

# Contents

<b>1 INTRODUCTION</b>	<b>3</b>
1.1 State of the Art in Data Science . . . . .	3
1.2 Goals of the Research . . . . .	3
1.3 Thesis Overview . . . . .	3
<b>2 TABULAR DATA ANALYSIS</b>	<b>5</b>
2.1 Project Overview . . . . .	5
2.2 Data gathering (pandas-datareader) . . . . .	6
2.3 Traditional stock visualizations (matplotlib) . . . . .	7
2.4 Stock correlation matrix visualization (matplotlib) . . . . .	8
2.5 Predictive analysis with a LSTM neural network (keras) . . . . .	9
<b>3 GRAPH DATA ANALYSIS</b>	<b>10</b>
3.1 Project Overview . . . . .	10
3.2 Preprocessing and igraph creation . . . . .	11
3.3 Community detection (igraph) . . . . .	12
3.4 Plotting large graphs (datashader)	13
3.4.1 Simple plot . . . . .	13
3.4.2 Plotting communities . . . . .	13
3.5 Plotting small graphs (igraph) . . . . .	14
3.5.1 Goals and section overview . . . . .	14
3.5.2 Regular plot . . . . .	14
3.5.3 Community plot . . . . .	20
3.5.4 Weighted community plot . . . . .	24
<b>4 GEODATA ANALYSIS</b>	<b>27</b>
4.1 Project Overview . . . . .	27
<b>References</b>	<b>28</b>

# Temporary Notes

■ Use stuff from the presentation . . . . .	3
■ Rewrite everything . . . . .	3
■ 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) . . . . .	13
■ citation needed . . . . .	14
■ move save_fname to the dictionary too . . . . .	15
■ add visual comparison . . . . .	16
■ list the number of nodes . . . . .	20
■ cite . . . . .	24

## 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<sup>1</sup>. 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.

---

<sup>1</sup>New York Stock Exchange

## **2 TABULAR DATA ANALYSIS**

### **2.1 Project Overview**

1. Goals
  - 1.1. Describe the project
2. Obtaining the data (pandas-datareader)
3. Traditional stock visualizations (matplotlib)
4. Stock correlation matrix (matplotlib)
5. LSTM training (keras)

## **2.2 Data gathering (pandas-datareader)**

## 2.3 Traditional stock visualizations (matplotlib)

## 2.4 Stock correlation matrix visualization (matplotlib)

## **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

## **3.2 Preprocessing and igraph creation**

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

### **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)

### 3.4.2 Plotting communities

## 3.5 Plotting small graphs (igraph)

### 3.5.1 Goals and section overview

#### Goals

1. Show how to create a regular graph plot, and consider the idea that its simplicity may make it harder to read
2. Show how to create a graph plot with marked communities on it, and compare it to the regular plot in terms of ease of read
3. 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 2000 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.

citation  
needed

### 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 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.

```

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

```

Listing 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 those kwargs in a separate dictionary, since it takes away from the dissarray of having to include many options into one function call.

move  
save\_fname  
to the  
dictionary  
too

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<sup>2</sup>:

- Circle layout
- Star layout
- Grid layout
- Fruchterman Reingold layout
- Fruchterman Reingold grid layout
- DrL layout
- Graphopt layout
- Kamada Kawai layout
- Large Graph layout
- Sugiyama layout
- Random layout
- Reingold Tilford layout (for trees)
- Bipartite layout (for 2-layer graphs)

