A Graph Visualisation Toolkit for Scraping Extremist Circles on Instagram

Sam[redacted]

15th of April, 2025

Abstract

An evident rise in far-right activity both online and offline has been a major topic of debate for academics in social fields in recent years (Baele et al., 2023; Munn, 2019). It is agreed that active measures must be taken to aid in reducing the effectiveness of online radicalisation methods, by either removing radicalisation content or informing the public about propaganda strategies (Ganesh & Bright, 2020; Saleh et al., 2024). In this project, a two-part software package was created in the context of visualising scraped data from accounts promoting far-right talking points on Instagram. The first part of this package is a tool that runs a breadth-first scrape on a target account, collecting information about the users scraped, this document also discusses the challenges faced when scraping user data from social media. The second part is a data visualisation toolkit that employs a wide range of different social network analysis methods such as force-directed layouts, centrality measures and community analysis. Together, this package aims to provide professions such as Journalism and Law Enforcement with a simple tool that can aid the process of tracking and understanding extremist groups.

Acknowledgements

There are many people I would like to thank for helping me in both the creation of this document, and my time on my Computer Science undergraduate degree. The first is my personal mentor and project supervisor Dr. [redacted] who I would like to thank for giving me swift and helpful feedback whenever I needed it, and supporting me along the way.

I would like to thank my wonderful girlfriend [redacted] for keeping my sanity intact through her endless support and compassion for me on the long nights spent on this project and throughout this degree. To my Mum, my Dad and my sister [redacted] Thank you all very much for supporting me as much as you could throughout my degree! To [redacted] [redacted] [redacted] and [redacted] Thank you all for the laughs after long days at work! And to Freeze, Masie, Venus, Nova, Indi and Lottie: thank you for reminding me that pets do not have to deal with things such as dissertations.

Contents

List of Figures

1 Introduction

1.1 Project Motivations

The rise of social media over the last three decades has dramatically changed the ways in which we socialise, as it is now a part of our day-to-day lives. Mølmen and Ravndal (2023) discussed that such a change in our society has also affected how extremist groups operate, and how they aim to radicalize people. They found that the anonymity and decentralized nature of the internet appeals to the extremist, making it easy to spread radicalization content online and hide behind different aliases and websites (Mølmen & Ravndal, 2023). The concept of online radicalization has concerned policy-makers and academics alike, with its affects being demonstrated recently through riots and lone actor terror attacks (Herath & Whittaker, 2023). While the methods of countering group radicalization akin to the 2024 Southport riots are crucial in curbing the recent rise in terror cases, this project focuses instead on the visualisation and analysis of the large-scale social media networks that can radicalise one into committing acts of lone-actor terrorism (Munn, 2019).

There has been a marked rise in terror attacks committed by lone actors over the last decade, with many being found to have been radicalised online (Hamilton, 2018). Gill (2015) finds that lone actor terrorists can be ideologically motivated by many factors, with far-right and islamic fundamentalist beliefs being popular motivators in recent years. They also discuss how lone actor terrorists will not prescribe themselves as members of any extremist organisation, despite potentially ideologically aligning with different groups . While lone actor terrorism is not a new phenomenon (with incidents dating as far back in the United Kingdom as 1894 (Hewitt, 2024)), academic literature attributes the rise in attacks to the internet as a radicalising factor (Aryaeinejad & Scherer, 2024). This is a significant issue facing modern-day counter terrorism, as Gill (2015) writes that beyond the aforementioned traits, the individual attributes of different lone actor terrorists vary greatly. As such, the primary strategy for preventing lone actor terror attacks should not involve predicting the attacks themselves due to their unpredictable nature — instead, we should focus on countering and eliminating the radicalising factors that may inspire one to act in the first place.

Despite the stereotypical concept of the lone actor terrorist being socially isolated, research conducted by Aryaeinejad and Scherer in 2024 on the behalf of the National Institute of Justice concludes that both social networks and online subcultures play a "significant role" in the radicalisation of a lone actor terrorist. Aryaeinejad and Scherer reinforce this by explaining that the process of radicalisation itself is a social act, including in cases where an individual may be inspired to act due to a lack of social interaction, or a search for a social circle where they fit in. However, they also note that radicalisation puts a strain on an individuals' social bonds - they are more likely to cut ties with more casual friends and will instead seek out people who affirm their beliefs. This is important to consider when discussing the process of radicalisation as it identifies that anybody can become a lone actor terrorist, despite whether or not they are fully socially withdrawn. It also demonstrates that social theories and models that can be applied to radicalisation offline may also be applicable online.

The aforementioned anonymity that the internet provides creates a perfect storm for the propagation of disinformation, created by political agitators. This misinformation, and other forms of propaganda create dozens of micro-causes that may inspire one to act (Vasist et al., 2024). A prime example of this would be the riots that followed the fatal stabbing of 3 children in Southport, Merseyside - an incident which left 10 more injured. These riots were sparked by the false claim that the perpetrator of the stabbing was an Afghan asylum seeker, whom arrived in the United Kingdom via a small boat. The claim originated on a controversial news aggregator website named "Channel3Now", the claim was then spread — wittingly or not — across social media. A former head of the MI6, along with several British media outlets have claimed that Channel3Now was affiliated with Russian disinformation efforts (Shirreff, 2024; Wilmot, 2024), and was a deliberate attempt to exploit the United Kingdom's already heated debate on immigration. However, an investigation headed by the BBC (Spring, 2024) found no evidence to back this up. Nonetheless, the months proceeding the riots left the incumbent's cabinet reviewing their current approach to extremism, where they considered changing their focus from dangerous ideologies to certain concerning behaviours, such as extreme misogyny, fascination with violence and conspiracy theories (Zeffman et al., 2025).

The effectiveness of radicalisation through misinformation online should not be understated. Hope not Hate's 2024 "state of hate" report found that in 2023, 23 far-right "activists and sympathisers" were convicted on terror-related offences in the United Kingdom ("Hope not Hate", 2024). They also found that 2023 saw a 20% rise in "anti-migrant activity" compared to 2022, and anti-migrant demonstrations increased 18-fold. In order to counter the effectiveness and risks of decentralised,

online radicalisation two main strategies could be implemented:

- The first strategy involves the government identifying extremist propaganda circles on social media platforms and forcing the host of the platform to ban the accounts responsible. This would be an effective method of minimising the extent of the normalisation of extremist talking points, but Ganesh and Bright (2020) debates that it has drawbacks in that it would take a large amount of time and government resources to reliably track and identify bad actor accounts, and may affirm extremists with anti-state views.
- An alternative strategy involves the concept of "active inoculation", which Saleh et al. states is where the population is warned about strategies that extremists may use and the disinformation that they spread (this is not dissimilar to inoculation in virology) (Saleh et al., 2024). They found that this method has proven to be effective in decreasing people's susceptibility to extremist propaganda, although suffers the same issue as the former in that it would require a large amount of time and effort to keep the population well informed against always-adapting threats.

Current literature on these radicalisation methods recognises the need for a data-centered approach to understanding the mechanics of online radicalisation. One approach that has been used for decades in security and intelligence organisations is that of Social Network Analysis (SNA). SNA applies sociology to network and graph theory, allowing for the analysis of the connections between groups and/or organisations with data visualisation (Martino & Spoto, 2006). The American NSA has used social network analysis in the past to map out terrorist organisations, or any other groups deemed as a threat to national security using electronic surveillance to gather data (McCulloh et al., 2013). Once an appropriate amount of data is collected, different centrality measures can used to determine the relative importance of different members within these groups (Martino & Spoto, 2006) - allowing for decapitation attacks to be launched against key members.
The ability to conduct social network analysis on popular social media in a user-friendly manner is crucial to the success of the previously mentioned methods of countering extremism. Intelligence would be able to quickly identify groups of accounts posting harmful radicalisation content, and ban them all together. Different metrics for analysing trends could empower journalists to write articles on emerging groups or dog-whistles, an effective method of active inoculation that keeps attempts at shifting the overton window in the public zeitgeist. There exists a technical barrier between the complex statistical analysis behind SNA and the poicy-makers who need these metrics to help counter extremism, and data visualisation software can help to lessen that barrier (Burruss & Lu, 2021).

1.2 Project Objectives

This project aims to produce:

1. A tool that is able to scrape data from Instagram and store it in a given data format, where:
   (a) The tool is able to successfully evade Instagram's scraping detection algorithms.
   (b) The tool collects different metrics from Instagram accounts, such as Username, Following, Bio, and whether or not the account is private.
   (c) The tool is able to gather anonymous data if required to, where all personally identifying information is obfuscated or not collected.
   (d) The tool respects the privacy of accounts marked as private, and will not scrape any data further than the private account's username.
2. A data visualisation tool that allows users to easily conduct social network analysis, where:
   (a) The tool generates interactive network visualisations from the data scraped.
   (b) The tool allows users to upload their own scraped data, for themselves or others to see.
   (c) The tool has some security measures that prevent users outside the organisation to view the scraped data.
   (d) Different centrality measures are used to see and rank the relative significance of users in the network.
   (e) Community detection is used to visualise a separation between groups of users.
   (f) The user can rearrange the graph into different layouts.

To address the gap highlighted above, the end goal of this project is to create a data visualisation tool that makes social network analysis much easier for the layman to understand. This tool should be able to display an interactive network graph of users on a given social media platform, where each node is a user and each edge is a single or omni-directional connection based on user A "following" user B. The choice of displaying edges dependant on who the user is following is multi-faceted. First, user A following user B implies that user B is admired by user A - on many platforms, user A will be shown more content from user B (Bakshy et al., 2015). If user B is a smaller account, then a personal relationship is implied between the two accounts (Ellison et al., 2007). Furthermore, semi-popular accounts are much more likely to have a larger following than the amount of accounts that they themselves are following, thus reducing the size of the data-set and the risk of any ongoing scraping operations being detected - a risk that will be discussed in the future.

The scale of the graphs produced by this program is expected to be large and unwieldy, so some form of culling should be introduced to reduce the size and the computational complexity of the network. The first method involves automatically hiding all "edge" nodes on the graph (i.e. all nodes that have an in-degree of 1 and an out-degree of 0), which focuses the graph solely on nodes that have had their following scraped, or are being followed by multiple of these accounts. To expand on this, users should have the ability to disable this filter or create their own filtering rule that hides all nodes with an in-degree lower than X and out-degree lower than Y. Also, user's should be able to click on a given user to hide all other users that are not connected to the selected account. These features combined will benefit the user experience by improving frame-rate, reducing clutter and preventing information overload.

Centrality measures are an essential metric for determining the relative importance of different nodes on a network graph (Freeman et al., 2002). In terms of this project, the size of each node should be determined by some centrality detection algorithm of the user's choice. Controlling the size of different nodes is a simple yet effective way of grabbing the user's attention and demonstrating the importance of the largest accounts. Another form of metric to be considered is that of community detection. Community detection algorithms aim to detect groups of nodes that share similar values, and are instrumental in revealing the structure of dense networks such as the ones generated by this tool (Fortunato, 2010). Different communities could be displayed to the user with different colours for each node, which demonstrates a clear separation between groups.

An adequately-sized dataset should also be obtained for this project, in order to test the tool's functions and measures accurately. The target demographic for this project is Instagram users posting extreme right-leaning content - Instagram is the target platform as it is a good example of social media using machine learning for content delivery and moderation. It is known to recommend content that is legal but harmful, such as extremism and misinformation (Whittaker et al., 2024). The demographic of right-wing extremists was chosen due to its aforementioned prevalence in recent years. In order to gather this dataset, a secondary tool should be developed that is able to scrape content from Instagram. The user should be able to give the tool a "start" account, and then the tool will recursively scrape the following (and basic details) of each account reached, in a breadth-first manner. Of all deliverables in the project, this one brings the most risk - while the EU allows for scraping in cases where the results fall within the public interest (European Parliament & Council of the European Union, 2016), it falls within Meta's interests to prevent automated usage of their platform where possible. It will be necessary to implement several different checks to make sure that the scraping operation is not compromised.

2 Background

As previously mentioned, social network analysis can been seen as a gateway to understanding how extremist groups operate both online and offline (Perliger & Pedahzur, 2011). The field of social network analysis is comprised of many different methods of analysis, two notable and essential methods being the measure of Centrality and the analysis of communities within the graph. Measuring centrality allows us to understand which nodes in the graph are the most active, exert the most control and are the most independent. Analysis of communities in the graph helps to reveal the underlying structure of the graph — which nodes most likely belong to groups of other nodes, and how these groups interact with eachother.

Moreover, the process of visualising these vast networks of nodes is another key part of this project. It is necessary to understand the different methods of visualising network graphs in order to build a tool that visualises them. To do this, comparisons will be made to the popular graph visualisation toolkit Gephi, which on top of including centrality measures and community analysis also allows

for a multitude of different layout methods.

2.1 Centrality in Social Networks

Freeman et al. provide numerous examples of how the analysis of centrality (and decentrality) can be applied to different scenarios in their 1978 essay "Centrality in social networks — conceptual clarification". They note how it can be used to measure political integration in society, communication paths for urban development and even patterns of diffusion in steel. One significant case provided was Rogers, 1974, where different centralities were modeled in inter-organisational relations. Its significance is derived from how Rogers found that the centrality of a given organization was one of its most predictable attributes. However, Freeman et al. found that there was an inconsistent definition of centrality being used in literature, leading improper measures being used in the wrong situations. The aim of their essay was to clarify and resolve conceptual problems with centrality, and provide examples of different measures and how they apply to different scenarios. This essay is useful for this project as it will aid in realising which centrality measures will be the best to include in the application, and what they mean for the data being visualised.
In the essay, Freeman et al. divide the activity of centrality analysis into three main categories, and assign different measures to them:

- Freeman et al. note that the most "intuitively obvious" measure of centrality is that of degree centrality. In their essay, they establish that they are describing non-directed graphs and as such, degree centrality is the overall count of adjacent nodes for each node. They define degree centrality as:

$$C_D^{'}(P_k) = \frac{\sum_{i=1}^{n} a(P_i, P_k)}{n-1}$$

  Where:

  - n is the count of all nodes in the graph.
  - $a(P_i, P_k)$ is 1 if there exists an edge connecting P_i and P_k — otherwise it is 0.

  In a review of previous examples of degree centrality being used, Freeman et al. write that nodes on the graph that have the greatest degree centrality are the most *active* members of the graph, being known to many other members and also being crucial in the flow of information throughout the network.
  This centrality measure could be easily adapted to work in this project. The primary change that would need to be made is the subdivision of degree centrality into three more specific measures, due to the fact that the edges in the graphs being generated will be directed. There should be centrality measures for in-degree, out-degree and the sum of the previous two. Due to this separation, Freeman et al.'s theories on how central nodes are the most active may not be as accurate. For example, nodes with a high in-degree but low out-degree may receive large amounts of information, but are not responsible for a large output of information.

- While activity may be an accurate measure for centrality in a network, Freeman et al. verify that many academics have also considered that nodes with the highest degree of *control* over a network could also be the most central. They write that determining control of a node over a network cannot be done with degree centrality, and instead the nodes that lie on the shortest path between any two nodes on the graph — or, those with the greatest **betweenness** — hold the most control. Freeman et al. write that this is because these nodes could effectively cut off the flow of information, or compromise it if they so wished to. They write a lengthy arithmetic process for determining the betweenness centrality of a given node (P_k):

  1. First, the function $b_{ij}(P_k)$ determines the partial betweenness of a point in the graph by dividing the number of geodesics[1] linking two nodes that contain P_k by the number of total geodesics:

$$b_{ij}(p_k) = \frac{g_{ij}(P_k)}{g_{ij}}$$

     Where:

     - $g_{ij}(P_k)$ is "the number of geodesics linking p_i and p_j that contain p_k".
     - g_{ij} is the total number of geodesics linking p_i and p_j.

