock_fname), index_col=0)
87         stock = stock.reindex(time_arr, fill_value=np.nan)
88         stock.to_csv(os.path.join(save_dir, stock_fname))
89     gc.collect()
```

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.

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

    merged_df.to_csv(save_path)
```

```
156     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 2.4.

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

Listing 2.7: Creating a correlation matrix

Summary

-execute the functions-

```
173 # We want to get the data from 2000 till 2017
174 START_DT = dt.datetime(2000, 1, 1)
175 END_DT = dt.datetime(2017, 1, 1)
176 # We want to place the merged file in data/merged
177 STOCK_FOLDER = os.path.join('data', 'stocks')
178 MERGED_FOLDER = os.path.join('data', 'merged')
179
180 # Downloading the data
181 get_stock_data(START_DT, END_DT, max_tries=5, ticker_fname='NYSE.csv',
182                dest_folder=STOCK_FOLDER, timeout=0.1, provider='yahoo')
183
184 # Reindexing the csvs so we can np.concatenate them later
185 reindex_csv(STOCK_FOLDER, STOCK_FOLDER, get_timearr(STOCK_FOLDER, 'A.csv'),
186             reload_data=True)
187
188 # Merging the dataframes
189 merge_dfs(STOCK_FOLDER, MERGED_FOLDER, reload_data=True)
190
```

```
191 merge_dfs(STOCK_FOLDER, MERGED_FOLDER, save_fname='stocks_close_merged.csv',
192           reload_data=True, add_per_oc=False, add_per_lohi=False, add_volume=False)
193
194 # Making the correlation matrix
195 make_corr_matrix(MERGED_FOLDER, MERGED_FOLDER, reload_data=True)
```

Listing 2.8: Executing the functions

2.3 Traditional stock visualizations (matplotlib)

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

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

print('Getting data...')

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

print('Merged stocks...')
df_merged = pd.read_csv(
    os.path.join(MERGED_FOLDER, 'stocks_close_merged.csv'),
    parse_dates=True, index_col=0)

print('Finished getting data')
```

Listing 2.9: Reading the data into the memory

2.3.1 Basic plot

A basic plot of one stock can be plotted without having to resort to creating your own figures in matplotlib, since a pandas DataFrame has a built-in plot function.

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

Listing 2.10: Very basic plot of a stock

Figure 2.2: Basic plot

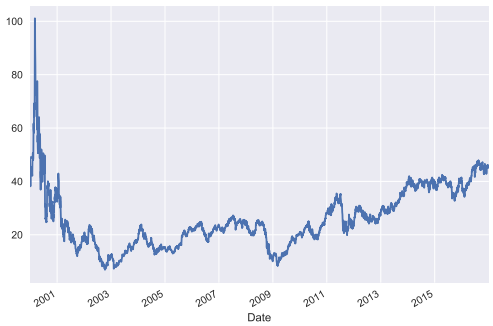
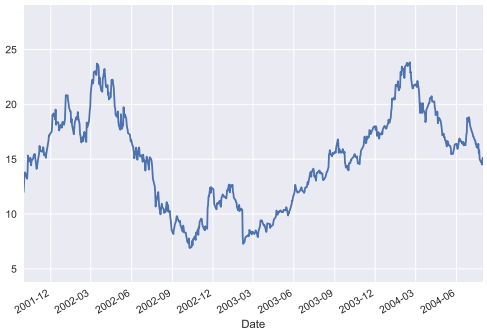


Figure 2.3: Basic plot - zoomed



2.3.2 Moving average plot

```
23 # Moving average
24 # Preparing the data
25 df_ma = df.copy()
26 # Creating the rolling avg column
27 df_ma['50MA'] = df_ma['Adj Close'].rolling(window=50, min_periods=0).mean()
28
29 # Plotting
30 plt.figure()
31
32 # Axis 1
33 ax1 = plt.subplot2grid((12, 1), (0, 0), rowspan=11, colspan=1)
34 ax1.xaxis_date()
35 ax1.plot(df_ma.index, df_ma['Adj Close'], color='#56648C')
36 ax1.plot(df_ma.index, df_ma['50MA'], color='#FF5320', linewidth=1)
37 plt.title('50 days moving average', fontsize=10, color='k')
38
39 # Axis 2
40 ax2 = plt.subplot2grid((12, 1), (11, 0), rowspan=1, colspan=1, sharex=ax1)
41 ax2.fill_between(df_ma.index, df_ma['Volume'],
42                  0, color='#56648C', label='Volume')
43
44 # Plot general
45 plt.setp(ax2.xaxis.get_majorticklabels(), rotation=45)
46 plt.suptitle('Agilent Technologies, Inc.', fontsize=12, color='k')
47 plt.show()
```