add visual  
comparison

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 extenstions (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'
```

---

<sup>2</sup>Visual comparison is available at the page

Listing 2: Creating a style dictionary

### Plotting and saving the resulting plot to a file

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

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

Listing 3: Plotting the image

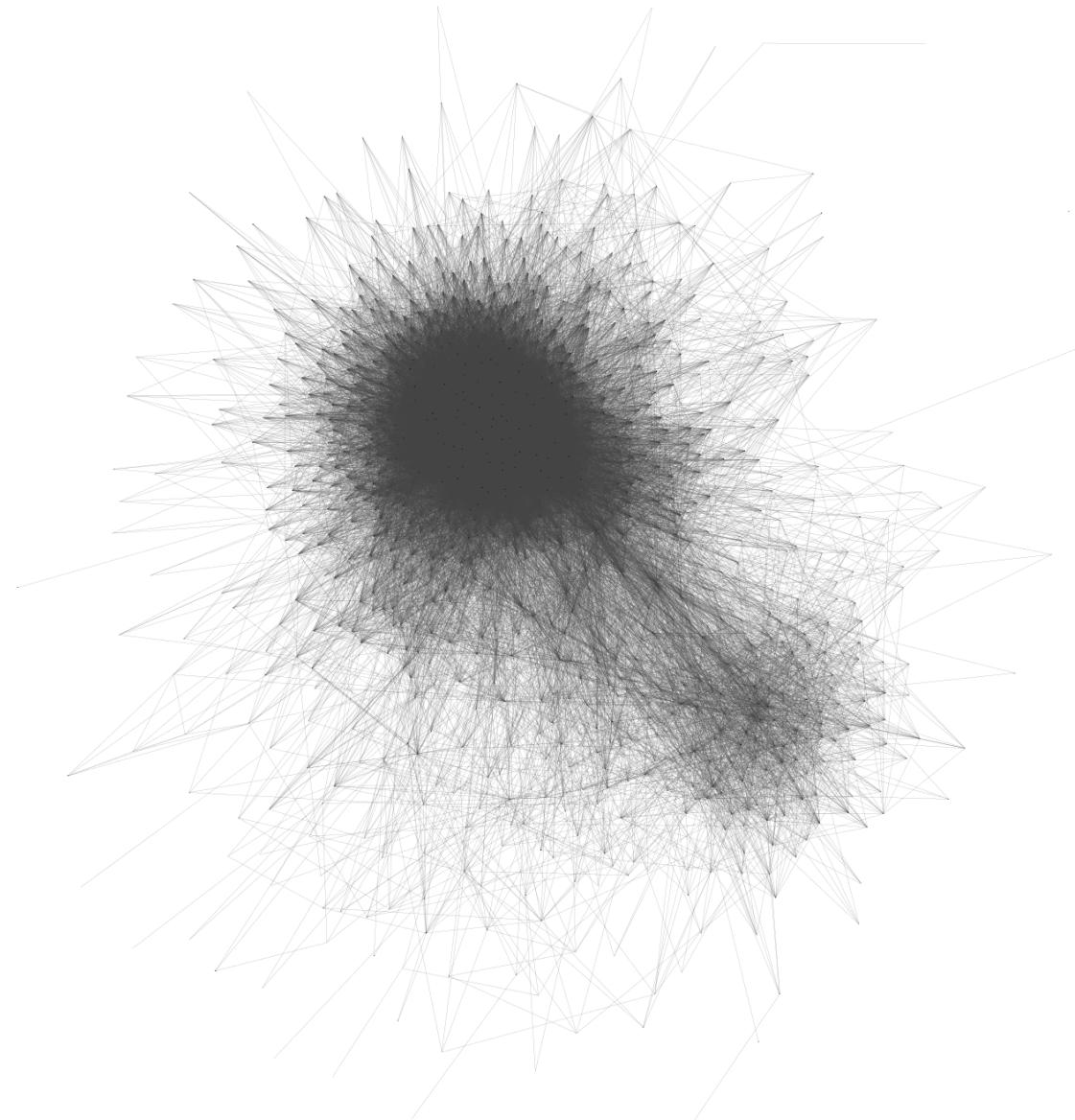


Figure 1: Regular plot (Fruchterman Reingold layout)

## Reviewing the results

While igraph makes it quite easy to plot regular plots like these, they don't provide one with much insight into the data itself. While 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. Furthermore, most