[1] We define a geodesic as the shortest path in a network that links two given nodes.

2. Next, we use $b_{ij}(P_k)$ to determine the "overall centrality" of any point by calculating the sum of $b_{ij}(P_k)$ for all pairs of i and j:

$$C_B(P_k) = \sum_{i<j}^{n}\sum^{n} b_{ij}(p_k)$$

3. Finally, we can establish that the *relative* centrality of a point in the graph can be shown as:

$$C_B^{'}(P_k) = \frac{2C_B(P_k)}{n^2 - 3n + 2}$$

Where

$$\frac{n^2 - 3n + 2}{2}$$

Is known to be the maximum value taken by $C_B(p_k)$(Freeman, 1977).

Whilst it is clear that some measure for betweenness centrality is crucial for the analytic capabilities for this project, the computational complexity of the process provided above is undeniable, especially when considering larger graphs with thousands of nodes. There should be further research done on less complex algorithms that could be used to achieve the same result. Otherwise, it is necessary to implement some form of pre-processing or multi-threading into the tool itself.

- The final centrality measure described in the essay is that of closeness centrality. Freeman et al. describes closeness centrality as how "close" a given node is in the graph to all other nodes. They note that Leavitt, 1951 has a unique perception of closeness centrality, where he writes that the closeness of a given node to all other nodes demonstrates its relative "independence" from other nodes. A node's "independence" makes it central to the graph because it is "not dependent upon others as intermediaries or relayers of messages". Conversely, we may also define the node with the highest closeness as the node wherein messages will take the least time to propagate across the network. Freeman et al. uses a definition for a node's relative closeness centrality across a network by (Beauchamp, 1965):

$$C_c^{'}(P_k) = \frac{n-1}{\sum_{i=1}^{n} d(P_i, P_k)}$$

Where we consider $d(P_i, P_k)$ to be the number of edges linking the geodesic between P_i and P_k — Or in layman's terms, the distance between P_i and P_k.
There are multiple reasons why a measure like this is important when considering this project. Closeness centrality can be used to determine the "independence" of any node in the graph from its others, allowing for another valuable metric to be gathered. It may also be used to improve the overall performance of the graph, where nodes that have the lowest (or highest) closeness centrality may be culled per the user's request. There are still concerns over the complexity of an algorithm applying this, but compared to the algorithm for betweenness centrality this process seems a lot more computationally relaxed.

(Freeman et al., 2002) emphasizes the importance of the proper use of centrality measures in the analysis of networks, and accurately divides the use of centrality measures in literature at the time into three distinct categories. It clearly outlines how these different categories allow for different assumptions surrounding the network to be taken. However, the algorithms used to determine both closeness and betweenness centrality are complex, and may be difficult to calculate for graphs with thousands of nodes, such as the ones generated for this project. As such, it should be a priority to search for less complex algorithms that are able to obtain the same result, or alternatively a multi-threaded solution to this computational problem. Furthermore, many could consider the essay to be outdated as it was written in 1978. While much of the information included still applies to modern day social network analysis it is important to consider centrality measures that have been developed since this point, and what metrics may be gathered from them. There should also be a clear distinction made to the user about what these different centrality measures mean for the graph, as Freeman et al. found that the original problem with centrality measures were their often improper application. This relates to the general problem with the project of making the field of social network analysis understandable to the layman.

2.2 Community Detection with the Louvain Algorithm

Blondel et al. (2008) echo a common issue in social network analysis, wherein the size of networks being analyzed is becoming too big. They write that newer algorithms and methods of analysis are needed in order to reliably and quickly interpret large networks with millions of nodes. Community detection is introduced by Blondel et al. as an aid to the analytical process, by giving the researcher a way to break the graph down into smaller "communities" with a high degree of inter-connectivity. These methods are essential in modern-day social network analysis and can provide useful metrics about the nodes within the communities, but community detection also helps our understanding of the overall structure of the graph that is being analyzed. Blondel et al. divide current community detection algorithms into three categories:

- Divisive algorithms — The process of removing edges from the graph, using different metrics to determine which to remove.
- Agglomerative algorithms — A recursive process that involves merging similar nodes/communities
- Optimization algorithms — Greedy algorithms that aim to maximise a given metric

Blondel et al. state that all three of these categories use the same metric to determine the "quality" of the output partitions of the graph — modularity. Newman (2004) provides a formal definition for modularity, where its output for a partition P is between -1 and 1, 1 represents a higher quality partition and -1 represents a completely chaotic partition. The definition provided by Newman may be written as:

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$$

Where:

- A_{ij} represents the edge weight between i and j.
- k_N represents the sum of the weight of all edges directly connected to node N.
- c_N is the community that N is in.
- The δ function is 1 if both of its parameters are equal, and 0 otherwise.
- m is $\frac{1}{2} \sum_{ij} A_{ij}$

Blondel et al. (2008) discuss using modularity as a goal for an optimisation algorithm, but conclude that it is to computationally expensive to be run multiple times for each node. Blondel et al. (2008) introduce their own community detection algorithm (named the Louvain algorithm), which is an optimisation algorithm using the change in modularity (ΔQ) as a metric. The algorithm is iterative, repeating itself over two phases:

1. In the first step, each node N in the graph is represented as its own community C_N. We consider all neighbors for each node N, and calculate whether the modularity of the community of each neighbor C_M increases if N is moved from C_N to C_M. If none of the potential modularities are above 0, then N is not moved at all. Otherwise, N is moved to C_M. This phase is complete when a local maxima has been reached: there are no further changes that will yield a positive change in modularity. Blondel et al. implements the algorithm ΔQ to find the change in modularity mentioned previously as:

$$\Delta Q = \left[\frac{\sum_{in} + 2k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m} \right)^2 \right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m} \right)^2 - \left(\frac{k_i}{2m} \right)^2 \right]$$

   Where:

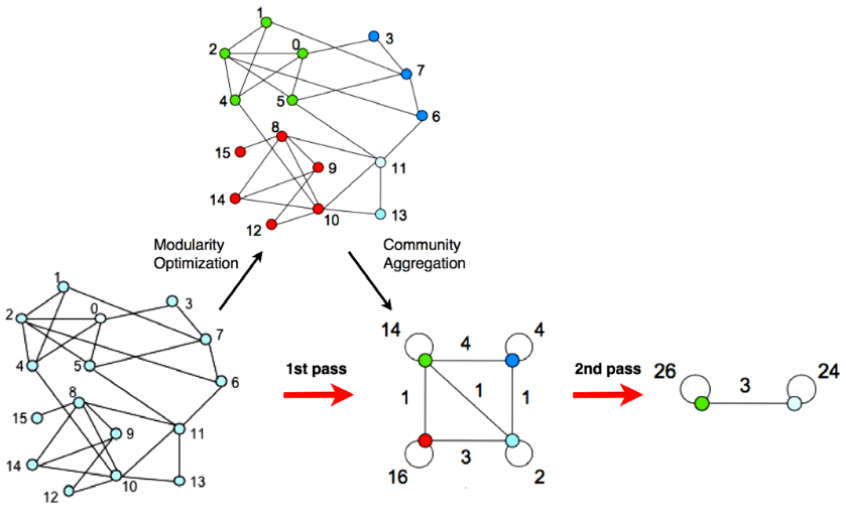
   - $\sum_{in}$ is the sum of all weights of edges inside community C.
   - $\sum_{tot}$ is the sum of all weights of edges that connect to nodes in C.
   - k_N is the sum of the weight of all edges connected to N.
   - $k_{N,in}$ is the sum of the weight of all edges that are connected to N and nodes in C.
   - m is the sum of all weights of all edges in the network.

2. The second phase is much simpler, and less expensive than the first. It involves representing the communities detected in the first phase as individual nodes, with all edges merged. This prepares for a repetition of step 1, where instead of the original nodes being used, collections of these nodes are used.

These iterations should continue until the modularity of the graph has been maximised, or no further changes can be made.

Blondel et al. found that the computational complexity of this algorithm increased in a linear

Figure 1: Visualisation of the Louvain algorithm, created by Blondel et al. (2008).



fashion with input size, citing its speed as one of its advantages. They write that the complexity of the algorithm ΔQ increases with the size of C, but the number of communities decreases drastically with each iteration, which helps to counter this. Community detection algorithms are essential for this project, as they can aid the user in visualizing the separation between the (potentially) tens of thousands of nodes in the network. The speed of the Louvain algorithm makes it a desirable choice for the web-application, as the user will need to run this algorithm each time they open the page — A slow community detection algorithm will make the tool seem clunky and worsen user experience.

However, there exist some drawbacks to this algorithm that should be acknowledged. While Blondel et al. found that — on average — the Louvain algorithm scored a higher modularity with some benchmark data-sets than other community detection algorithms, it should be considered that this paper is now 17 years old, and better algorithms may now exist that this has not been tested against. Furthermore, there exists some criticism of the modularity metric itself. Fortunato and Barthelemy (2007) found that the modularity algorithm does not identify communities that are smaller than a certain value. This is relevant to both the Louvain algorithm and the comparison conducted, as it suggests that modularity results for the first stage of the algorithm may be unreliable. It also suggests that the comparison conducted against other community detection algorithms may be inaccurate, as Blondel et al. (2008) fail to acknowledge this inaccuracy when conducting the comparison.

Despite these issues with the algorithm, Louvain is a good example of a fast community detection algorithm, which fits well within the scope of the data visualisation toolkit this project aims to develop. Although, the concerns around the accuracy of the algorithm suggest that the user should have the option to choose between different community detection algorithms.

2.3 Gephi

Gephi is an open-source graph visualisation and analysis toolkit, built on top of the Apache Netbeans platform[2]. It is downloaded as a portable executable file, and uses OpenGL as a rendering back-end. Created in 2008, Gephi has been used by journalists and academics for a multitude of purposes, such as visualising the dissemination of Islamophobic tweets after the Oslo terror attacks

[2]Gephi — Github. Last accessed on the 21st of March 2025
https://github.com/gephi/gephi

in 2011 (Aouragh et al., 2011), or analysing the ways in which *New York Times* articles associate with each other geographically (Leetaru, 2011). Due to Gephi's applications in different forms of network analysis over the last two decades, it makes for a good benchmark application to test the development of the web application against.
Of course, it should also be acknowledged that Gephi and this project have some significant differences. Gephi is a general purpose graph visualisation tool, whereas this project aims to develop a much more specialised tool, designed solely to analyse social media networks. This will affect the analysis and comparison between the two tools because there will exist features in Gephi that may not be necessary in this project, and vice versa. Furthermore, as previously mentioned, Gephi was programmed in Java and uses OpenGL as its rendering back-end, while this project will be programmed in Javascript and will use WebGL as its back-end. While both renderers utilize the GPU well, WebGL will always be markedly slower due to how it performs security checks on code being executed (such to prevent buffer overflow exploits being possible over the internet)[3]. WebGL is derived from OpenGL ES 3.0, which is intended for use in embedded systems such as smart phones and tablets[4]. Due to this, it is lacking features such as geometry, tesselation and compute shaders which are used thoroughly by its parent system[5].

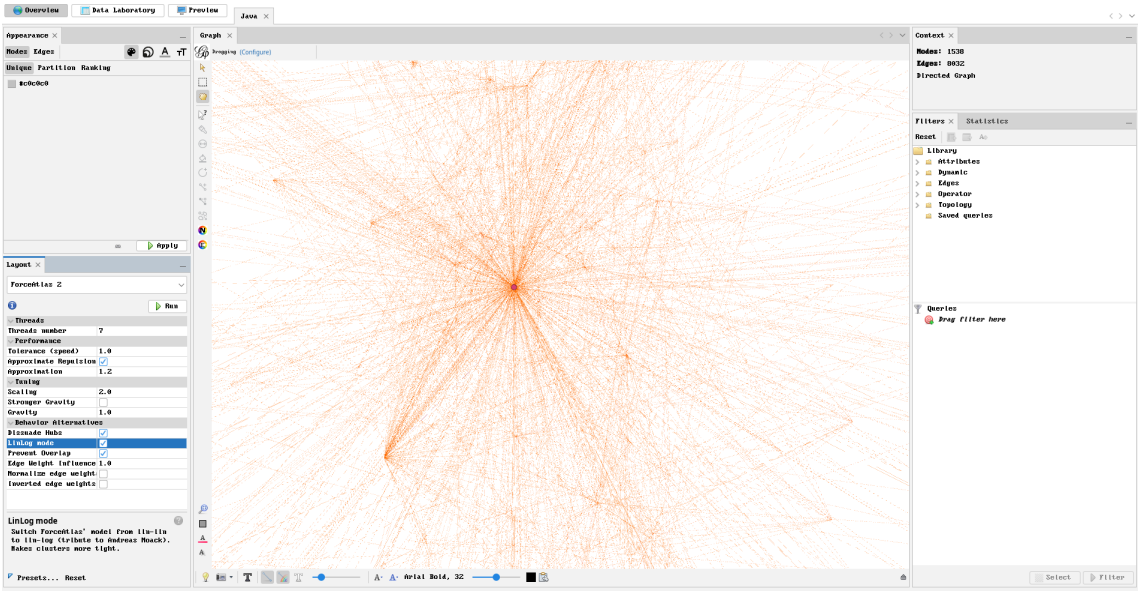


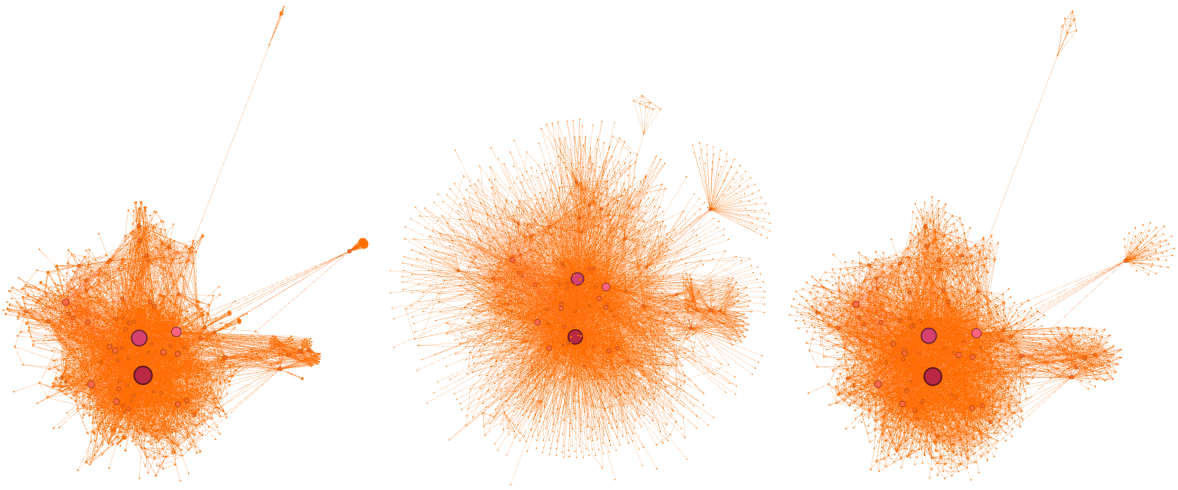
Figure 2: An image of Gephi rendering a visualisation of connections within the Java standard library.