Listing 2.11: Moving average plot

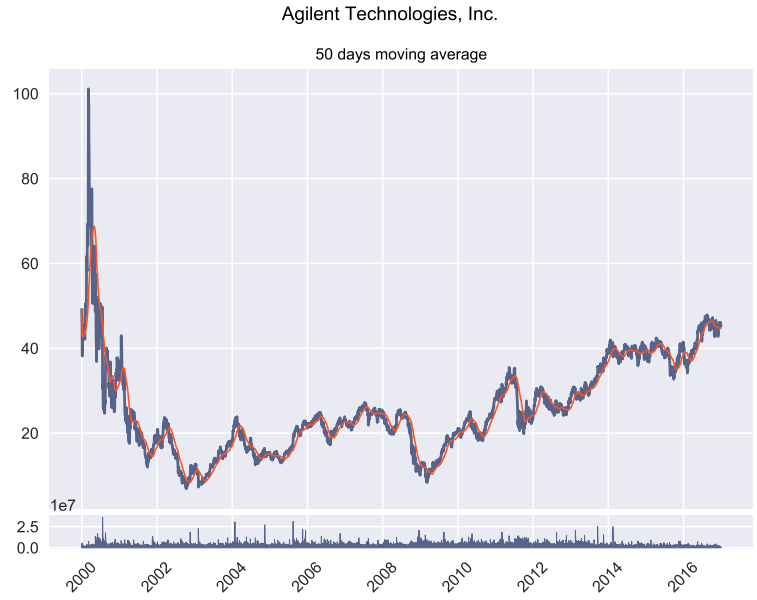


Figure 2.4: Moving average plot

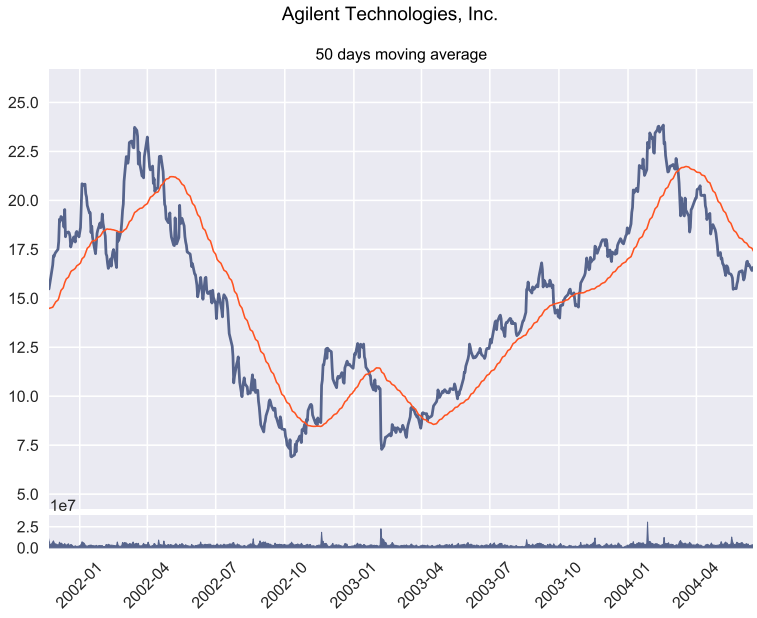


Figure 2.5: Moving average plot - zoomed

2.3.3 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 [8]. 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.

Figure 2.6: Candlestick plot - zoomed

The pandas package has some support for the candlestick plots, which, as you can see in the listing 2.12, 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.

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

```
48 # Candlestick
49 plt.figure(figsize=(8, 6))
```

```
50  date2num = matplotlib.dates.date2num
51
52  df_ohlc = df['Adj Close'].resample('10D').ohlc()
53  df_vol = df['Volume'].resample('10D').sum().to_frame()
54
55  df_ohlc.reset_index(inplace=True)
56  df_vol.reset_index(inplace=True)
57
58  # Converting the dates to the matplotlib's date format
59  df_ohlc['Date'] = df_ohlc['Date'].map(date2num)
60  df_vol['Date'] = df_vol['Date'].map(date2num)
61
62  # Axis 1
63  ax1 = plt.subplot2grid((12, 1), (0, 0), rowspan=11, colspan=1)
64  ax1.xaxis_date()
65  # Plotting the candlestick
66  matplotlib.finance.candlestick_ohlc(ax1, df_ohlc.values, colorup='g', width=2)
67  ax1.set_title('Candlestick plot', fontsize=10, color='k')
68  ax1.yaxis.set_label_text('Stock price ($)')
69
70  # Axis 2
71  ax2 = plt.subplot2grid((12, 1), (11, 0), rowspan=1, colspan=1, sharex=ax1)
72
73  # Fill between two curves
74  # (x, y_high, y_low)
75  ax2.fill_between(df_vol['Date'], df_vol['Volume'], 0, label='Volume')
76  ax2.xaxis.set_label_text('Date')
77
78  # General
79  plt.setp(ax2.xaxis.get_majorticklabels(), rotation=45)
80  plt.suptitle('Agilent Technologies, Inc.', fontsize=12, color='k')
81  plt.show()
```