of them look just black balls and are quite chaotic, which is a pretty common problem when dealing with plotting graphs.

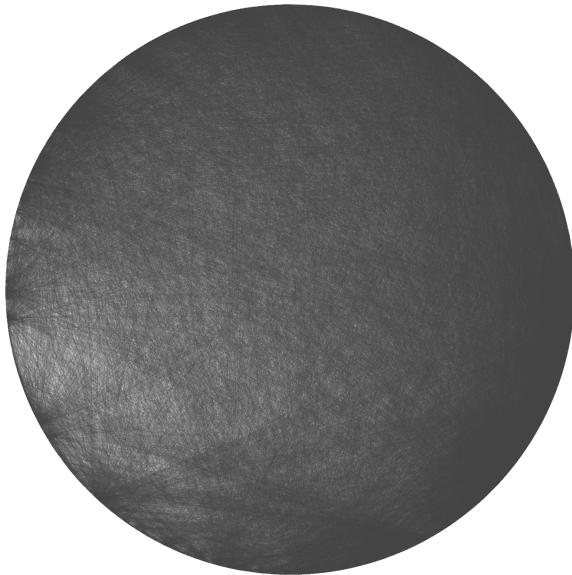


Figure 2: Circular layout

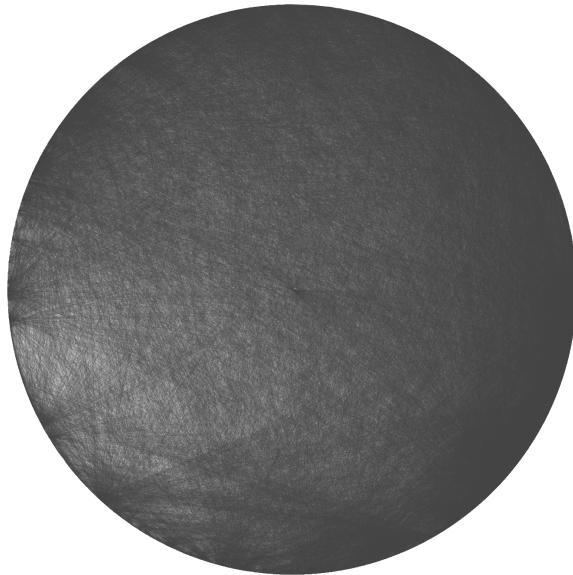


Figure 3: Star layout

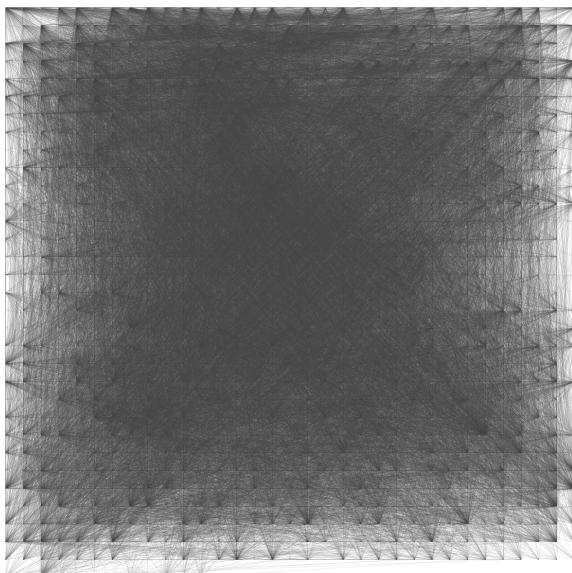


Figure 4: Grid layout

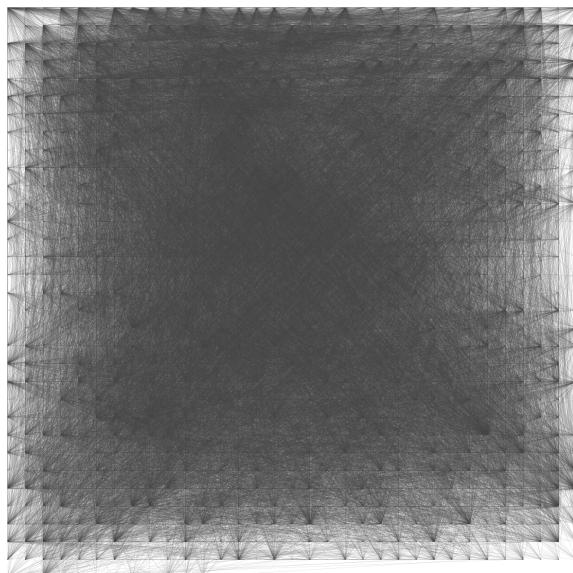


Figure 5: Fruchterman Reingold grid layout



Figure 6: DrL layout

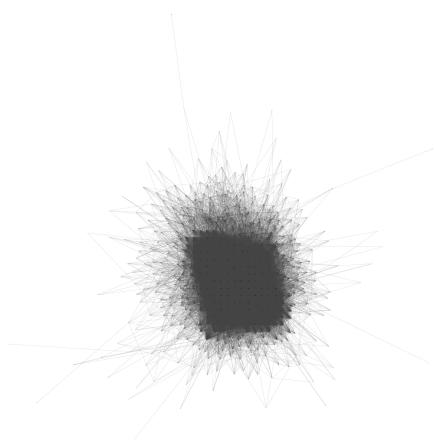


Figure 7: Ghopt layout

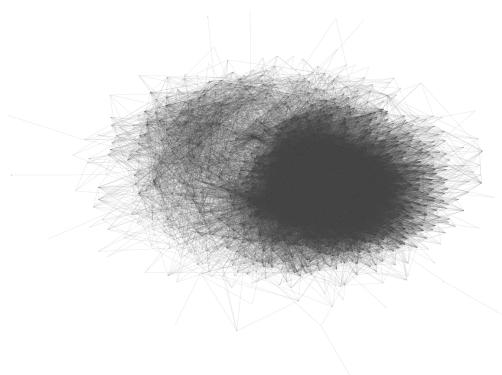


Figure 8: Kamada Kawai layout

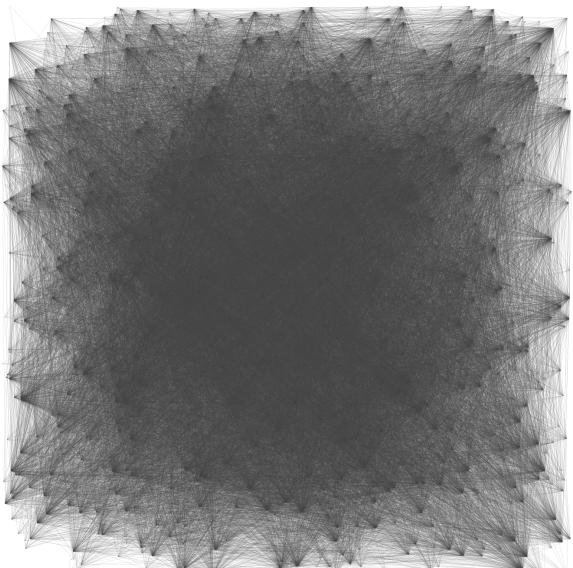


Figure 9: Random layout

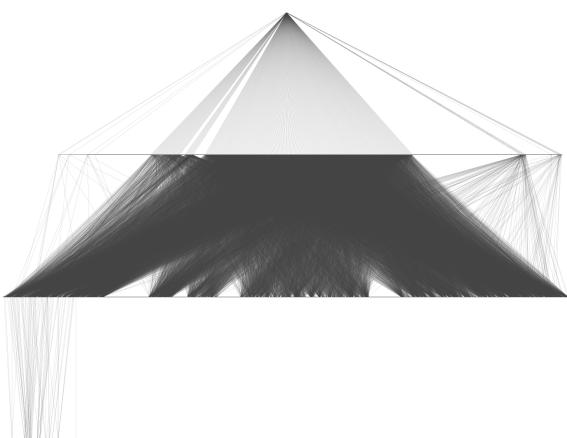


Figure 10: Sugiyama layout

### 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 (only `nodes`), so we can use the `infomap` clustering algorithm on it right away.

list the  
number of  
nodes

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

Listing 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 vertex with their community label. Some of those techniques are only effective when applied to very small graphs (like labelling), while others are a better fit for a medium-sized graph like the one used in this example.

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

Listing 5: Initializing color and weight lists

Something about the algorithm here.

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

Listing 6: Assigning color to vertices