Gephi offers a multitude of different graph analysis features adjacent to what this project plans to implement. The most striking of these, and one not already discussed is the ability to apply different force-directed graph layout algorithms. Kobourov (2012) writes that force-directed graph layout algorithms aim to compute a layout for the graph without any advanced knowledge of the structure of the graph itself. He details that while aesthetically pleasing, these algorithms can aid the user in understanding any underlying structures in the graph; especially larger graphs that may appear "tangled" when given a random layout. Kobourov states that these algorithms can also be known as "spring embedders", due to how they generally work by representing each edge in the graph as a spring, and then applying an iterative process that aims to place each spring in a state where they hold a minimal amount of kinetic energy. Gephi contains many different graph layout algorithms, although three of these stand out the most amongst others:

- Force-atlas 2 is a layout algorithm developed specifically for Gephi (Jacomy et al., 2014). Jacomy et al. (2014) outline an algorithm that follows the same instructions as other spring embedders, but each node is repulsed from each-other, and edges act as springs which pull the nodes together. They write that this algorithm is intended to be run until the user is staisfied with the layout, or ideally until the graph reaches a state of equilibrium.

[3]WebGL 2.0 Specification — Khronos Group. Last accessed on the 21st of March 2025 https://registry.khronos.org/webgl/specs/latest/2.0/
[4]Ibid
[5]Ibid

- Fruchterman and Reingold (1991) specify a graph layout algorithm that prioritises the even distribution of nodes across the graph. This is an early example of force-directed layouts, but produces a unique look to the graphs created.
- Thenoverlap algorithm, while not force-directed, is a useful layout algorithm for cleaning up the results of previous algorithms that may appear cluttered. While there is no concrete definition of *noverlap*, Gephi's implementation of the algorithm is greedy, and iteratively aims to minimise the number of nodes that are overlapping.



(a) Force-atlas 2 (b) Fruchterman-Reingold (c) Noverlap after Force-atlas 2

Figure 3: The same dataset as used in Figure 2, applied to different force-directed graph layout algorithms

Instead of offering all sorts of metrics such as centrality measures and community detection algorithms out-of-the-box, Gephi allows for a vast collection of different plugins to be installed alongside some basic metrics. These metrics can be applied to every node in the graph with the "statistics" section on the user interface. Once the metric has been evaluated for each node, the user is presented with a "report" of the metric, usually represented as a line graph. This metric is saved as an attribute for each node, and the nodes and edges can then be coloured/sized by it. The modularity of this process is a good example of future-proofing in Gephi, so that cutting-edge measures and layouts can be tested using the software. However, a feature like this is too complex for this project, as the process of selecting a metric and applying it to every node in the graph consists of too many stages, often hidden behind user interface tabs. The concept of a software suite that contains all of these metrics as-is fits well within the scope of this project instead.
Considering that Gephi is a general-purpose tool for graph analysis, this project aims to include more features that are fine-tuned for the analysis of social media networks. One of these is the ability to instantly get information about the accounts of users in the network. Gephi allows the user to access custom-defined attributes for each node if the user goes to edit the node, and the user may generate visualisations of the graph with node labels added if they select the "preview" mode. This project aims to find a middle-ground between pre-processed graph previews and fast graph visualisations by giving the user information about the account they are hovering over with the mouse. This will instantly let the user know everything that has been scraped about the account, such as their username, biography, follower count and so on — assuming that the account is not private or a leaf node. Furthermore, if the user clicks on a node, the tool will hide all accounts that are not connected to the account, allowing them to quickly see what accounts in the network are connected to the node. Gephi does not allow for features like this, which can lead to dense networks being difficult to decipher.
Overall, it is clear why Gephi is a useful and important tool for academics and journalists alike, and it will serve as a good benchmark to compare against when creating this project. It is able to quickly generate valuable visualisations of large networks, and its modular nature makes it very future-proof. As Gephi is a reasonable point of comparison, it is reasonable to hypothesise that many of this projects' features are streamlined versions of processes that are available in Gephi, specialised more towards the analysis of social media.

3 Project Development

3.1 Methodology

3.1.1 Software Development Life-Cycle

Considering that a large portion of this project involves the development of software, it is necessary to discuss the processes and overlaying methodology of its development. This project aims to create two connected tools, where the data visualisation tool relies on the scraping tool to create a detailed dataset and the scraping tool follows the specification provided by the visualisation tool. This relationship between the two main components of the project make it evident that an agile software development methodology will be best-suited for this project.
The Agile Manifesto (Beck et al., 2001) recognizes the need for a flexible approach to software development that creates room for retroactive changes to the specification and other parts of the software. It values individual parts of the software instead of the final package as a whole, and understands the need for working prototypes. The agile methodology's focus on working parts and retroactive changes over final products and rigid specifications is a good fit for this project, as it will be necessary to develop the scraping tool first, and then potentially change it in the future to fit the needs of the visualisation toolkit. Furthermore, it is known that scraping data from social media is a volatile task that may fail at any given point (Trezza, 2023), and may require a wholly different implementation past that point. The agile methodology accepts that these interruptions may occur and provides the developer with time to fix the code.

3.1.2 Data Points to Collect when Scraping

Before commencing with any scrapes on users, it is important to decide which data points should be collected to ensure that all datasets are equal. Notably, there is a separation between private and public accounts on Instagram — both ethically (in terms of scraping) and around the amount of data that is available to be scraped. Under the consideration that private Instagram users have made the choice to hide information about themselves on the platform, the scraping tool should not scrape any advanced details about private accounts, and only the very basics instead.
Moreover, it should be acknowledged that users of the scraping tool may wish to create a dataset with completely anonymous data, in order to be able to use the data in contexts that require a heightened level of privacy. Instead of using the account's username to identify a node in this case, a hashed version of the user's ID will be used instead.

Data Point	Public Account	Private Account	Anonymous dataset
User ID	✓	✓	✓
Private Status	✓	✓	✓
Follower Count	✓	✓	✓
Following Count	✓	✓	✓
List of Following	✓	X	✓
Username	✓	✓	X
Name of User	✓	X	X
Biography	✓	X	X
Post Count	✓	X	X
Attached URL	✓	X	X

Figure 4: Table of data points to be collected from accounts

3.1.3 Deliberation of the dataset file

It is vital that both the data visualisation toolkit and the scraping tool agree on a common file format for the scraped data. In this case, a structure for the datasets should be defined. The overall notation for these files is the JavaScript Object Notation (JSON) format, which is supported by a multitude of different programming languages. The general structure of each file should follow:

```
{
      "anonymous": True/False,
      "fileCreated": Unix time in seconds,
      "fileUpdated": Unix time in seconds,
```

```
        "users": [
          {
              "id": 11223344,
              "Attribute": "Value",
              "Following": [
                  [
                      44332211,
                      user_name,
                  ],
                  ...
              ],
          },
          ...
        ]
  }
```

Of course, it should be noted that the attributes that appear for each user, and user stubs in the "following" arrays will vary depending on the constraints defined in section 3.1. For example, an anonymous dataset will not include any information about the username of accounts.

3.1.4 Choice of API wrapper

The integrity and overall progress of this project has the potential to be marred by the several complications that may arise from scraping large amounts of data from Instagram. Of course, it lies within the interests of Instagram's parent company — Meta Platforms, Inc. — to prevent any unauthorized scraping of data hosted on their platform, and Instagram's terms of use[6] strictly prohibits it. However, as mentioned previously, the EU allows for scraping where the results fall in the public interest (European Parliament & Council of the European Union, 2016). In order to circumvent any potential bans from Instagram, several countermeasures should be prepared:

1. Instagram should only ever be accessed with a disposable "sock-puppet" account, that is comprised entirely of fake identifying information, and is linked to a disposable email address and phone number.

2. Devices scraping data should be connected to a VPN (Virtual Private Network) at all times to anonymise all internet traffic.

3. Scraping APIs being used should be connected to a rotating proxy, to further mask all requests.

4. Reasonable and varied rate limits should be applied to circumvent automation detection algorithms

Creating an API to scrape data from Instagram is out of scope for this project due to the magnitude of operations that would need to be supported, alongside the need for the code base to be flexible based on any changes that Instagram may implement. Instead, we should consider some already available methods for calling data from Instagram:

- Instaloader is an unofficial python library available on GitHub[7] that allows for scraping of most data points available on Instagram, unless they are hidden behind a private account. It is a regularly maintained repository, with the last release being made on January 24$^{\text{th}}$, 2025. The tool uses Instagram's web interface to gather data from public profiles, and requires a user account to log into in order to gather data. It allows for data to be exported to both CSV and JSON, which could be useful in the process of storing account data and can gather the public aspects of private accounts (i.e. the username, bio and profile picture). So, it is able to gather all of the data needed for this project. Instaloader also supports setting custom rate limits and the use of rotating proxies, which satisfy countermeasures 2 and 4.

[6] Instagram Terms of Use — Meta Platforms, Inc. Last accessed on the 31st of March 2025
https://help.instagram.com/581066165581870/?helpref=hc_fnav

[7] Instaloader: GitHub — Last accessed on the 31st of March, 2025.
URL: https://github.com/instaloader/instaloader

- Instagrapi is a newer unofficial python library, also available on GitHub[8]. Although it advertises many of the same features as Instaloader, it uses a reverse-engineered version of Instagram's private mobile API to collect data. The usage of the mobile API could be considered a risk due to its constant change, making the tool unreliable; although it is very regularly maintained on GitHub, with its last commit being made on the 26$^{\text{th}}$ of March, 2025. It allows for the same circumventions of Instagram's automatic detection algorithms as Instaloader. Instagrapi also integrates session management, a feature not used by Instaloader. With this, several user accounts can be listed, and the account scraping data can be swapped if it gets disconnected.

- Selenium is different compared to the previous two tools in that it is "platform and language-neutral"[9], and as such requires the user to manually program the scraping process. This may be considered a downside as it would increase the time to develop the scraping tool, and would also be limited to the Instagram web interface. However, Selenium's language neutrality allows the scraping tool to be developed on a language other than Python, if it becomes necessary. Unlike the previous two API-based libraries, selenium operates by mimicking clicks and keystrokes from the user, allowing for rate limits that may have affected the previous libraries to be bypassed. However, this method also includes some caveats, such as web navigation being potentially slow and changes to Instagram's web interface would require constant maintenance. Furthermore, if presented with a CAPTCHA or other form of automation detection that requires manual authentication, a user will need to interrupt the process.

Of the three, Instagrapi was chosen as the API wrapper to work with. While it is inconclusive whether or not the usage of Instagram's mobile API will hasten the speed of scraping by helping to circumvent rate limits, its session management feature makes it easier to circumvent bans by helping to create a new identity for the scraping tool. Furthermore, the Python programming language is known for being easy to read and write (Ahmed et al., 2021), which are desired features in an agile development environment such as the one for this project. Although there are concerns about Python's speed due to its interpreted nature (Krishna et al., 2025), this should not necessarily be a concern for this project because instead program execution will be time-limited by arbitrary delays inserted to circumvent rate limits.
In the case that Instagrapi is to break or not work outright, Instaloader should be used instead to scrape data. This is because Instaloader accesses Instagram in a similar way to Instagrapi, and so ensuring some degree of code portability. If Instaloader is to also break, then Selenium will need to be used as a final contingency to ensure that some amount of data is scraped from Instagram.

3.1.5 Selection of Target Accounts to Scrape

Instead of creating one large dataset to analyse, this project aims to create several smaller datasets from individual user accounts and comparing them instead. This is becuase the complexity — both visually and computationally — of network graphs increases drastically with size. Each of these datasets could potentially have tens of thousands of nodes, and many more edges, combining these all into one dataset and rendering it would be an incredibly slow task for WebGL to compute.
The following figure is a list of the accounts that will act as the root node for each individual dataset. For the sake of privacy, all personally identifying information has been removed and instead a list of themes that the account presents will be given. All accounts promote some degree of far-right rhetoric, and actively push bigoted themes such as antisemitism, homophobia and anti-feminism. Some of these accounts were chosen due to their prominence in debate surrounding the far-right, others were chosen because they masquerade themselves as motivational accounts, and some are simply meme accounts run by members of the far-right.

[8]Instagrapi: GitHub — Last accessed on the 31st of March, 2025.
URL: https://github.com/subzeroid/instagrapi
[9]Selenium: GitHub — Last accessed on the 31st of March, 2025.
URL: https://github.com/seleniumhq/selenium

Account	Themes	Follower Count	Following Count
A	Comedy, Homophobia, Transphobia, American Conservatism	$\sim 586,000$	180
B	Comedy, Ethno-nationalism, Homphobia, Anti-Semitism	$\sim 12,000$	69
C	Motivation, Men's Rights Activism, Anti-feminism, Homophobia	$\sim 49,000$	10
D	Canadian Nationalism, Xenophobia, Anti-feminism, Homophobia	$\sim 18,000$	57
E	European Ethno-nationalism, Anti-feminism, Homophobia	$\sim 189,000$	942

Figure 5: Table of accounts to act as root nodes in datasets.

Overall, if we were to scrape the following of each of these 5 accounts, and then the following of each of those accounts' followings (or in layman's terms, a depth of one), we would need to scrape roughly $1,263$ accounts. Although, this is only a rough estimate as some of these accounts may be private, and so will not need to be fully scraped. While this is not a massive scraping operation, accounts such as A and E are following many accounts, which may make it easier for Instagram to detect automated behaviour. For this reason, account E should be scraped last due to the size of the resulting dataset, and the issues that may be encountered in scraping it.

3.2 Scraping Tool Development

3.2.1 Command line arguments

Considering that the scraping tool is operated through the command line, it is ideal that several different command line argument options are available to the user. For this, the "argparse"[10] python library was used due to its simple implementation and wide range of types of arguments that could be implemented. The program's command line arguments are:

Argument	Is Required?	Description
file	Yes	The .json file to write scraped data to.
-h or –help	No	Displays information about command line arguments.
-a pr –anon	No	Sets the scrape to anonymous data collection. Cannot add anonymous data to non-anonymous dataset.
-v or –verbose	No	Displays useful debugging information in the program.
-d x or –depth x	No	Sets the desired depth of the scrape to x, it is 2 if not specified.
-t x or –target x	Exclusively -t or -i must be given.	The username of the root account to scrape.
-i x or –id x	Exclusively -t or -i must be given.	The UUID of the root account to scrape.

Figure 6: Table of command-line arguments for the scraping tool.

Translating this process into python was an effortless task due to argparse's array of different methods to parse parameters. `-h`, `-a` and `-v` can all be handled as on/off flags using the `store_true` action parameter, meaning that if they are not parsed, they are automatically false. The required `file` argument could be handled as a positional argument, which is handled at the end of the argument array and is required by default. Finally, the `-d` argument can be assigned a default value through the `default` parameter if it is not specified.
Upon implementing this, there were no major errors with parsing or running command line arguments. The args object contained all of the user's inputs, and flag styled arguments such as `--anon` were false if not specified.

[10] Argparse Library — Python 3. Last accessed on the 4th of April 2025
https://docs.python.org/3/library/argparse.html

3.2.2 File Creation and Validation

The next part of this program that is structurally vital to its execution is the creation of functions that handle the creation, reading and writing of the .json formatted dataset files. For this, a file is defined, `file_manager.py`, which handles all reading and writing of data. The first function in this file, `validatePath()` is called upon first execution of the program. It checks the following:

- That the file exists.
- That the file is a .json file.
- That Python has write access to the file.
- If the file exists, that the file's `--anon` flag matches with the one supplied in the program arguments.

If the file exists, but cannot be validated, the user is prompted to overwrite the file. If they choose to do so, the standard file creation procedure is called, which is also called if the file specified does not exist. If a valid existing file is loaded, the timestamp flag on the data-set is overwritten and all existing users are appended to the global array `reachedUsers`, which keeps track of all non-leaf node users in the dataset to avoid self-loops.
Once the file has been verified as a valid document, the timestamp has been updated and any already existing users have been recorded, the execution of the program may continue. However, the `file_manager.py` file also includes functions to save new data to the dataset, via the `addUser(user)` function. This function is intended to be called every time a user's data and following is scraped, instead of when the scrape completes its execution, due to the fact that scrapes may often be halted by Instagram blocking access to their servers. The function checks that the user does not currently exist in the dataset, and then opens the file and writes the user data to it. Overall, this combination of functions should outline all ways in which the program will need to read and write from files. The concept of updating the file sequentially instead of only writing once the scrape was completed was a retroactive change after noticing that large amounts of data could be lost if Instagram was to stop a scrape of a popular account midway through its execution. Beyond this, the functions make good use of assertions and exceptions to ensure that the contents of loaded files are valid.