Listing 2.12: Candlestick chart

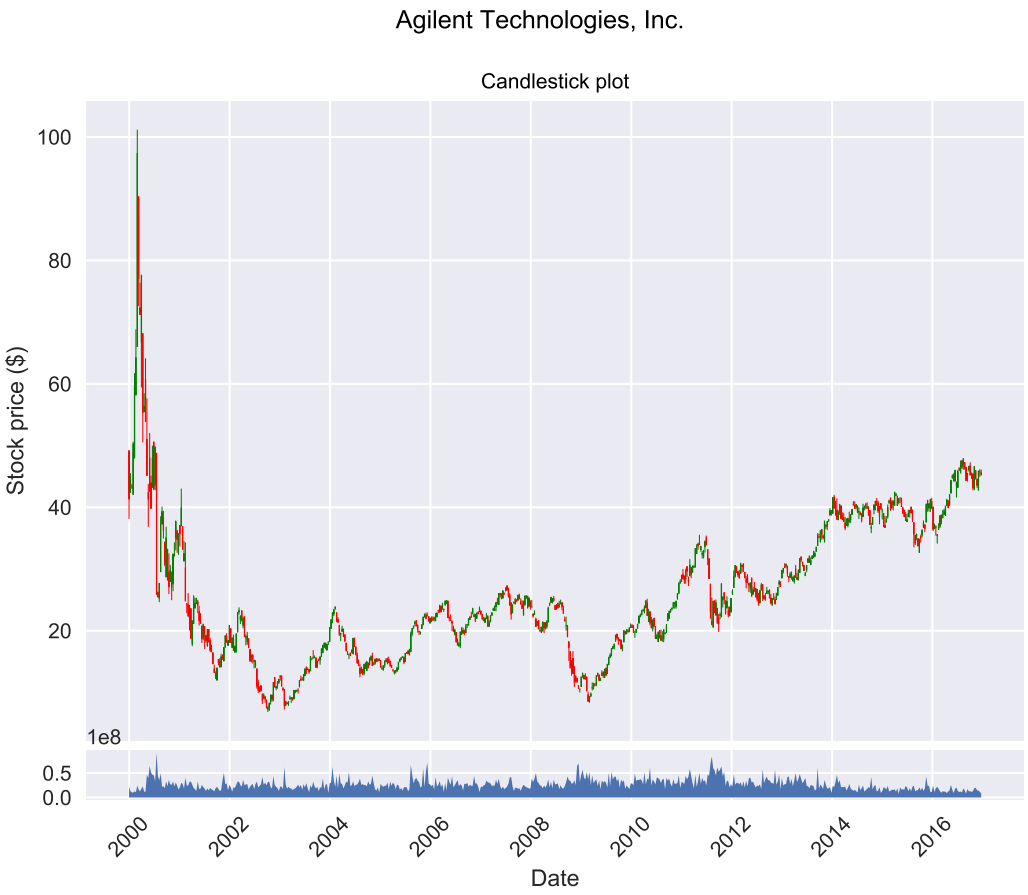


Figure 2.7: Candlestick plot

2.4 Stock correlation matrix visualization (matplotlib)

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.

Research like [4] 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.

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

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.

```
1  def get_root():
2      return os.path.abspath(os.sep)
3
4
5  STOCKS_FOLDER = os.path.join(get_root(), 'stocks')
6  MERGED_FOLDER = os.path.join('data', 'merged')
7
8  print('Getting data - merged correlation matrix...')
9
10 df_corr = pd.read_csv(
11     os.path.join(MERGED_FOLDER, 'corr_matrix.csv'), index_col=0)
12 print('Finished getting data')
```

Listing 2.13: Reading the data into the memory

```
13 # PROBLEM - LABELS ARE WRONG
14 # Making a correlation matrix
15 fig = plt.figure()
16
17 # Setting up the axis
18 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)

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

Listing 2.14: First attempt at making a correlation matrix

```
class StockFormatter(matplotlib.ticker.Formatter):
    def __init__(self, cols):
        self.cols = np.array(cols)

    def __call__(self, x, pos=None):
        return self.cols[np.clip(x, 0, len(self.cols) - 1).astype('int')]


# Making a correlation matrix
fig = plt.figure()

# Preparing the axis formatters
cols = list(df_corr.columns)
formatter_x = StockFormatter(cols)
formatter_y = StockFormatter(cols)

locator_x = matplotlib.ticker.MaxNLocator(10)
locator_y = matplotlib.ticker.MaxNLocator(10)