```
15 # Assigning the edge color and weight based on the vertices it connects
16 # Adding weights will make the group separation more visible
17 for edge in hdg_subgraph.es:
18     if hdg_membership[edge.source] == hdg_membership[edge.target]:
```

```

19     edge_colors.append(vert_colors[edge.source])
20     edge_weights.append(3 * hdg_vcount)
21 else:
22     edge_colors.append('#bdbdbd')
23     edge_weights.append(0.1)

```

Listing 7: Assigning color to edges

## Styling the resulting plot

Something about using graph properties to style the plot.

```

24 # Styling the plot - graph properties
25 hdg_subgraph.vs['color'] = vert_colors # Adding color as a vertex 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 8: Styling using graph properties

Something about using the style dictionary to style the plot.

```

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'] = (500, 500)
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['palette'] = ig.PrecalculatedPalette(community_colors)
40 # hdg_comm_style['mark_groups'] = True # Activate if you want to delineate the groups

```

Listing 9: Styling using a style dict

## Plotting and reviewing the results

Something about the plot.

```

41 # Plotting
42 save_fname = 'plot_infomap.svg'
43 ig.plot(hdg_imap, save_fname, **hdg_comm_style) # mark_groups=True

```

Listing 10: Saving the plot to a file

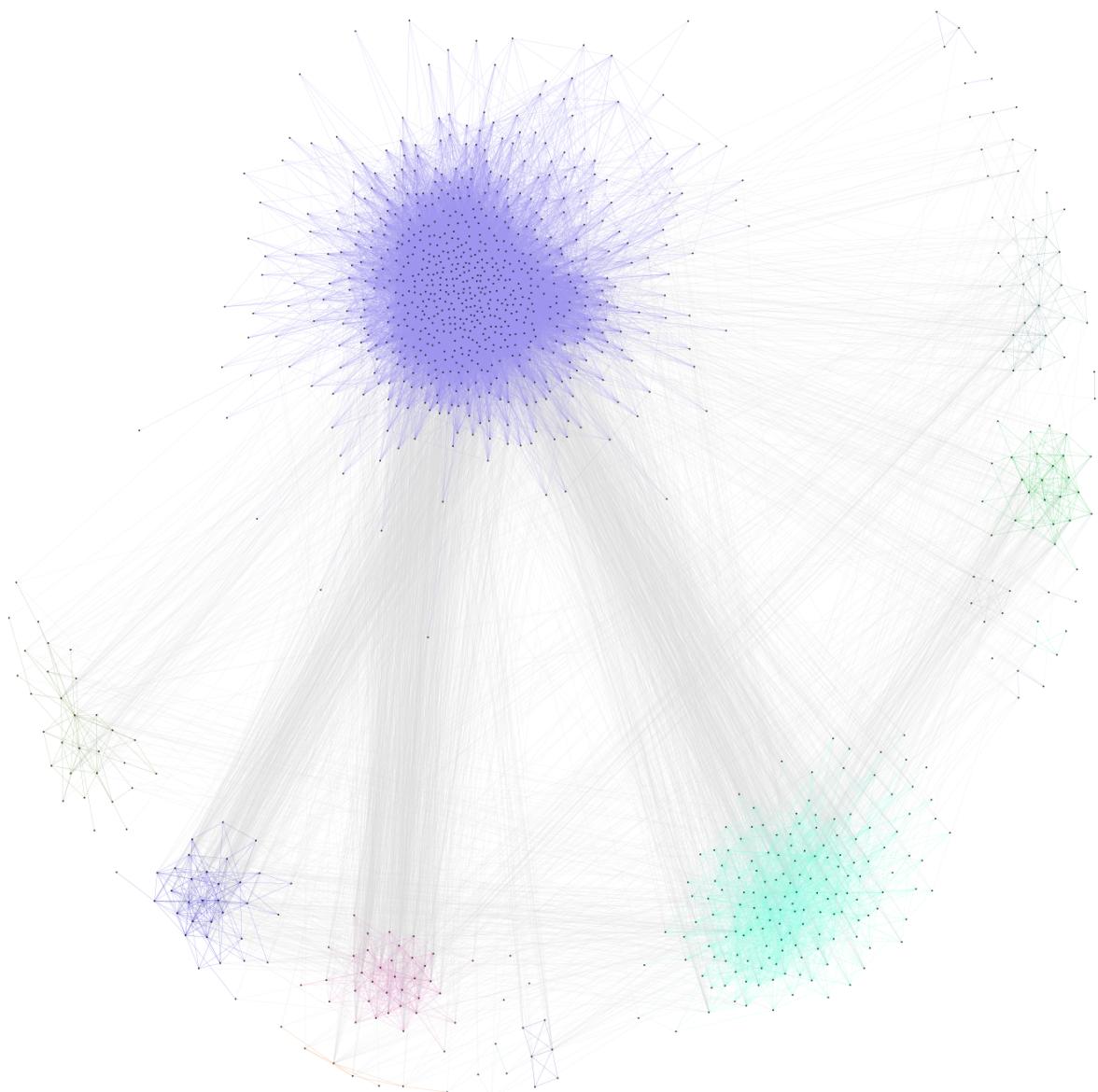


Figure 11: Community plot

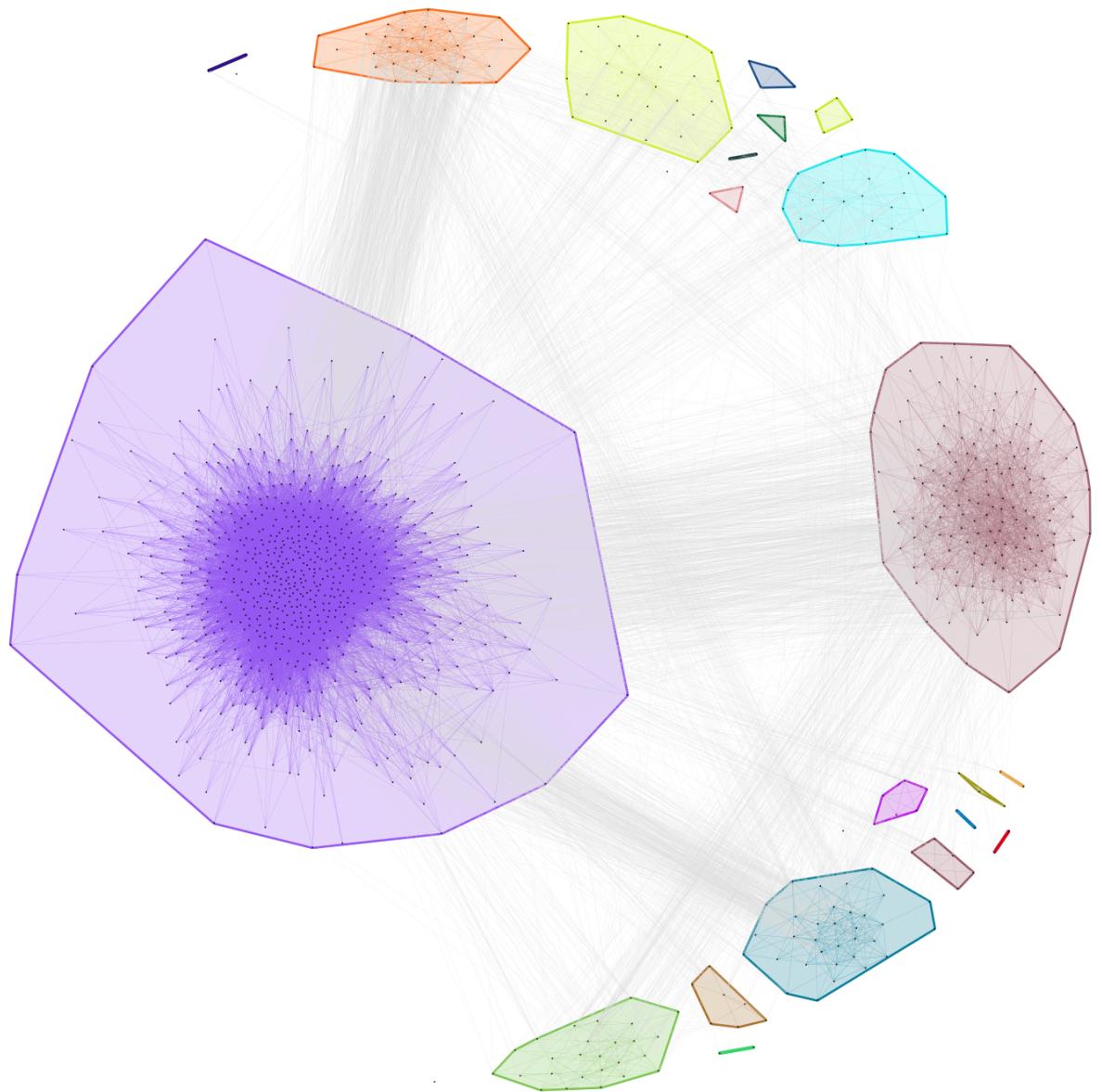


Figure 12: Community plot - marked groups

### 3.5.4 Weighted community plot

#### Pagerank application

Pagerank is an algorithm developed by google.