3.2.3 Handling Instagram API calls with Instagrapi

Before starting the scraping process, it is important to ensure that the user is logged in to Instagram and possesses a valid token. All functions that handle API calls to Instagram through Instagrapi are defined in the file `instagrapi_wrapper.py`, which serves as a level of abstraction from direct API calls. This project defines a function, `validate_session()`, which handles this process. To set-up this process, a random and variable delay for each HTTPS request to the interval of 1 to 2 seconds, using `cl.delay_range[1,2]`. This is faster than the interval that will be used for the scraping process, but Instagram should not consider the process of signing in to be suspicious behaviour, and so should not block these requests. To continue this process, a HTTPS proxy link is then provided to the Instagrapi `Client` object, which will route all requests through a rotating residential proxy.
Next, the program checks for a file named "session.json" in the user's current working directory — if this file exists, the user is asked whether they wish to load it. This is an implementation of the aforementioned session management system offered by Instagrapi, which allows the program to seamlessly log back in to a session without needing to re-enter details. If there is no session token, or the user chooses not to load it, the `manual_login()` procedure is called, which prompts the user for a username and a password (note that the password is omitted on the user's prompt with the `getpass` function from the getpass library). If the user's details are correct and the user is logged in, the session token is dumped into session.json for future usage. If this process fails, then the user will be asked if they wish to try again, or exit the program. Once this process has completed, `cl.get_timeline_feed()` is called with the purpose of testing that the client can now make requests to Instagram. If this fails, the user is prompted to manually log in.

```python
def validate_session():
    cl = glob.client
    cl.delay_range = [1,2]
    cl.set_proxy( --- OMITTED ---)

    if input("Would you like to try a new proxy?\t[Y/N] = ") in ["Y", "y"]:
        cl.set_proxy(input("Please enter a new proxy URL! (Please use HTTPS)"))

    if os.path.exists("session.json") and input("User client token found! Would you like to load it?\t[Y/N] = ") in ["Y", "y"]:
        settings = cl.load_settings("session.json")
        cl.set_settings(settings)
        cl.login("null", "null") # Details not needed if logging in with session token

        try:
            cl.get_timeline_feed()
        except LoginRequired:
            print("Session token expired! Starting manual login...")
            old_session = cl.get_settings()
            cl.set_settings({})
            cl.set_uuids(old_session["uuids"])
            manual_login()

    else:
        manual_login()

    cl.delay_range = [1,5]
```

Figure 7: Implementation of session validation using Instagrapi.

Several features were added to these functions retroactively. Of these, adding the ability to set a proxy had the greatest effect on the execution of the code. This is because originally the code was run on a VPN without a secondary proxy, the IP addresses of which Instagram had likely tracked due to their common usage. Without the usage of a residential proxy, it was impossible to make a string of requests to Instagram without being blocked from their services. The decision to make the usage of a proxy required stems from this issue: If the user runs this program on their own network without a VPN, they could risk their IP address getting banned from accessing Instagram. Furthermore, the process of restarting the program every time the user was logged out by Instagram (in an attempt to block automated behaviour) or was unable to login became cumbersome during the development process, so both functions were made recursive. The procedure cannot succeed unless the Instagrapi client is validated, otherwise it will recur until it is validated or the user exits the program. Under this assumption, the user was also given the ability to manually specify a new proxy URL, in the case that the previous proxy was blocked.
Also in this file exists functions that allow the program to easily get a user by username/UUID, and get the following for a user. The implementation of the `get_user_by_name()` function was an elementary task. It takes one parameter, a string which is the user's username, and returns an Instagrapi User object[11] — which contains important details about the user such as their following count and biography. The function improves on the mistakes made whilst making the `validate_session()` function, and allows the user to revalidate the session in the case that an error occurs in the process of getting this data (such as being logged out).

[11] User Object — Instagrapi docs. Last accessed on the 7th of April 2025
https://subzeroid.github.io/instagrapi/usage-guide/user.html

```python
def get_user_by_name(name):
    try:
        user = glob.client.user_info_by_username_v1(name)
        return user
    except Exception as e:
        glob.log(e)
        if input(f"Failed to fetch user {name}! This is likely because instagram has logged you out.\nWould you like to revalidate the session?\t[Y/N] = ") in ["Y", "y"]:
            validate_session()
            print("Session revalidated!")
            return get_user_by_name(name)
        else:
            raise SystemExit
```

Figure 8: An implementation of the `get_user_by_name()` wrapper function.

The `get_following_for_user()` function details a very similar process, accepting the UUID of the user and the number of the users that they are following. Like with `get_user_by_name()`, the function will revalidate the session and recur if an error occurs.
However, one fatal flaw was discovered during the scraping process that required retroactive action: Instagram was able to detect automated behaviour when scraping accounts that are following a large amount of users. This is likely due to repetitive requests being sent, but there is no concrete number of following that Instagram blocks this for. There were cases where the tool was able to scrape accounts with over 4,000 following, and cases where Instagram blocked scrapes for users following anywhere from 1,500 to 3,000 accounts. This has troublesome implications for the coverage of the data scraped, which will be covered in section 4. A rudimentary solution to this problem was implemented instead: accounts following more than 1500 users will not be scraped.

```python
# Get all of a user's following, provided the user's id.
# If the amount = 0, all followers will be retrieved - otherwise the amount parameter will act as a limit
def get_following_for_user(id, amount):
    try:
        if int(amount) > 1500:
            glob.log(f"User {id} is following more than 1500 accounts! scraping this will get us banned, skipping...")
            return []
        return glob.client.user_following_v1(id, 0)
    except Exception as e:
        glob.log(e)
        if input(f"Failed to fetch following of {id} with amount {amount}! This is likely because instagram has logged you out.\nWould you like to revalidate the session?\t[Y/N] = ") in ["Y", "y"]:
            validate_session()
            print("Session revalidated!")

            return get_following_for_user(id,amount)
        else:
            raise SystemExit
```

Figure 9: An implementation of the `get_following_for_user()` wrapper function.

The implementation of these functions provide a sturdy foundation of which to build the rest of the code base. Despite some retroactive action being taken, the development of these wrapper functions was not plagued by many errors — instead, the process of finding the correct series of accommodations for Instagrapi's functions found to be the most time-consuming task of this

segment, with variables such as the variable delay time needing to constantly be tweaked to avoid rate-limits.

3.2.4 Scraping Loop and Data Filtering

Considering that the `user` objects returned by the previously defined Instagrapi wrapper functions include data that was not specified to be collected in section 3.1.2, functions will need to be created that filter this data into a use-able format. The file `user_manager.py` defines functions that handle the main loop of scraping this data in a breadth-first manner, and also functions to filter this data. One of these functions is `user_as_dict(user, following)`, which takes a User object and a list of UserShort objects (obtained from `get_following_for_user()`) and returns the formatted data as a list. A companion function, `anon_user_as_dict(user, following)` exists for anonymous datasets that have different requirements, but will not be shown for the sake of conciseness, as the functions are very similar.
The final significant part of this scraping tool is the loop that combines all of the previously defined methods and functions. It is, at heart, a glorified `for` loop, which handles the breadth-first search process by getting the details and following of the initial account and then creating a queue of users to scrape by appending the following of each account scraped. It checks that users have not already been scraped with the `userExists()` function previously defined in `file_manager.py` file, preventing any potential self-loops. Once a user has been fully scraped, the `addUser()` function is called to add it to the dataset.

```python
def start_search(id, depth, isId):

    # Get Initial user for scrape
    iUser = wrapper.get_user_by_id(id) if isId else wrapper.get_user_by_name(id)
    following = fetch_following(iUser)
    userD = anon_user_as_dict(iUser, following) if glob.anonMode else user_as_dict(iUser, following)
    fm.addUser(userD)

    userQueue = following
    newUserQueue = []

    # Start loop until desired depth is reached
    for d in range(int(depth)):
        for user in userQueue:

            # Don't scrape if the user has already been scraped
            if not fm.userExists(user.pk):
                glob.log(f"Getting full user profile for {user.username} at depth {d}")

                # Get user details and following
                fullUser = wrapper.get_user_by_id(user.pk)
                fullUserFollowing = fetch_following(fullUser)
                newUserQueue += fullUserFollowing

                glob.log(f"Adding user {fullUser.pk} to DB...")

                # Convert user to dict and add to DB
                fm.addUser(anon_user_as_dict(fullUser, fullUserFollowing) if glob.anonMode else user_as_dict(fullUser, fullUserFollowing))

            else:
                glob.log(f"User {user.pk} already reached! skipping... ")

        # Swap out user Queue for new depth layer
        userQueue = newUserQueue
        newUserQueue = []
```

Figure 10: Implementation of the main scraping loop

3.3 Data Visualisation Software Development

As with any large-scale web application, a reliable foundation is needed that can easily provide support for user authentication, database management and dynamic user interfaces. For this project, Laravel[12] with Livewire[13] has been chosen as a web framework. Laravel integrates a fine-grained database management system that shines in its efficiency and performance when compared to other PHP-based web frameworks such as Yii (Latif & Kusumasari, 2018). Considering that this project involves users handling large amounts of potentially sensitive information, a good standard of security is required on the web application. Laravel can provide this with the *Breeze* starter kit, which includes a default table of users, and routes to handle authentication, logging in and out. Finally, Livewire is an extension of Laravel that offers easy implementations of dynamic user interfaces — this is useful for this project as it allows for actions such as loading new graphs or

[12]Home Page — Laravel Last accessed on the 9th of April 2025 https://laravel.com/
[13]Home Page — Laravel Livewire Last accessed on the 9th of April 2025 https://laravel-livewire.com/

adding new datasets to make an immediate effect on the page, instead of requiring a refresh.

3.3.1 Database Management

Considering that this project extends from Laravel Breeze, which includes a table for users, the only table that is needed to be added to the web application is the table `Datasets`, which handles storing information on each dataset in the application, such as the user count or private account count of each dataset. datasets uploaded to the website have the option of being global; if they are, then all users on the web application may view the data, otherwise only the user who uploaded the data may view it. For this reason, each dataset must reference the id of the user that uploaded it, and includes an attribute `is-global` that references whether or not the data is global. It is intended that when a dataset is uploaded, a swift parse is done of the data to take some metrics to display to the user. The metrics are the number of users in the dataset, the number of private accounts, and whether or not the dataset is anonymous.

```
public function up(): void
    {
        Schema::create('datasets', function (Blueprint $table) {
            $table->id();

            $table->unsignedBigInteger('user-id');
            $table->foreign('user-id')
                  ->references('id')
                  ->on('users')
                  ->onDelete('cascade');

            $table->bigInteger('user-count');
            $table->bigInteger('private-count');
            $table->boolean('is-anon');
            $table->boolean('is-global');

            $table->string('file-name')->unique();
            $table->string('mime-type');
            $table->string('path');
            $table->string('disk')->default('local');
            $table->unsignedBigInteger('size');

            $table->timestamps();

        });
    }
```

Figure 11: The `up()` function for the `Datasets` table.

In order to allow the user to upload the previously created datasets to the web application, some further attributes are required in the `Datasets` table. The name of the file (which should be a unique value) is stored, along with the path of the file and which disk the file should be uploaded to. This allows the file to be easily retrieved when loading it for visualisation. Next, the user needs an interface to upload the datasets to. This can be achieved through a Livewire component that executes a validation function and then uploads the data if it is valid; or displaying an error message if it is invalid. The `submit` function handles the user pressing the "submit" button on the upload form, `json_decode` is called to ensure that the JSON data is valid, then `validateJsonStructure` parses the data and checks that the structure is the same as what was defined in section 3.1.3. If either of these processes fail, the appropriate error message will be displayed to the user in real-time. Otherwise, the file will be uploaded and will be available for the user to visualise.
In terms of design, the datasets available to the user should be displayed on a side-bar, with the information recorded about the data alongside them. There is a clear divide between sets that are locally and globally available. In Laravel, this is done simply by defining components for both the

list of datasets and the individual components, and then creating the components in this list with a `@forelse` loop, that displays placeholder text if none exist.

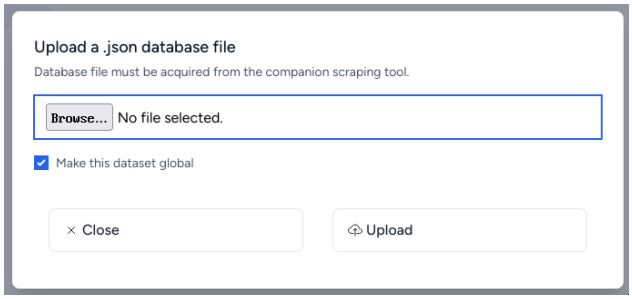


Figure 12: The user upload form.

The process of creating the user upload form and verifying the submitted data was an unexpectedly complex part of this project. While it may seem simple, Laravel's database system involves many working parts that must all work for the program to work. This aspect of the project took longer than expected due to a multitude of errors with Laravel's complex error handling system and the long implementation of the `validateJsonStructure` function, which is necessary for ensuring the file follows the required structure.

3.3.2 Implementation of Sigma.js

Sigma.js[14] is a JavaScript library focusing on large-scale graph rendering using WebGL. It allows nodes to easily be moved around and animated, and it offers built-in support for community detection and centrality measures through the Graphology library. This project uses the Bun package manager for handling installation of packages, and installing sigma.js was as simple as running `bun install sigmajs`.

A HTML `<div>` element is declared over the rest of the page, with the id tag "container". When the user clicks on a dataset to load, sigma.js destroys all other instances of itself and attaches to the container element. Then, it creates a new Graph object, and loads all of the nodes and edges onto the graph, using the included `addNode()` and `addEdge()` functions. However, there is some discussion to be had about whether this was the most efficient way to load graphs, as the sigma.js library also includes the ability to load `.gexf` files — including this might reduce the amount of code needed to import data. This is exacerbated by the fact that every node in the graph must be iterated over twice in this loading process, once to add all nodes to the graph, then again to add all edges. This is done to ensure that data duplication does not occur, as in previous versions of the graph generation algorithm duplicate versions of the same node appeared when adding the node's neighbors at the same time as the node itself. On top of creating an inaccurate representation of the data, this also led to issues with calculating centrality measures and force-directed layouts. So instead, this less-efficient but safer method was chosen.