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

# Plotting the table itself
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)

```

```
65  plt.xticks(rotation=45)
66
67  # Setting tickers for Y axis
68  y.set_major_formatter(formatter_y)
69  y.set_major_locator(locator_y)
70
71  # Plotting the heatmap
72  plt.title('Stock correlation heatmap - fixed', color='k')
73  plt.show()
```

Listing 2.15: Fixing the labelling

2.5 Predictive analysis with a LSTM neural network (keras)

3 GRAPH DATA ANALYSIS

3.1 Project Overview

1. Goals

    1.1. to show how to run community detection algorithms in igraph

    1.2. to show how to plot the communities using two different methods - datashader (larger data) and cairo (smaller data)

    1.3. to show how to make the communities visually separable and how to incorporate node weights in the plot

2. Tools(Libraries) used

    2.1. Why I chose igraph

Packages needed

1. igraph
2. cairocffi

[1]

3.2 Preprocessing and igraph creation

1. Importing the data form konekt
2. Optimizing edges renaming with numpy vectorize/jit

   Data source [6]

3.3 Community detection (igraph)

3.4 Plotting large graphs (datashader)

3.4.1 Simple plot

Add a note about how datashader failed to plot the entire graph and why it's not a good idea in the first place (hard to see the points)

Goals

1. To show how you can plot regular large graphs with datashader
2. To show how you can plot large graphs while distinguishing communities in them

Description

3.4.2 Regular plot

Datashader introduction

Results

Results

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.

3.4.3 Plotting communities

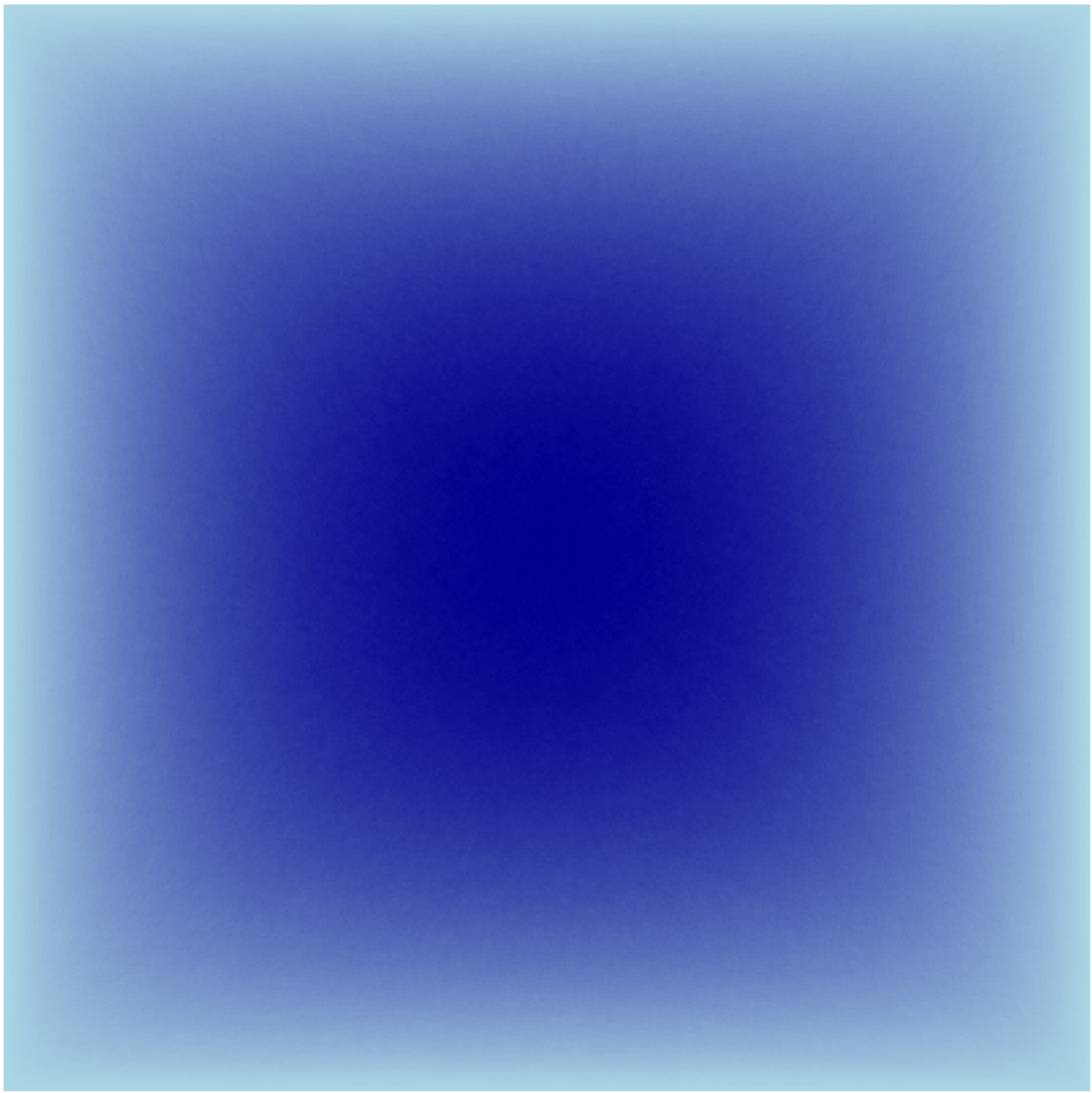


Figure 3.1: Directly connected layout - Plot of the full graph

3.5 Plotting small graphs (igraph)

3.5.1 Goals and section overview

Goals

1. To show how to create a regular graph's plot, and consider the idea that its simplicity may make it harder to read
2. To show how to create a graph plot with marked communities on it, and compare it to the regular plot in terms of ease of reading
3. 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

Description

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.

3.5.2 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.

```
# 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 3.1: 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 [2]
- Fruchterman Reingold grid layout
- DrL layout [5]
- Graphopt layout [11]
- Kamada Kawai layout [3]
- Sugiyama layout [12]
- Random layout
- Large Graph layout
- Reingold Tilford layout (for trees) [10]
- 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.

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.

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

[3]Visual comparison is available on the pages 33-35

Listing 3.2: Creating a style dictionary

Plotting and saving the resulting plot to a file

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

Listing 3.3: 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.