cite

```
1 hdg_pgrank = hdg_subgraph.pageRank()  
2 hdg_pgrank_arr = np.array(hdg_pgrank)
```

Listing 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 regular community plot, with an addition of ...

```
3 hdg_comm_style = {}  
4 hdg_comm_style['layout'] = \  
5     hdg_subgraph.layout_fruchterman_reingold(maxiter=1000,  
6                                                 weights=hdg_subgraph.es['weight'],  
7                                                 area=hdg_vcount**3,  
8                                                 repulserad=hdg_vcount**3)  
9 hdg_comm_style['bbox'] = (500, 500)  
10 hdg_comm_style['vertex_size'] = hdg_pgrank_arr * 3000  
11 hdg_comm_style['vertex_shape'] = 'circle'  
12 hdg_comm_style['edge_width'] = 0.1  
13 hdg_comm_style['palette'] = ig.PrecalculatedPalette(community_colors)
```

Listing 12: Styling using a style dict

#### Plotting and reviewing results

```
14 save_fname = 'plot_pagerank_fg.svg'  
15 ig.plot(hdg_imap, save_fname, **hdg_comm_style)
```

Listing 13: Saving the plot to a file

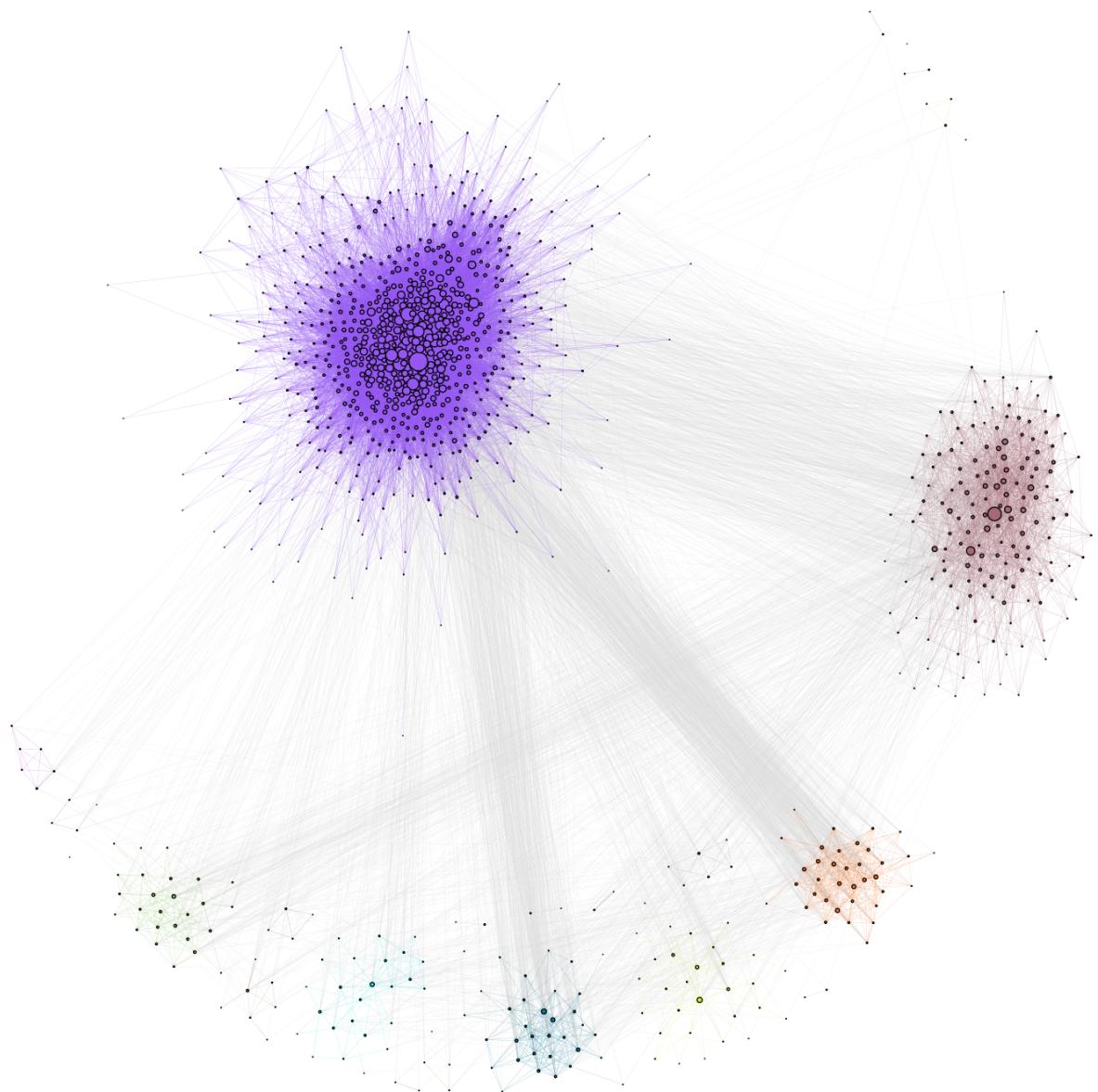


Figure 13: Weighted plot - Fruchterman Reingold layout

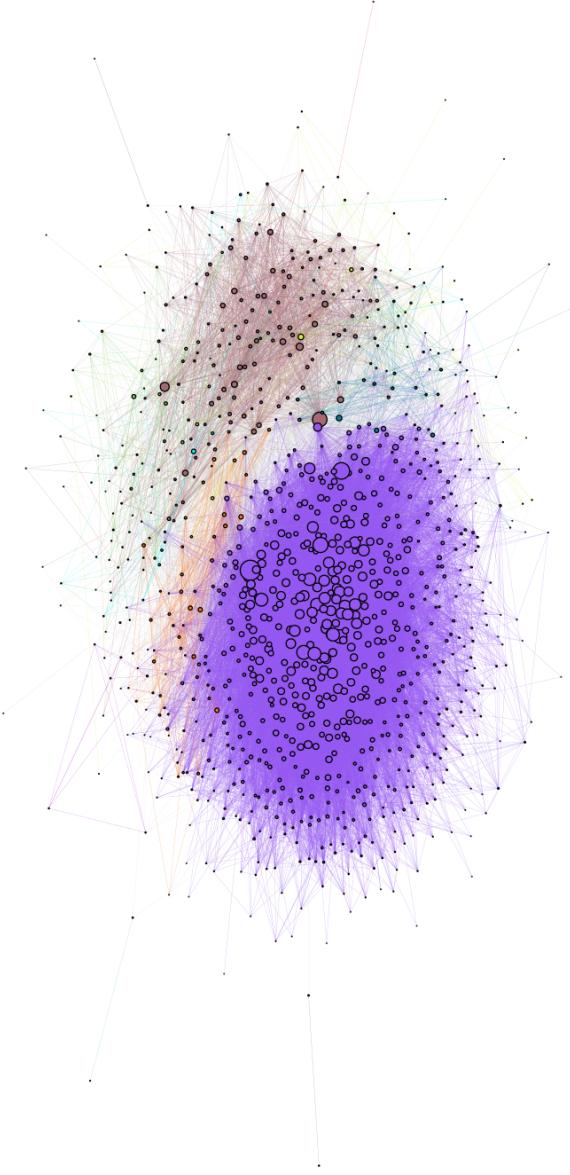


Figure 14: Weighted plot - Kamada Kawai layout

## **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 - KPI per state in a country (libnamehere)
  - 2.3. Plot - Chicago Taxi (datashader)

## References

- [1] Albert Einstein. Zur Elektrodynamik bewegter Körper. (German) [On the electrodynamics of moving bodies]. *Annalen der Physik*, 322(10):891–921, 1905.
- [2] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The L<sup>A</sup>T<sub>E</sub>X Companion*. Addison-Wesley, Reading, Massachusetts, 1993.
- [3] Jean Francois Puget. The most popular language for machine learning and data science is à€.