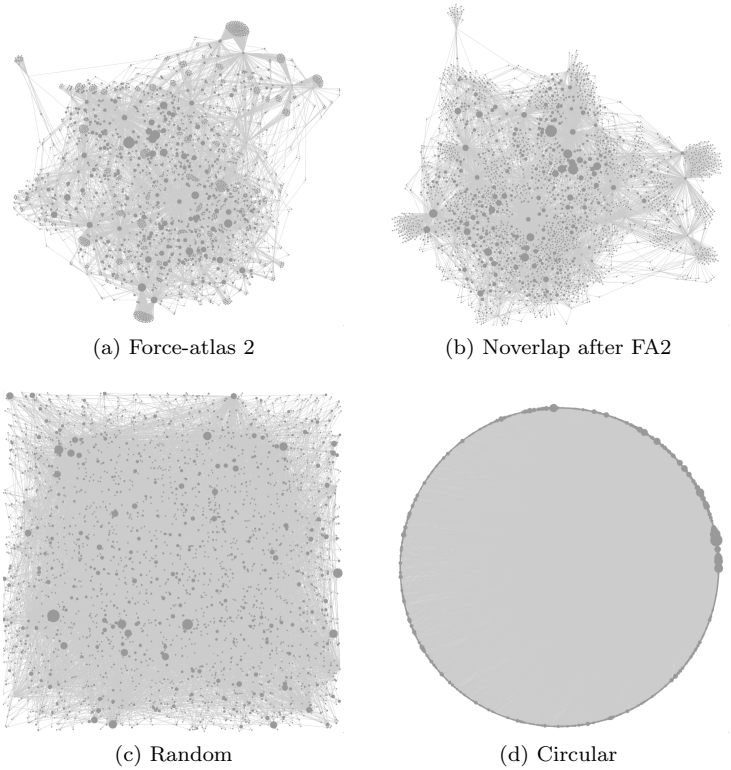
Sigma.js also includes multiple different graph layout algorithms available through the Graphology library. Of the layouts mentioned in section 2.3, the library implements Force-Atlas 2, Noverlap and random layouts; it also includes the circular graph layout, which formats the nodes in the graph in a circle, with edges inside the circle. There is also the circle-pack layout, which arranges the nodes in a bubble graph with two hierarchical parameters; the first determines which bubbles will be shown and the second orders the arrangement of the bubbles. While the random, circular and circle-pack layouts are a simple assignment of each node to a given position, Force-atlas 2 and Noverlap are both greedy algorithms; they are iterative algorithms that only consider their immediate state. This means that these functions will be harder to implement, as Graphology instead exposes workers that can run these algorithms while the graph renders. To implement this, a class `layoutWorker` has been defined in a separate file, which will act as a wrapper class for the workers. This class stores the instance of the layout and any HTML elements that will interact with the worker (i.e. buttons), it also defines `start`, `stop` and `toggle` functions that manage the state of the worker. If one worker is already running, and another worker is started or another layout is selected, a utility function will be run to cancel all current workers and animations before starting the new layout. This assists the programming process in that it is both modular and encapsulated. If any more layout workers need to be added to the program, it is as easy as importing the new worker, creating a new class and adding the new button to run the worker. Furthermore, the majority of the code that handles the worker is either in the `layoutWorker` class or in the file that holds it.

Of the layouts available, Force-atlas 2, Noverlap, Random and Circular were added at this point.

[14] Home Page — Sigma.js Last accessed on the 9th of April 2025
https://www.sigmajs.org/

CirclePack could not be properly implemented at this point due to the fact that it depends on centrality measures and community detection to be implemented to work well. It will be covered in 3.3.3 and 3.3.4 respectively. Furthermore, the decision was made to drop the implementation of the Fruchterman-Reingold layout. This is because it would require the creation of a new worker from scratch for the Graphology library. This is out of scope for two reasons: The time-frame required to write a worker for this algorithm usurps much of the development time for this project, and there are concerns about the efficiency of the layout algorithm (Jeowicz et al., 2013) that imply that the algorithm would be too slow to run on large graphs in the first place (yet alone on WebGL, which as previously mentioned can be inefficient in some areas).

Figure 13: Image of the selector bar for different layouts in the web application.



(a) Force-atlas 2

(b) Noverlap after FA2

(c) Random

(d) Circular

Figure 14: Chart of different layouts being run on dataset D.

3.3.3 Centrality Measures

As mentioned before, Graphology has built-in support for a number of different centrality measures. All centrality measures discussed in section 2.1 (in/out/degree centrality, betweenness centrality and closeness centrality) are included, along with centrality measures such as eigenvector centrality and pagerank. These two measures will also be included in the application, due to how Graphology implements all centrality measures in the same way, and their utility in understanding the general structure of the network. Eigenvector centrality is a measure of the cumulative influence a node has over a network, nodes with many in-neighbors with a high eigenvector centrality will also have a high eigenvector centrality (Fletcher & Wennekers, 2018). Pagerank is an algorithm best known for its use by Google Search; it uses an algorithm not dissimilar to eigenvector centrality to rank nodes by their popularity (Page, 1998). Pagerank in particular will be useful for this project in helping to determine which accounts in the network are the most popular, and when paired with community analysis it could aid in identifying who the most popular users are in different cliques. On the user interface, the user will be able to select different centrality measures via a panel of buttons. When clicked, these buttons will run JavaScript functions to calculate the desired measure for each node in the graph — these measures will then be normalised between a minimum

and maximum value, and then applied to the size of each node. Using node size to demonstrate centrality is an intuitive method in which to convey the importance of a node to a user (Yu et al., 2022). When a new network is loaded, the default measure should be in-degree centrality; this is because it is a simple and efficient method to get a basic understanding of the structure of the graph. In-Degree centrality begs the question *"Which account is being followed by the most people in this graph?"* in the context of this project, and as such is a valuable metric to greet the user upon loading the graph.
Furthermore, when the user selects a new centrality measure, a table in the bottom-right corner of view pane will be updated to display the top 20 users by the measure selected. This is a effective method for instantly providing the user with important metrics, and is essential in graphs that are large and appear tangled. However, this feature was not easy to implement. The current solution to this involves sorting the list of every user in the graph by a centrality measure when that measure is selected, then taking the top 20 values of that list. Sorting a list of possibly tens of thousands of users may quickly become a cumbersome task for the browser, and as such there are concerns about the efficiency of this method. In order to waive these concerns, the centrality measure for each node will be cached as a node attribute when processed, reducing the necessary number of repeat operations. The table of users is stored as a Livewire component, that updates whenever a new measure is picked. Once the list is sorted, an event is broadcast from Livewire's JavaScript library to the component that contains all of the necessary updated values.

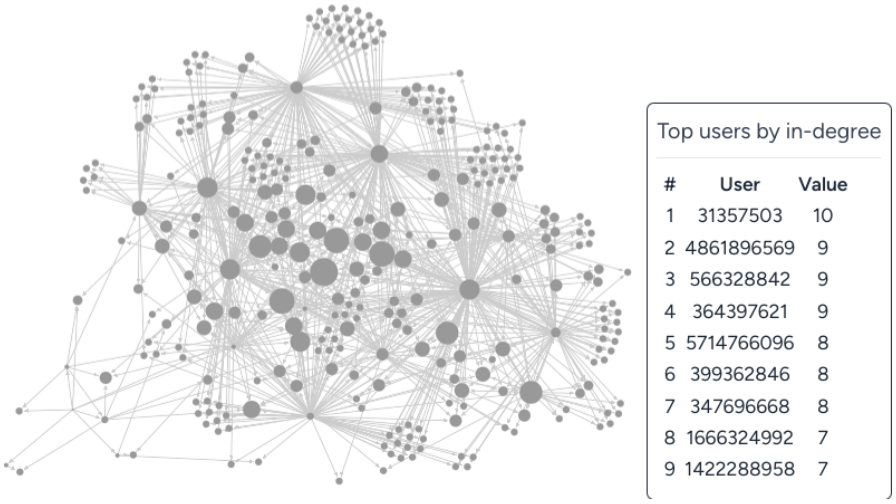
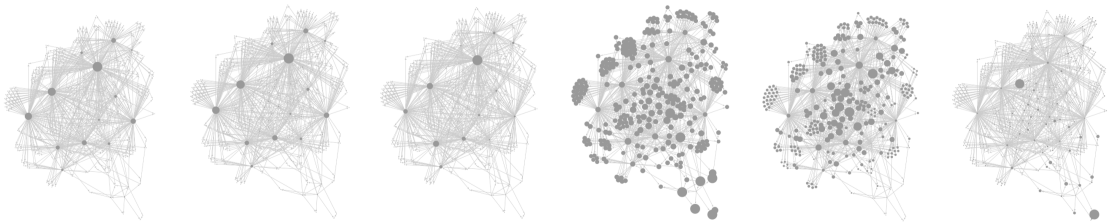


Figure 15: dataset C, laid out with Force-atlas 2 and with nodes sized by In-Degree, with the table of top 20 centrality values.



(a) Out-Degree (b) Degree (c) Betweenness (d) Closeness (e) Eigenvector (f) Pagerank

Figure 16: Chart of different centrality measures being run on dataset C.

3.3.4 Community Detection

The Louvain community detection method is also available in the Graphology library. In this project, community detection should be automatically computed when a graph is loaded, and then the colour of each node should be determined by its community. Assigning each community a vibrant colour should indicate to the user a clear separation between these groups (O'Connor, 2015). In this case, the IwantHue library will be used to generate a distinct colour palette for the graph, as recommended by the sigma.js documentation. Originally, edges were assigned the same

colour as their source node, though this led to nodes becoming hard to distinguish from edges in more cluttered graphs. To fix this, edges are made a 50% lighter colour, making the nodes in the graph easy to distinguish from edges while still being able to determine the source node of edges.

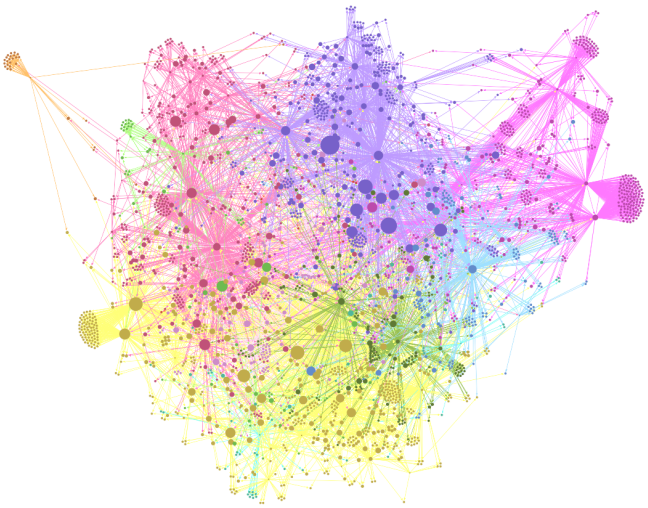
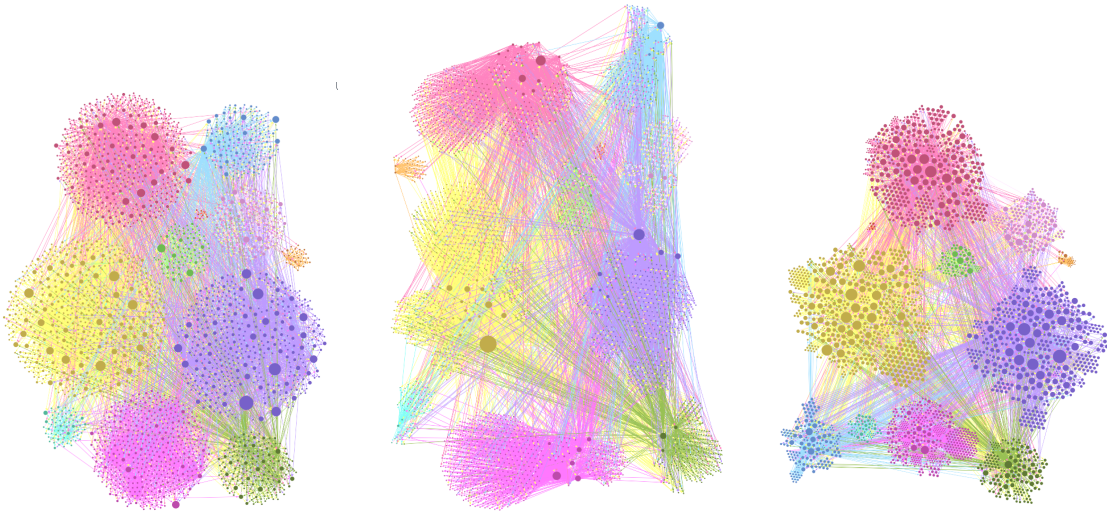


Figure 17: dataset D, laid out with force-atlas 2, nodes sized by in-degree centrality, coloured with the Louvain method.

With community detection and centrality measures implemented, it is now possible to make meaningful representations with the circle-pack layout. This is because the layout requires (at least) two parameters, each specifying what node parameters to group the nodes by. In the case of these network graphs, the first parameter should always be the community of the node, to easily be able to determine the community. The second parameter should be the centrality measure that the user has currently selected, which will group the nodes in each circle by their centrality.



(a) In-Degree Centrality (b) Degree Centrality (c) Eigenvector Centrality

Figure 18: Chart of different centrality measures used as a second parameter for the circle-pack layout on dataset D.

3.3.5 Searching, Filtering & Highlighting

Considering that this application could be needed to render graphs of potentially millions of nodes, some considerations should be taken to aid the user in analysing the graph. Currently, all "leaf" nodes (nodes with an in-degree of 1 and out-degree of 0) are culled from the graph by default. This is because this is the modal node, and it does not reveal much about the underlying structure of the data whilst also taking a large toll on rendering. Naturally, this should be expanded to allowing the user to filter all nodes by a minimum in-degree and out-degree, allowing them to disable this

default limit if they so wish. Furthermore, the user should be able to filter out all nodes that were not fully scraped, allowing them to view the core users in the graph.
Moreover, the user should be able to manipulate the nodes visible to them in the graph visually. This can be done by implementing a search bar, which will only show users with the specified query in their name. Also, clicking on a node should hide all nodes that are not neighbouring it. This should allow the user to quickly identify the local community of a node. In a similar light, hovering the mouse over a node should give the user an insight into that account, displaying the textual information scraped about it.
While this seems like a large suite of features to implement at once, they are mostly all handled by the same programming patterns: events and reducers. In the file which handles defining the sigma.js instance, several events are defined when a new instance is loaded. These events handle when a node is clicked on and when a node is hovered over. When a node is clicked on, it toggles the state of the clickedNode variable, which will be used later by the node & edge reducers. When a node is hovered, it makes the display box visible and dispatches a livewire event to set its contents.

```
sigmaInstance.on("clickNode", ({node}) => {
    console.log(returnGraph.getNodeAttributes(node));
    clickedNode = (clickedNode == node) ? null : node;
});

sigmaInstance.on("enterNode", ({ node }) => {
    Livewire.dispatch('update-userbox', returnGraph.getNodeAttributes(node));
    dispBox.style.display = "absolute";
});

sigmaInstance.on("leaveNode", () => {
    if (clickedNode == null){
        dispBox.style.display = "none";
        setHoveredNode(null);
    }
});
```

Figure 19: Definition of event handlers for sigma.js.

Upon hovering over the filter box, the user is presented with an array of options. The user is able to set a minimum value for in-degree and out-degree, and they are able request that only fully scraped accounts are shown. Upon clicking "reload", the graph is fully reloaded, but the filters stored in these HTML elements are applied. Filtering by degree is a simple task, however showing only nodes that were fully scraped was difficult to implement. This is because the method for generating the graph had to be modified to consider this parameter. Initially, an auxillary function was defined to handle this that added edges at the same time as nodes. However, this would result in several connected components being generated due to the how the neighbors of each node were not generated when the edges were generated. This was fixed by merging both functions, doing a first parse of the nodes to add every node to the graph, then skipping creating additional nodes for the node's neighbors. The second parse then generates all edges in the graph, but only if the target node exists.

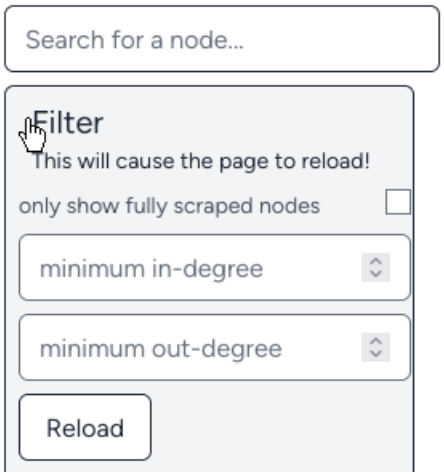


Figure 20: The search bar and the filter box, upon being hovered over.