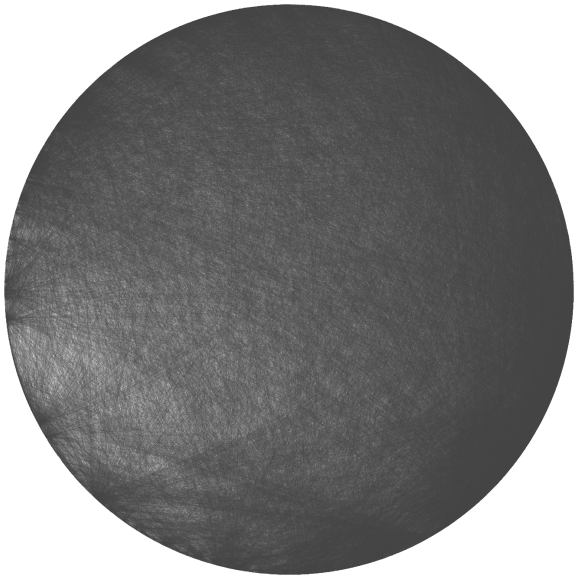


Figure 3.2: Circular layout

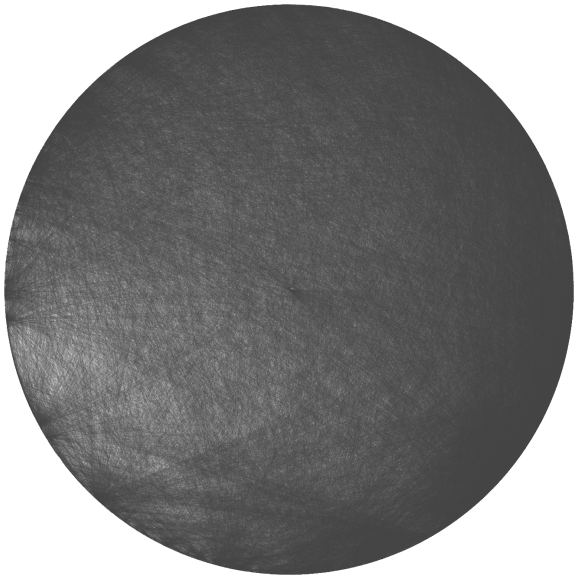


Figure 3.3: Star layout

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.

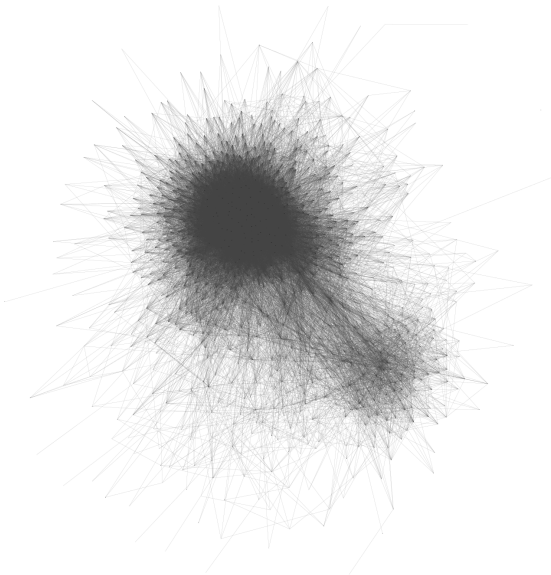


Figure 3.4: Fruchterman Reingold layout

Figure 3.6: DrL layout

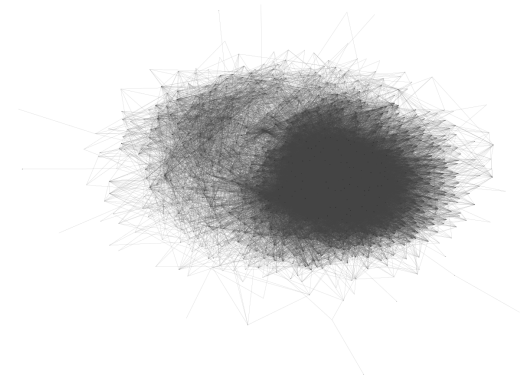


Figure 3.5: Kamada Kawai layout

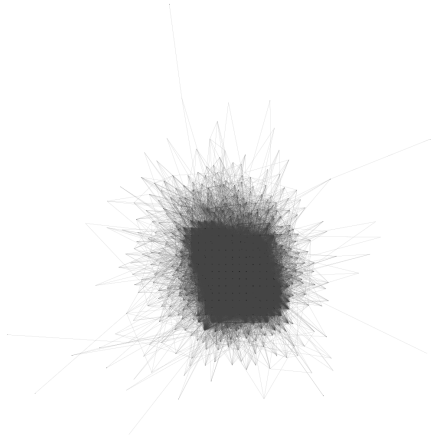


Figure 3.7: Graphopt layout

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.

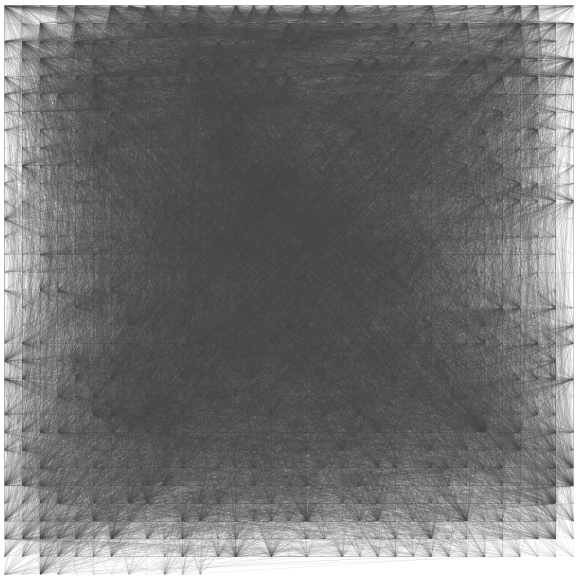


Figure 3.8: Grid layout

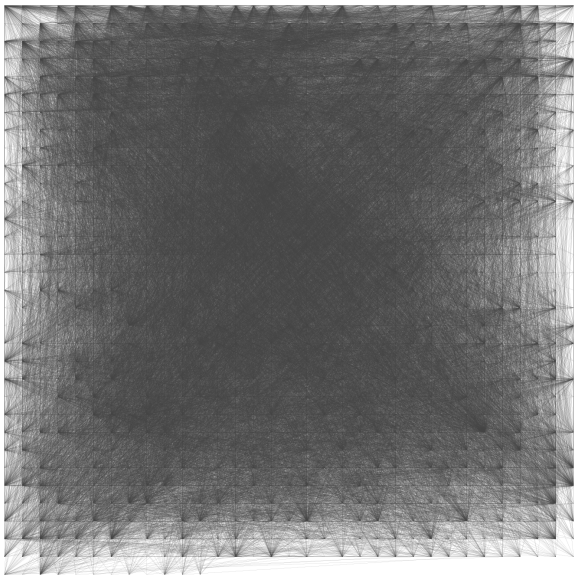


Figure 3.9: Fruchterman Reingold grid layout

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.

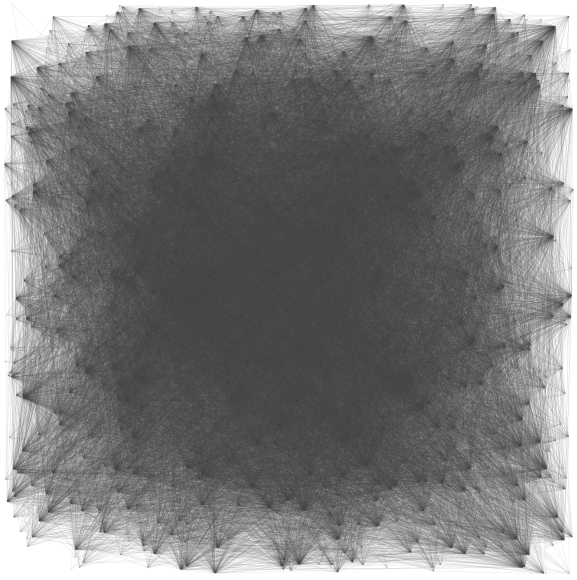


Figure 3.10: Random layout

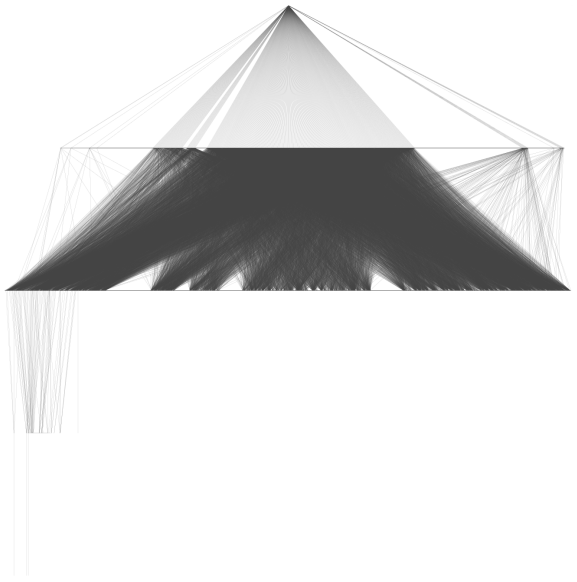


Figure 3.11: Sugiyama layout

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.

3.5.3 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.

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

Listing 3.4: 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 3.5: 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 3.6: 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 3.7: 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 3.8: 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 3.9: Styling using a style dict

[4]You can see how it looks on the pages 39-40

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 3.10: Saving the plot to a file