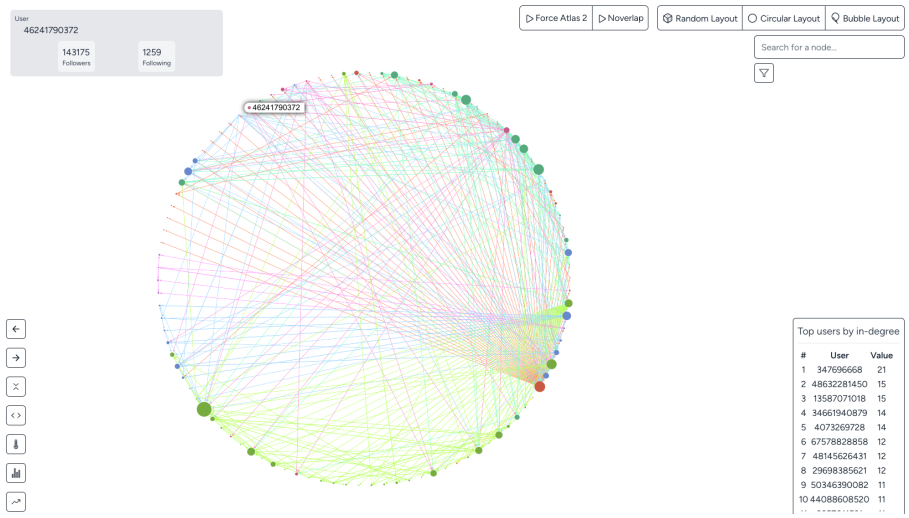


Figure 21: Dataset D, filtered to only show fully scraped nodes, with a node hovered.

Next, reducers must be defined for both nodes and edges. Node and edge reducers control the visibility of each node and edge, upon different conditions being met. For nodes, a node is hidden if another node is clicked on and that node is not in the array of the clicked node's neighbours. A node is also hidden if a search string is present in the search box, and the node's name does not contain the search string. Edges are always hidden if they do not connect two visible nodes. This is a simple implementation, but the concept that this method is called for every node and every edge, every frame, raises some concerns about the overall performance of the program.

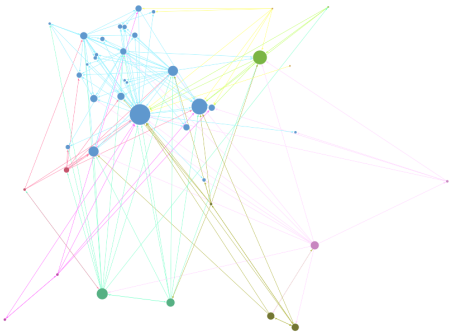


Figure 22: Dataset D, after clicking on the node with greatest in-degree.

4 Evaluation & Discussion

4.1 Evaluation of Scraping Tool

In section 1.2, the Objectives for this section of the project were defined as:

(a) The tool is able to successfully evade Instagram's scraping detection algorithms.

(b) The tool collects different metrics from Instagram accounts, such as Username, Following, Bio, and whether or not the account is private.

(c) The tool is able to gather anonymous data if required to, where all personally identifying information is obfuscated or not collected.

(d) The tool respects the privacy of accounts marked as private, and will not scrape any data further than the private account's username.

Of these criteria, it can be said with confidence that objectives (c) and (d) were easily met. The `--anon` flag was added early in development, and data collected with this flag enabled are not in

any way personally identifiable. As mentioned in 3.2.4, companion methods exist for anonymising data, and the scraping tool is fully compatible with anonymous data. It is also impossible to add identifiable data to the anonymous dataset, to avoid mixing information. In a similar light, if an account is marked as private then the only data points that will be scraped from the account are the username, private status and follower/following counts.
There was, however, a large error that makes it hard to justify stating that criterion (b) has been met. A bug existed in the `user_manager.py` file that set the `is_private` flag on all accounts scraped to `True`, regardless of whether the account was private or not. While the attributes collected for each class of user remained the same, this had the potential to create major issues with the loading of datasets in the data visualisation tool. Fortunately, a work-around was discovered that was also able to identify if an account was private or not: on private accounts, the `following` array did not exist. Unfortunately, all scraping of account data was conducted while this bug was present, so the only way to work around the error was to work with it. In the data visualisation toolkit, account access status is instead determined by whether or not the user object contains an array of following.
In section 3.1.4, it was correctly predicted that the process of scraping account data from Instagram had the potential to halt the development process due to the array of countermeasures employed by Instagram, linking to criteria a. Even with the use of a residential proxy, an array of different sock-puppet accounts to use and variable rate limits, the scraping process was limited to roughly 200 accounts per day. Past this point, Instagram would refuse to return any user data and instead invalidate any current session tokens for the account, effectively rendering the account useless. Furthermore, when these accounts started to scrape user data, Instagram would repeatedly request face and phone verification for the accounts, although this was circumvented by using an anonymous phone number and a fake image of a person. It remains unclear how Instagram was able to identify the link between the sock-puppet accounts although this could be due to the social media keeping track of used IP addresses, phone numbers, or even the activities of different accounts suspected of scraping. Moreover, it was mentioned in section 3.2.3 that the following for accounts that follow over $1,500$ users was unable to be scraped. This leaves a concerning implications on the coverage of the dataset, as some accounts that were scraped were following up to $9,000$ users, which are unable to be added to the dataset. These coverage issues could mean that different key measures such as centrality measures and community detection are not accurate, hindering the process of analysis. There is no clear work-around for this using the current methods, and instead perhaps users of the data visualisation tool should be notified which accounts were unable to be scraped fully with a visual spoiler such as an exclamation point.

4.2 Evaluation of Data Visualisation Tool

In section 1.2, the Objectives for this section of the project were defined as:

(a) The tool generates interactive network visualisations from the data scraped.

(b) The tool allows users to upload their own scraped data, for themselves or others to see.

(c) The tool has some security measures that prevent users outside the organisation to view the scraped data.

(d) Different centrality measures are used to see and rank the relative significance of users in the network.

(e) Community detection is used to visualise a separation between groups of users.

(f) The user can rearrange the graph into different layouts.

Criteria B and C were handled without any issue in section 3.3.1, where Laravel Breeze was used to create a default login page and user table in the database. The process of uploading data was, while not plagued with errors, a time-consuming process that involved writing functions to filter and validate uploaded JSON datasets. The use of Livewire in this process should be highlighted, however, due to how it helped to decrease development time. The alternative to Livewire would have been JavaScript AJAX statements, which offer a similar functionality but without the integration to Laravel. The integration with the Laravel ecosystem hastened the process of adding dynamic components.
Sigma.js and Graphology allowed for the relatively simple implementation of the network graphs

into the program. However, it should be noted that the efficency of the graph loading process is not optimal. Instead, a network graph layout file format should have been used, such as the *.gexf* file format, which is able to be fully imported by Graphology. This file format is able to store all of the data needed by the application, such as usernames and biographies, in the attributes for each node. Moreover, the overall performance of the tool is notably poor. It can struggle with an inconsistent frame-rate for datasets such as B and C, which do not require that over $1,000$ nodes are rendered by default. This issue becomes especially prevalent when running any form of force-directed layout on the network as instead of a fixed layout being rendered, physics calculations are constantly being run on the graph. As mentioned in section 2.3, this is largely the fault of sigma.js's use of the WebGL 2.0 rendering pipeline, which is a heaily simplified version of OpenGL, based on its library for embedded systems[15]. This version is missing several key features that could assist this application's rendering process, such as compute, tessellation and geometry shaders. Because of this, the application's performance pails in comparison to that of Gephi, which is able to make full use of the GPU. Unfortunately at this time there is no simple solution to improving performance, apart from attempting to streamline the loading process. Instead, we should look towards graph rendering libraries that utilize emerging graphics pipelines such as WebGPU, which is able to make use of compute shaders and parallel processing[16].

The web application offers a wide range of centrality measures to the user, represented by the size of each node. This feature is simple and easy to use, with the centrality measures available being represented in the bottom left corner of the viewing pane, and the name of each centrality measure appearing when hovered over. Not only does this satisfy objective d, it also offers a more user-friendly experience than a tool made for data analysts such as Gephi, which requires that the entire network diameter is calculated to view some centrality measures and the user must also manually apply that statistic to the visual representations of the nodes. This is one of the many ways in which this project specialises the very general field of graph visualisation towards social network analysis.

Another way in which this tool is more specialised is how it allows the user to instantly get metrics on a target account. This could be by hovering over a node to get the account's details, or clicking on it to view all of its neighbors. Not only are these methods fast when considering the program's overall slow execution, but they offer a streamlined process that more general programs like Gephi cannot produce. This level of interactivity satisfies the criterion for a; however the performance of the program does hinder this slightly.

The application also successfully implements community detection through the Louvain method. It should be considered that Gephi requires the user to install a plugin for many forms of community detection, which is noteworthy because community detection is one of the most valuable metrics in network analysis. If a centrality measure is calculated in Gephi, then the user must also generate a colour palette and and then assign that palette to the community. Similarly to centrality measures, this process is completely streamlined in this application. Clusters are calculated automatically upon the graph loading, and then the colour palette for the clusters is assigned with help from the IwantHue JavaScript library, providing the user with instant metrics on communities in the graph and satisfying objective e.

4.3 Evaluation of Both Components when Used Together

Considering that this project is a two-part software package, it is vital to discuss the use of the entire package together. For this, we should look back to the example datasets specified in section 3.1.5. These datasets were scraped over a 2 month period, starting in December 2024. Datasets B and C were scraped while the scraping tool was still being developed, due to their smaller size it would make it easier to use them as a benchmark. The datasets were scraped fully, and then an anonymous copy was made of them using a Python script to be used as a reference in this document.

[15] WebGL 2.0 Specification — Khronos Group. Last accessed on the 14th of April 2025 https://registry.khronos.org/webgl/specs/latest/2.0/

[16] WebGPU API — Mozilla Web Documentation. Last accessed on the 14th of April 2025 https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API

Dataset	Users	Unique Usernames	Accounts following > 1,500 users	Size (KB)
A	181	36,859	53	856
B	59	15,550	24	311
C	26	3,580	4	66
D	102	32,787	10	585
E	143	29,789	6	540

Figure 23: Table showing the results of scraping the accounts detailed in Figure 5.

From these metrics, we can deduce that the coverage problem detailed in section 4.1 has a large effect on the data produced, although the rate of coverage loss is inconsistent across datasets. If we divide the total users by the users that are following over 1,500 accounts, we find that dataset B has the highest ratio of accounts scraped to not scraped, with roughly 2 in every 5 accounts being unable to be scraped. On the other hand, dataset E offers the lowest ratio, with roughly 1 in every 24 accounts being unable to be scraped. With this in mind, we will conduct analysis on dataset E using the visualisation tool, however images of different processes being used on the other datasets will be available in appendix .12.
The time taken from clicking on dataset E to the dataset being visibly displayed was 11.03 seconds, which highlights concerns about the efficiency of the loading times discussed in section 4.2. Once the dataset is loaded, it immediately starts running force-atlas 2; which is a very visually appealing process. The performance of the graph improves notably as force-atlas 2 continues to "converge" on a state of equilibrium within the graph, as it does not need to perform as many calculations.

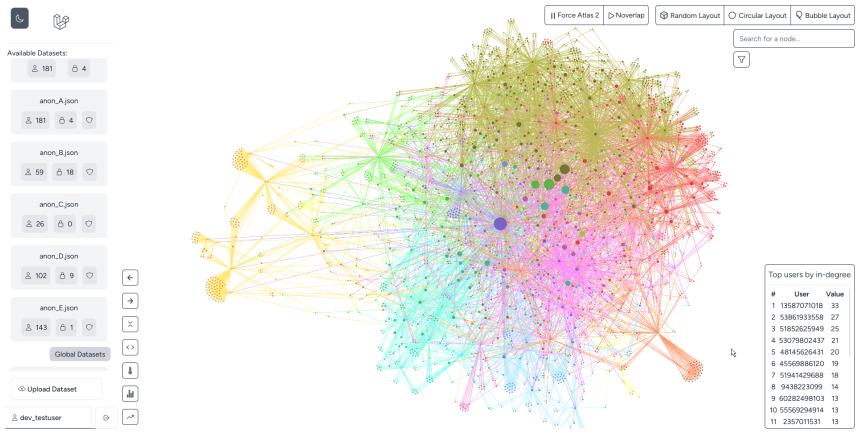


Figure 24: Dataset E, after running force-atlas 2 for several seconds.

One striking aspect of the graph is how force-atlas 2 and the Louvain method seem to synergize. A clear correlation can be seen in Figure 24 between the clusters that have been detected and the layout being created by force-atlas 2. In dataset E, this separation is very clear, however dataset A shows that some graphs have a much more inter-connected structure, where force-atlas 2 struggles to separate the graph into the same clusters that the Louvain method has detected. Furthermore, we can see a correlation in users belonging to each cluster in E: for example, the yellow community on the left-hand side of the graph consists mostly of accounts dedicated to architecture, while the light blue community is mostly christian fundamentalist meme pages and the turquoise group is primarily "ex-feminists", who have rejected women's liberation and moved to christian fundamentalism. This clear distinction between communities demonstrates the power of community detection, despite the concerns about the effectiveness of the Louvain method highlighted in section 2.2.

Figure 25: Dataset E in the circle-pack layout.

Another graph layout that shines in conjunction with the clustering algorithm is the circle-pack layout. This layout is able to neatly organise each community into a circle, and then sort each

circle by whichever centrality measure the user has selected. With this, we can clearly determine which clusters are the biggest, which contain the most users with a high measure of centrality and which clusters interact the most with others. By applying this to dataset E, we can see that the pink cluster at the right of the graph contains the most users, who are predominantly accounts dedicated to catholicism. If we sort the clusters using the Pagerank algorithm, we discover that the cluster that contains the most users with a high Pagerank score is the mustard-coloured cluster at the bottom; it contains accounts that promote the "looksmaxxing" trend (Halpin et al., 2025) and parrot conservative talking points.
However, the effectiveness of two of the other layouts, the random and circular layouts, is questionable. For the random layout, the graph is hard to interpret because the nodes have no clear structure; instead the graph looks tangled. The circle layout is also hard to interpret, this becomes the case especially for large graphs that have many edges. In Gephi, the user is given the option to order the nodes in the circle by a given measure or community. This may be a better approach to circular graph layouts than what is implemented in sigma.js.

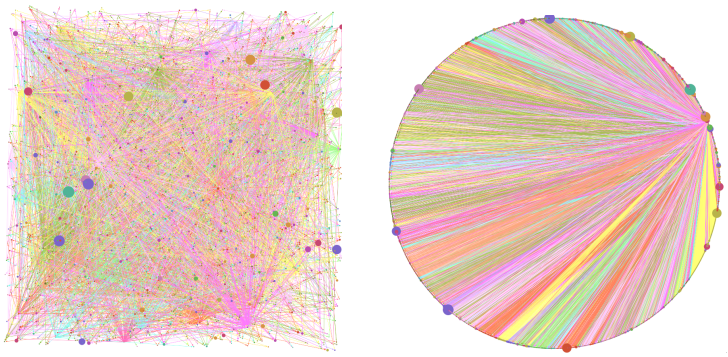


Figure 26: The random and circular layouts being run on dataset E.

While the interactive features of this program detailed in section 3.3.5 offer a wide range of different ways to analyse individual accounts, the act of hovering over a node - only to discover it hasn't been fully scraped is a disappointing one, and it reveals that perhaps further indication is needed that a node belongs to a fully scraped account. Although the process of clicking on a node to show only its neighbours is an incredibly useful feature in this application, in the case of dataset E it demonstrates that accounts with a clear theme are almost always following other accounts with a clear theme: such as european ethno-nationalist pages following each other. But, it does highlight the limitations of the breadth-first search that the scraping tool utilises; as the scope of the scrape is incredibly limited. If the user wanted to look more into a community highlighted, they would need to run another scrape on a target account, and then append that to the dataset.