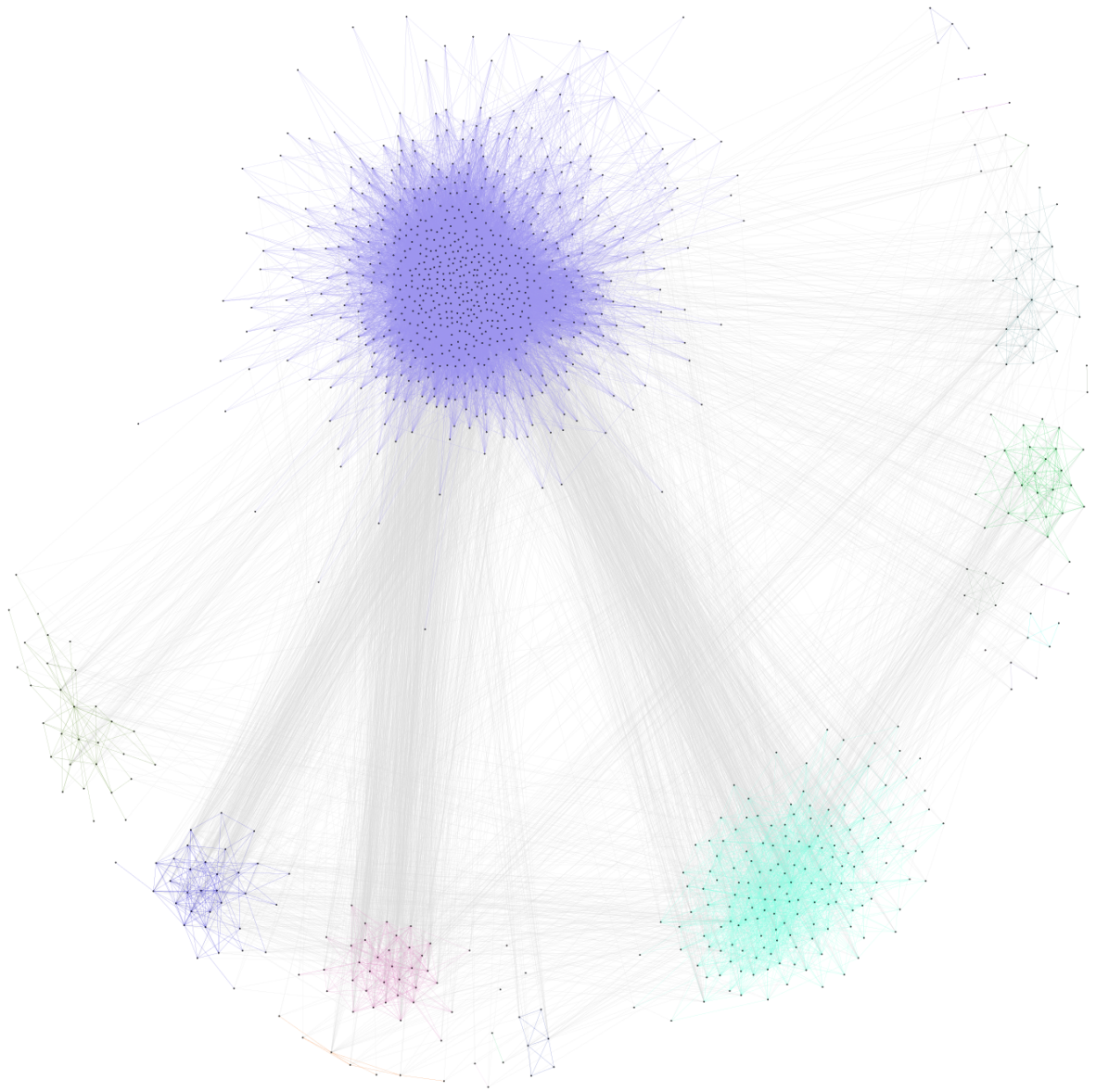


Figure 3.12: Community plot - Fruchterman Reingold layout

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.

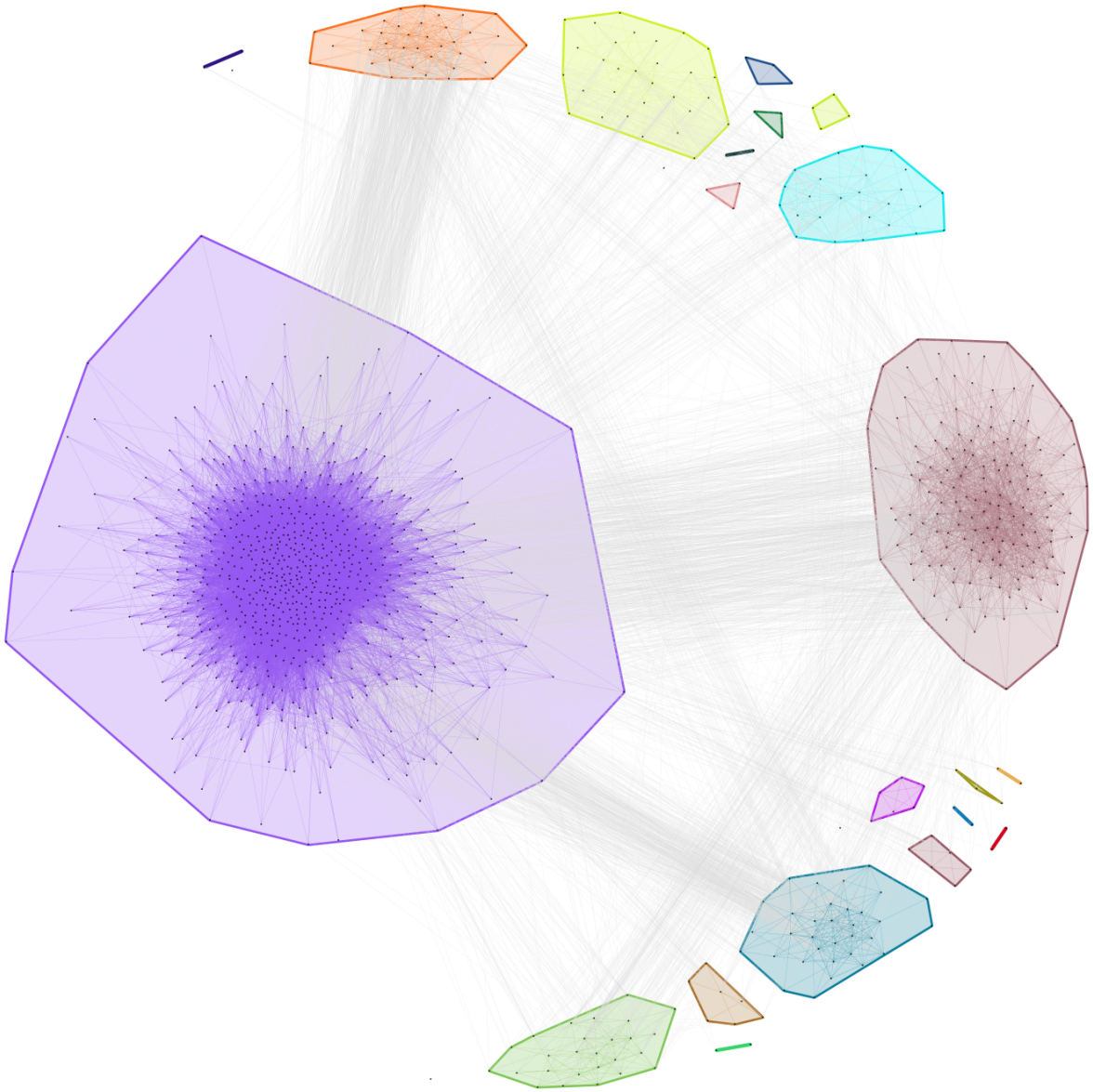


Figure 3.13: Community plot - delineated groups - Fruchterman Reingold layout

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.