4.4 Future Work

Looking towards the future, there are a wide array of improvements that could be made to both better the software produced by this project and the field of social network analysis as a whole. A major problem that plagues both this project and the field is the issue of tackling large-scale graph visualisation (Jeowicz et al., 2013). In the scope of this project, the next major step in improving the efficiency of the program would involve switching from a WebGL 2.0 rendering pipeline to a WebGPU pipeline instead, which (as previously discussed) would enable the web browser to utilize a wider range of the GPU's features. As the size of datasets used in social network analysis advances, the field looks towards analysing samples of given networks, instead of trying to visualise the entire network (Zhang, 2017). While this project does show smaller samples of a larger network, this is more out of technical limitations than by any intention. In the future, perhaps a feature that scrapes data directly from Instagram while the tool is being used is possible — where the user can specify an account to scrape and immediately see it added to the dataset.
As mentioned in section 1.1, this project could be used by a range of professions. Of these, it was mentioned that journalists could benefit from the data visualisation tool specifically as it could highlight ongoing trends in radicalisation content or extremist groups. Looking forwards, some more tools to calculate the frequency of different substrings within usernames or biographies would be an incredibly valuable tool to the journalist. For example, many of the usernames in the data gathered contained the dog-whistle "1488", referring to the two infamous white supremacist

concepts of the "14 words" and "88 precepts" (Bhat & Klein, 2020). Being able to quickly identify how many users in a dataset contain a term like this, or instead being able to quickly determine which terms are used the most, would be a fast and user-friendly method to understand ongoing trends and the underlying strcuture of the graph.
While this project focuses entirely on building a tool for visualising Instagram accounts, it should be acknowledged that many other social media platforms such as TikTok, X (formerly known as Twitter) and TruthSocial follow a very similar structure. For this reason, the data visualisation tool could (and should) be adapted to accept data from any social media platform. This could easily be done by storing an attribute in datasets regarding the social media that they were scraped from. Then, user data could be displayed in a more modular manner to accomodate for different data points that other social media platforms may use.
Under the consideration that social network analysis is an ever-adapting field with new methods of analysis being discovered frequently, this project needs to be actively updated in order to keep up with the field. This issue is directly implemented in Gephi where, as mentioned previously, the user may install any range of user-made plugins that add different forms of functionality such as layouts or other measures.
From the limitation of scope encountered in section 3.2.3 it is clear that more work needs to be done in terms of methods for gathering user data. Exacerbated by the 2018 Cambridge Analytica scandal, the relationship between data scrapers and social media platforms is a tug of war between the need for access to data and the enforcement of a user's privacy (Mancosu & Vegetti, 2020). In an ideal future, social media platforms should expose an API for verified academic institutions, such that researchers can use their data easily for sociological analysis while also preventing the data from being harvested by bad actors. Although, a more actionable improvement that lies within the scope of this project is to make the scraping API more modular, such that the user can choose and switch between API wrappers with ease — in the case one breaks.

5 Summary

As asserted in section 1.1, online radicalisation has become a widely discussed and prevalent issue in modern social studies. However, there exists a gap between academic literature surrounding the issue and actual action being taken to prevent the radicalisation of individuals (Mølmen & Ravndal, 2023). We can observe that when the radicalisation of someone either online or offline continues unmoderated, one may become motivated to commit acts of lone actor terrorism Whittaker, 2022. Whilst preventing radicalisation is a responsibility carried by society as a whole (Saleh et al., 2024), this project highlights two professions, Law Enforcement and Journalism, as roles that could have the greatest impact in preventing online radicalization.
For these roles, this project created a two-part software package that allows the user to scrape data from social media (using Instagram as a benchmark) in a breadth-first manner, and then generate interactive visualisations of that data in a network graph. The first part of this package, the scraping tool, was created in Python and is operated via the command-line. It utilizes the Instagrapi API wrapper for Instagram to gather data, and then outputs the dataset in a JSON-formatted file. The second part is a web-application, made with Laravel for simple web database management and sigma.js as a library for graph visualisation. The web application provides a more specialised experience than tools like Gephi, while suffering in terms of performance due to its WebGL 2.0 dependency. It makes use of different force-directed layouts, centrality measures and community detection algorithms to intuitively present data to the user.
The process of scraping large amounts of user data is a difficult task, with many caveats. Over the course of 2 months, more than 100,000 users were scraped from Instagram to use as sample data for this project. These users originated from breadth-first scrapes of different accounts that posted a wide array of white supremacist and other far-right talking points. Over those two months, Instagram employed a wide range of different automation detection countermeasures that slowed the scraping process to a crawl, highlighting the difficulties that many academics face when scraping data from social media (Mancosu & Vegetti, 2020). Importing this sample data into the visualisation tool demonstrated the effectiveness of the Louvain method for community detection, as clear links were demonstrated between the users in each community. Furthermore, graph layouts such as force-atlas 2 and circle-pack were shown to be very effective in showing the relationship between these clusters, and visualising the graph in an appealing manner.

6 Bibliography

Ahmed, Z., Kinjol, F. J., & Ananya, I. J. (2021). Comparative analysis of six programming languages based on readability, writability, and reliability. *2021 24th International Conference on Computer and Information Technology (ICCIT)*, 1–6.

Aouragh, M., Buttar, S., Meeks, E., & Shereen Sakr, L. (2011). Collateral damage: Oslo attacks and proliferating islamophobia. *Jadaliyya*. Retrieved March 21, 2025, from https://www.jadaliyya.com/Details/24298/Collateral-Damage-#Oslo-Attacks-and-Proliferating-Islamophobia

Aryaeinejad, K., & Scherer, T. L. (2024). The role of social networks in facilitating and preventing domestic radicalization. *National Institute of Justice*. https://www.ojp.gov/pdffiles1/nij/305795.pdf

Baele, S. J., Brace, L., & Coan, T. G. (2023). Uncovering the far-right online ecosystem: An analytical framework and research agenda. *Studies in conflict and terrorism*, *46*(9), 1599–1623.

Bakshy, E., Messing, S., & Adamic, L. A. (2015). Exposure to ideologically diverse news and opinion on facebook. *Science*, *348*(6239), 1130–1132. https://doi.org/10.1126/science.aaa1160

Beauchamp, M. A. (1965). An improved index of centrality. *Behavioral science*, *10*(2), 161–163.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). Manifesto for agile software development. http://www.agilemanifesto.org/

Bhat, P., & Klein, O. (2020). Covert hate speech: White nationalists and dog whistle communication on twitter. *Twitter, the public sphere, and the chaos of online deliberation*, 151–172.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, *2008*(10), P10008.

Burruss, G. W., & Lu, Y. (2021). The value of data visualization for translational criminology. In *Visual criminology* (pp. 251–269). Routledge.

Ellison, N. B., Steinfield, C., & Lampe, C. (2007). The benefits of facebook "friends:" social capital and college students' use of online social network sites. *Journal of Computer-Mediated Communication*, *12*(4), 1143–1168. https://doi.org/10.1111/j.1083-6101.2007.00367.x

Regulation (EU) 2016/679 of the European Parliament and of the Council (2016, May 4). Retrieved March 1, 2025, from https://www.legislation.gov.uk/eur/2016/679/article/6

Fletcher, J. M., & Wennekers, T. (2018). From structure to activity: Using centrality measures to predict neuronal activity [PMID: 28076982]. *International Journal of Neural Systems*, *28*(02), 1750013. https://doi.org/10.1142/S0129065717500137

Fortunato, S. (2010). Community detection in graphs. *Physics Reports*, *486*(3–5), 75–174. https://doi.org/10.1016/j.physrep.2009.11.002

Fortunato, S., & Barthelemy, M. (2007). Resolution limit in community detection. *Proceedings of the national academy of sciences*, *104*(1), 36–41.

Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry*, *40*(1), 35–41. Retrieved March 5, 2025, from http://www.jstor.org/stable/3033543

Freeman, L. C., et al. (2002). Centrality in social networks: Conceptual clarification. *Social network: critical concepts in sociology. Londres: Routledge*, *1*(3), 238–263.

Fruchterman, T. M., & Reingold, E. M. (1991). Graph drawing by force-directed placement. *Software: Practice and experience*, *21*(11), 1129–1164.

Ganesh, B., & Bright, J. (2020). Countering extremists on social media: Challenges for strategic communication and content moderation.

Gill, P. (2015). *Lone-actor terrorists: A behavioural analysis*. Routledge.

Halpin, M., Gosse, M., Yeo, K., Handlovsky, I., & Maguire, F. (2025). When help is harm: Health, lookism and self-improvement in the manosphere. *Sociology of Health & Illness*, *47*(3), e70015.

Hamilton, M. (2018). A threat assessment framework for lone-actor terrorists. *Fla. L. Rev.*, *70*, 1319.

Herath, C., & Whittaker, J. (2023). Online radicalisation: Moving beyond a simple dichotomy. *Terrorism and Political Violence*, *35*(5), 1027–1048. https://doi.org/10.1080/09546553.2021.1998008

Hewitt, S. (2024). *Lone-actor terrorism: Can anything be done?* Retrieved February 28, 2025, from https://www.birmingham.ac.uk/research/perspective/lone-actor-terrorism

"Hope not Hate". (2024). *"state of hate 2024"*. https://hopenothate.org.uk/2024/03/14/press-release-state-of-hate-2024/

Jacomy, M., Venturini, T., Heymann, S., & Bastian, M. (2014). Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PloS one*, *9*(6), e98679.

Jeowicz, T., Kudelka, M., Plato, J., & Snael, V. (2013). Visualization of large graphs using gpu computing. *2013 5th International Conference on Intelligent Networking and Collaborative Systems*, 662–667.

Kobourov, S. G. (2012). Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv:1201.3011*.

Krishna, S. V., Jadhav, O., Ardhapurkar, P., Valmiki, M., Agrawal, S., Bulusu, R., Dinde, P., Wandhekar, S., Cherif, A. R., Tyagi, V., Gupta, P. K., Tomar, R., Flusser, J., Ören, T., & Singh, M. (2025). Performance comparison of julia with c and python for solving computational problems. In *Advances in computing and data sciences* (pp. 35–45, Vol. 2194). Springer Nature Switzerland.

Latif, U. K., & Kusumasari, T. F. (2018). Comparison between yii frameworks and laravel in 3 different version for viewing large data of shipyard industry in indonesia. *International Journal of Innovation in Enterprise System*, *2*(1), 13–18.

Leavitt, H. J. (1951). Some effects of certain communication patterns on group performance. *The journal of abnormal and social psychology*, *46*(1), 38.

Leetaru, K. (2011). Culturomics 2.0: Forecasting large-scale human behavior using global news media tone in time and space. *First Monday*, *16*(9). https://doi.org/10.5210/fm.v16i9.3663

Mancosu, M., & Vegetti, F. (2020). What you can scrape and what is right to scrape: A proposal for a tool to collect public facebook data. *Social media + society*, *6*(3).

Martino, F., & Spoto, A. (2006). Social network analysis: A brief theoretical review and further perspectives in the study of information technology. *PsychNology Journal*, *4*, 53–86.

McCulloh, I., Armstrong, H., & Johnson, A. (2013). *Social network analysis with applications*. John Wiley & Sons.

Mølmen, G. N., & Ravndal, J. A. (2023). Mechanisms of online radicalisation: How the internet affects the radicalisation of extreme-right lone actor terrorists. *Behavioral Sciences of Terrorism and Political Aggression*, *15*(4), 463–487. https://doi.org/10.1080/19434472.2021.1993302

Munn, L. (2019). Alt-right pipeline: Individual journeys to extremism online. *First Monday*, *24*(6). https://doi.org/10.5210/fm.v24i6.10108

Newman, M. E. (2004). Fast algorithm for detecting community structure in networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, *69*(6), 066133.

O'Connor, Z. (2015). Colour, contrast and gestalt theories of perception: The impact in contemporary visual communications design. *Color research and application*, *40*(1), 85–92.

Page, L. (1998). Method for node ranking in a linked database. *US patent*, *7058628*.

Perliger, A., & Pedahzur, A. (2011). Social network analysis in the study of terrorism and political violence. *PS: Political Science & Politics*, *44*(1), 45–50.

Rogers, D. L. (1974). Sociometric analysis of interorganizational relations: Application of theory and measurement. *Rural Sociology*, *39*(4), 487.

Saleh, N. F., Roozenbeek, J., Makki, F. A., Mcclanahan, W. P., & Van Der Linden, S. (2024). Active inoculation boosts attitudinal resistance against extremist persuasion techniques: A novel approach towards the prevention of violent extremism. *Behavioural Public Policy*, *8*(3), 548–571. https://doi.org/10.1017/bpp.2020.60

Shirreff, L. (2024). The obscure russian-linked 'news' outlet fuelling violence on britain's streets. *The Telegraph*. Retrieved March 1, 2025, from https://www.telegraph.co.uk/news/2024/08/03/obscure-russian-linked-news-outlet-fuelling-violence/

Spring, M. (2024). The real story of the news website accused of fuelling riots. *BBC News*. Retrieved March 1, 2025, from https://www.bbc.co.uk/news/articles/c5y38gjp4ygo

Trezza, D. (2023). To scrape or not to scrape, this is dilemma. the post-api scenario and implications on digital research. *Frontiers in sociology*, *8*, 1145038.

Vasist, P. N., Chatterjee, D., & Krishnan, S. (2024). The polarizing impact of political disinformation and hate speech: A cross-country configural narrative. *Information Systems Frontiers*, *26*(2), 663–688.

Whittaker, J. (2022). Rethinking online radicalization. *Perspectives on Terrorism*, *16*(4), pp. 27–40. Retrieved October 7, 2024, from https://www.jstor.org/stable/27158150

Whittaker, J., Rogers, E., Micallef, N., & Correia, S. (2024). *What are the links between social media algorithms, generative ai and the spread of harmful content online?* (Tech. rep.).

Swansea University. Retrieved March 1, 2025, from https://committees.parliament.uk/writtenevidence/132875/pdf/

Wilmot, C. (2024). Did russian disinformation fuel the southport protests? *The Bureau of Investigative Journalists*. Retrieved March 1, 2025, from https://www.thebureauinvestigates.com/stories/2024-08-02/did-russian-disinformation-fuel-the-southport-protests/

Yu, N., Ouyang, Z., Wang, H., Tao, D., & Liang, J. (2022). The effects of smart home interface touch button design features on performance among young and senior users. *International Journal of Environmental Research and Public Health*, *19*(4), 2391. https://www.proquest.com/scholarly-journals/effects-smart-home-interface-touch-button-design/docview/2632750512/se-2

Zeffman, H., Symonds, T., & Catt, H. (2025). No plans to expand definition of extremism - minister. *BBC News*. https://www.bbc.co.uk/news/articles/c3rwvjlln9wo

Zhang, F. (2017). Large-scale graph visual analytics.