3.5.4 Weighted community plot

Pagerank application

Pagerank[9] 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 3.11: 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 3.12: 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)`.

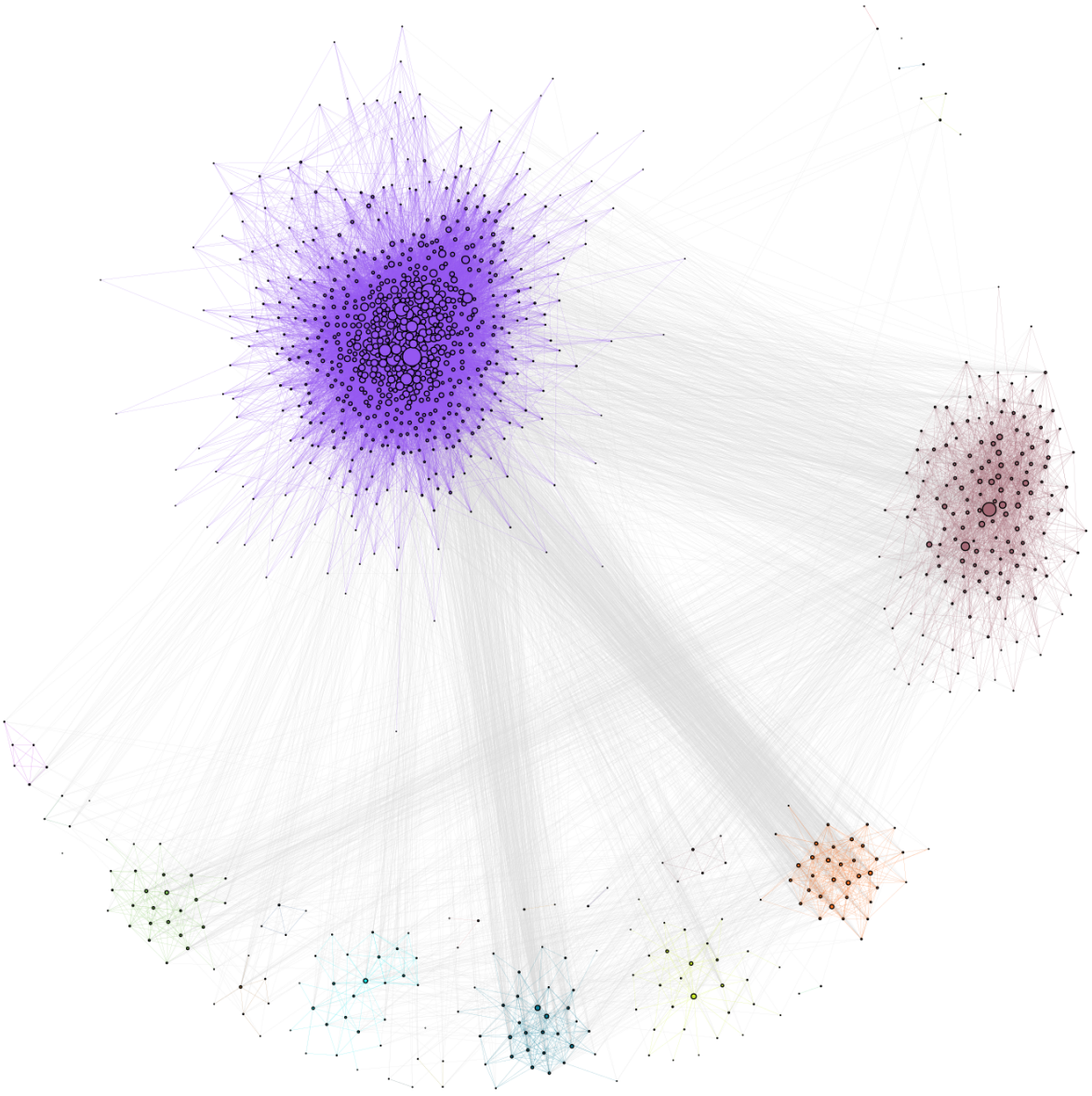


Figure 3.14: Weighted plot - Fruchterman Reingold layout

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.

4 GEODATA ANALYSIS

4.1 Project Overview

1. Goals
    1.1. to show how to deal with geodata in python
2. Plots
    2.1. KPI per country (Basemap)
    2.2. Plot - Chikago Taxi (datashader)

5 CONCLUSION

Include data types that we omit - biological data, food data, describe what you'd like for others to research.

References

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

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

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

[4] 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.

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

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

[7] 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).

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

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

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

[11] Schmuhl, M. *graphopt*. 2003. URL: `https://web.archive.org/web/20180116173939/http://www.schmuhl.org/graphopt/` (visited on 16/01/2017).

[12] 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.