7 Appendix: Code Samples

.1 Implementation of command-line parsing using argparse.

```
import argparse as arg

(...)

def main():
    argparser = arg.ArgumentParser(description="Create, or add to sociogram databases (.sn) by supplying a root user to scrape.",
                                   prog="Sociogram Scraper")
    argparser.add_argument("-v", "--verbose", help="Verbose output mode",
                           action="store_true")
    argparser.add_argument("-d", "--depth", help="depth of search", default=2)
    specGroup = argparser.add_mutually_exclusive_group(required=True)
    specGroup.add_argument("-t", "--target", help="Username of account to start scrape on, either this or -i are required")
    specGroup.add_argument("-i", "--id", help="ID of account to start scrape on, either this or -t are required")
    argparser.add_argument("file", help="File to create/use, required")
    argparser.add_argument("-a", "--anon", action="store_true", help="Anonymous data collection mode: Minimal data collected. Only applicable on new or already anonymous databases")
    args = argparser.parse_args()

(...)

if __name__ == "__main__":
    main()
```

.2 Implementation of file verification and creation

```
import global_variables as glob
import time, os, json

def validatePath():
    glob.log(f"validating {glob.db_file}")

    if not glob.db_file.endswith(".json"):
        raise FileNotFoundError("File is not of .json filetype!")

    if os.path.isfile(glob.db_file) and os.access(glob.db_file, os.W_OK):
        glob.log("File is writeable and exists... importing")
        try:
            with open(glob.db_file, "r") as file_inst:
                jsonDict = json.load(file_inst)
                assert(jsonDict["anonymous"] == glob.anonMode)
                jsonDict["fileUpdated"] = time.time()
                for user in jsonDict['users']:
                    glob.reachedUsers.append(user['id'])

            with open(glob.db_file, "w") as file_inst:
                file_inst.write(json.dumps(jsonDict))

        except AssertionError:
             print("Attempted to open file with wrong anonymity tag!")
             raise SystemExit

        except:
```

```
            if input("Failed to validate file!\nWould you like to overwrite
            ↪  it?\t[Y/N] = ") in ["Y", "y"]:
                createFile()
            else:
                raise SystemExit

    else:
        glob.log("File does not exist, creating")
        createFile()

def createFile():
    with open(glob.db_file, "w") as file_inst:
            unixTime = time.time()
            newJsonData = {
                "anonymous": glob.anonMode,
                "fileCreated": unixTime,
                "fileUpdated": unixTime,
                "users": []
            }
            file_inst.write(json.dumps(newJsonData))
```

.3 Implementation of file addition via adding users

```
def addUser(user):
    if not userExists(user['id']):
        glob.reachedUsers.append(user['id'])
        with open(glob.db_file, "r") as file_inst:
            jsonDict = json.load(file_inst)

        jsonDict["users"].append(user)

        with open(glob.db_file, "w") as file_inst:
            file_inst.write(json.dumps(jsonDict))

    else:
        glob.log(f"User {user['id']} already exists in DB!")

def userExists(pk):
    return pk in glob.reachedUsers
```

.4 Logging in manually with Instagrapi

```
def manual_login():
    cl = glob.client
    username =   input("Please enter Instagram username >>> ")
    password = getpass("Please enter Instagram password >>> ")

    try:
        cl.login(username, password)
        cl.dump_settings("session.json")
    except Exception as e:
        glob.log(e)
        if input("Failed to login! Would you like to try again?\t[Y/N] = ") in
        ↪  ["Y","y"]:
            validate_session()
        else:
            raise SystemExit
```

.5 An implementation of the `user_as_dict` function.

```python
def user_as_dict(user, following):
    if user.is_private:
        userDict = {
        "id": user.pk,
        "username": user.username,
        "follower_count": user.follower_count,
        "following_count": user.following_count,
        "url": user.external_url,
        "is_private": True
        }
    else:
        userDict = {
            "id": user.pk,
            "username": user.username,
            "follower_count": user.follower_count,
            "following_count": user.following_count,
            "url": user.external_url,
            "is_private": False,
            "bio": user.biography,
            "full_name": user.full_name,
            "post_count": user.media_count,
            "following": []
        }
        for pk in following:
            userDict["following"].append([pk.pk, pk.username])

    return userDict
```

.6 Processing of user uploaded datasets

```php
public function submit()
    {
        $this->validate();
        $this->isUploading = true;
        $file = $this->file;
        $fileContents = file_get_contents($file->getRealPath());
        $decodedJson = json_decode($fileContents, true); // Validate JSON

        if (json_last_error() !== JSON_ERROR_NONE) {
            $this->message = 'JSON format invalid! ' . json_last_error_msg();
            $this->isUploading = false;
            return;
        }
        $validationResult = $this->validateJsonStructure($decodedJson);
        if (isset($validationResult['error'])) {
            $this->message = $validationResult['error'];
            $this->isUploading = false;
            return;
        }
        $privateAccountCount = $validationResult['privateAccounts'];
        $userCount = count($decodedJson['users']);
        $isAnonymous = $decodedJson['anonymous'];

        $path = $file->storeAs('datasets', $file->getClientOriginalName(),
        ↪  'public');
        Dataset::query()->create([
            'user-id'       => Auth::id(),
            'user-count'    => $userCount,
```

```
            'private-count' => $privateAccountCount,
            'is-anon'       => $isAnonymous,
            'is-global'     => $this->isGlobal,
            'file-name'     => $file->getClientOriginalName(),
            'mime-type'     => $file->getClientMimeType(),
            'path'          => $path,
            'disk'          => 'local',
            'size'          => $file->getSize(),
        ]);

        session()->flash('message', 'File uploaded successfully!');
        $this->reset();
        $this->isUploading = false;
        $this->dispatch('dataset-created');
    }
```

.7 Verification of dataset structure

```
public function validateJsonStructure(array $decodedJson)
    {
        $privateAccounts = 0;

        if (!isset($decodedJson['anonymous']) ||
        ↪  !is_bool($decodedJson['anonymous'])) {
            return ['error' => '"anonymous" must be a boolean.'];
        }

        if (!isset($decodedJson['fileCreated']) ||
        ↪  !is_numeric($decodedJson['fileCreated'])) {
            return ['error' => '"fileCreated" must be a numeric timestamp.'];
        }

        if (!isset($decodedJson['fileUpdated']) ||
        ↪  !is_numeric($decodedJson['fileUpdated'])) {
            return ['error' => '"fileUpdated" must be a numeric timestamp.'];
        }

        if (!isset($decodedJson['users']) || !is_array($decodedJson['users']))
        ↪  {
            return ['error' => '"users" must be an array.'];
        }

        foreach ($decodedJson['users'] as $user) {
            if (!isset($user['id']) || !is_string($user['id'])) {
                return ['error' => 'User must have a valid "id" string.'];
            }

            if (isset($user['username']) && !is_string($user['username']) &&
            ↪  $user['username'] !== null) {
                return ['error' => 'User must have a valid "username"
                ↪  string.'];
            }

            (...)

            if (!isset($user['following'])) {
                $privateAccounts++;
            }

            if (isset($user['following']) && is_array($user['following'])) {
```

```
                foreach ($user['following'] as $following) {
                    if (!is_array($following)) {
                        return ['error' => 'Each "following" entry must be an
                        ↪  array with a user ID and username.'];
                    }
                }
            }
        }

        return ['privateAccounts' => $privateAccounts];
    }
```

.8 Generating graphs from .json data sets

```
const anon = jsonData['anonymous'];
var currentUsers = [];

// First parse for adding Nodes
jsonData['users'].forEach(function (user) {
    if (!currentUsers.includes(user['id'])) {
        var nodeLabel = anon ? user['id'] : user['username'];
        graph.addNode(user['id'], { label: nodeLabel, x: randomCoord(), y:
        ↪  randomCoord(), size: 2 });
        currentUsers.push(user['id']);
    }

    // Add nodes for following of each node if onlyScanned is false
    if (user.hasOwnProperty('following') && !onlyScanned) {
        user['following'].forEach(function (fUser) {
            if (!currentUsers.includes(fUser[0])) {
                var followNodeLabel = anon ? fUser[0] : fUser[1];
                graph.addNode(fUser[0], { label: followNodeLabel, x:
                ↪  randomCoord(), y: randomCoord(), size: 2 });
                currentUsers.push(fUser[0]);
            }
        });
    }
});

// Second go-over for edges
jsonData['users'].forEach(function (user) {
    if(user.hasOwnProperty('following')){
        user['following'].forEach(function (target) {

            // if onlyScanned is set, target node must already be in graph
            if (currentUsers.includes(target[0]) || !onlyScanned){
                graph.mergeEdge(user['id'], target[0], {type: "arrow"});
            }
        });
    }
});
```

.9 Implementation of the LayoutWorker class

```
// Store all current workers here
const layoutWorkers = [];

// Utility to stop all workers if necessary
function stopLayouts() {
```

```
    layoutWorkers.forEach(worker => worker.stop());
    if (cancelCurrentAnimation) {
        cancelCurrentAnimation();
        cancelCurrentAnimation = null;
    }
}

// Layout worker class, acts as a wrapper for workers
export class LayoutWorker {
    constructor({ layoutInstance, startLabelId, stopLabelId, buttonId }) {
        this.layout = layoutInstance;
        this.startLabel = document.getElementById(startLabelId);
        this.stopLabel = document.getElementById(stopLabelId);
        this.button = document.getElementById(buttonId);

        this.button.addEventListener("click", () => this.toggle());

        layoutWorkers.push(this);
    }

    stop() {
        this.layout.stop();
        this.startLabel.style.display = "flex";
        this.stopLabel.style.display = "none";
    }

    start() {
        stopLayouts();
        this.layout.start();
        this.startLabel.style.display = "none";
        this.stopLabel.style.display = "flex";
    }

    toggle() {
        if (this.layout.isRunning()) {
            this.stop();
        } else {
            this.start();
        }
    }
};
```

.10 Example centrality measure setters & normalisation

```
// set the size of each node to the normalized centrality measure
function normalizeSize(graph, vals, attribute){
    const vMax = vals[1];
    const vMin = vals[0];
    const vRange = vMax - vMin;

    graph.forEachNode(node => {
        graph.mergeNodeAttributes(node, {
            size: MIN_SIZE + (((graph.getNodeAttribute(node, attribute) -
            ↪  vMin)*MINMAX_RANGE) / vRange),
          });
    });
}

// Display Eigenvector Centrality for graph
export function sortEigen(graph){
```

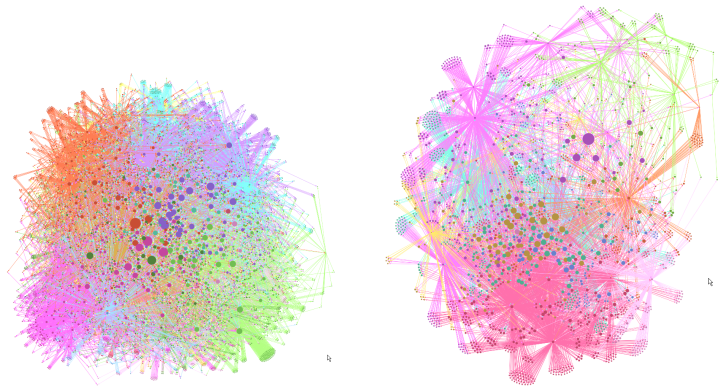
```
    var att = "eigenvectorCentrality";
    eigenvectorCentrality.assign(graph);
    normalizeSize(graph, getMinMaxAttribute(graph, att), att);
    setDisplayBox(graph, "eigenvector centrality", att);
}

// Display Pagerank for graph
export function sortPagerank(graph){
    var att = "pagerank";
    pagerank.assign(graph);
    normalizeSize(graph, getMinMaxAttribute(graph, att), att);
    setDisplayBox(graph, "pagerank", att);
}

// Set the table to the current centrality measure
function setDisplayBox(graph, method, attribute){
    var objectArray = [];
    currentAtt = attribute;

    if(attribute == "in"){
        graph.forEachNode(node => {
            var atts = graph.getNodeAttributes(node);
            objectArray.push([atts['label'], graph.inDegree(node)]);
        });
    } else if(attribute == "out"){
        graph.forEachNode(node => {
            var atts = graph.getNodeAttributes(node);
            objectArray.push([atts['label'], graph.outDegree(node)]);
        });
    } else {
        graph.forEachNode(node => {
            var atts = graph.getNodeAttributes(node);
            objectArray.push([atts['label'],
            ↪  Number(atts[attribute].toPrecision(3))]);
        });
    }

    // Sort the array by centrality, descending
    objectArray.sort((a, b) =>  a[1] - b[1] || a[0] - b[0]).reverse();
    if(objectArray.length > MAX_VALUES){
        objectArray = objectArray.slice(0,MAX_VALUES);
    }
    Livewire.dispatch('update-toptable', {sortString: method, currentArray:
    ↪  objectArray});

}
```

.11 Setting node & edge reducers

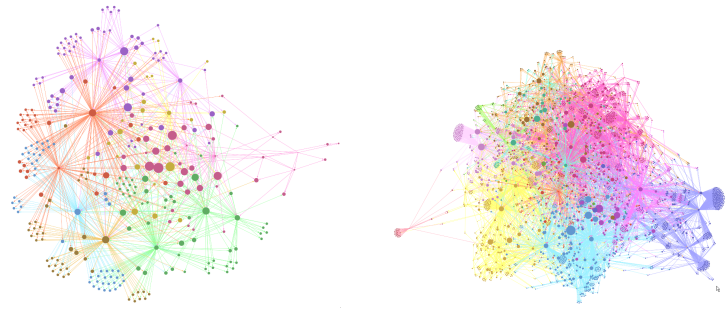
```
function setReducers(returnGraph){
    sigmaInstance.setSetting("nodeReducer", (node, data) => {
        const res = { ...data };

        // Hide if node clicked but this isn't a neighbour
        if (clickedNeighbors && !clickedNeighbors.has(node) && hoveredNode !==
        ↪  node) {
          res.label = "";
          res.hidden=true;
        }

```

```
        // Highlight if selected
        if (selectedNode === node) {
          res.highlighted = true;

        // Hide if not in search query
        } else if (suggestions) {
          if (suggestions.has(node)) {
            res.forceLabel = true;
          } else {
            res.label = "";
            res.hidden=true;
          }
        }

        return res;
      });

    sigmaInstance.setSetting("edgeReducer", (edge, data) => {
        const res = { ...data };
        // Get source and target nodes for edge
        var source = returnGraph.source(edge)
        var target = returnGraph.target(edge)

        // Hide if either are hidden
        if(source.hidden || target.hidden){
          res.hidden = true
        }
        return res;
      });
}
```

.12 Visual representations of datasets A-D

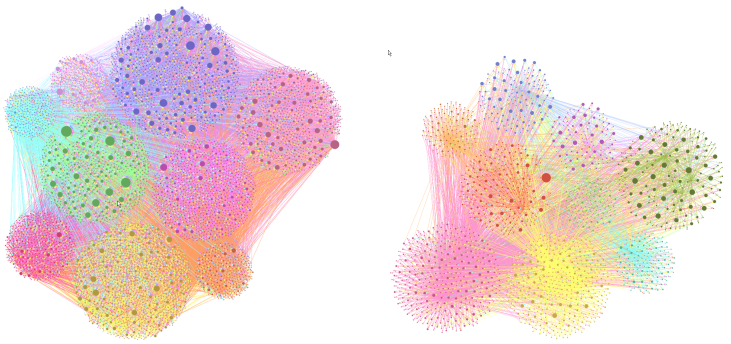


(a) Dataset A (b) Dataset B

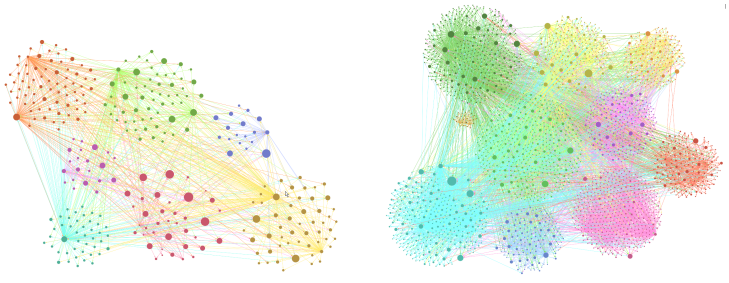


(c) Dataset C (d) Dataset D

Figure 27: Datasets A-D with the force-atlas 2 layout.



(a) Dataset A (b) Dataset B



(c) Dataset C (d) Dataset D

Figure 28: Datasets A-D with the circle-pack layout.