Applying Network Analysis to Humanities

Yann Ryan, Iiro Tihonen

2022-12-01

Contents

Chapter 1

About this Book

This book is intended to be read alongside the the "Applying Network Analysis to Humanities' course at the University of Helsinki, beginning November 2022. This course is aimed at complete beginners to both R and network analysis, although you'll still get plenty out of it if you have experience with either.

It focuses on **applied approaches** to network analysis and humanities data. Rather than cover network science in exhaustive detail, you'll learn how to **find**, **extract**, **clean**, **visualise**, and **analyse** humanities and cultural datasets from a network perspective. Additionally, we'll focus on the **problems** and **pitfalls** of using networks, specific to humanities data.

There are two chapters for each week, one for each session. Browse through the chapters using the menu to the left.

For most classes, there is an equivalent set of exercises. We'll begin them during the class but they can be completed afterwards. You'll probably want to have this book at hand to complete them. They are available as R markdown notebooks, using the CSC notebooks service. You'll need a CSC account, and you'll be sent a join code at the beginning of the course. If you don't have either of these things, contact the course leader.

1.1 About the course

The course is held online, via Zoom, on Wednesdays and Fridays, between 08.15 and 09.45, starting on November 02.

Assessment is the following:

- Completion of weekly assignments (pass/fail, carried out during class and finished afterwards if necessary).

- Presentation on final project (1/5 of the grade), **given on Wednesday or Friday of the final week.**
- Final project (4/5 of the grade, **submitted on January 17 at the latest.**

Full details on the course page on Studies Service

1.2 Final Project

The final project is due on January 17.

You can start on the project as early as you like. To help you with your project, you'll present your ideas and proposed methods in the final week of the course. At this point, you should hopefully have some preliminary research and outputs, or at the least a plan for your project and how it will come together, which you will communicate to the group in a presentation. Because of this, it's recommended to begin preliminary work on your project a couple of weeks before this.

You'll also be asked to give feedback and ask questions of the other class participants, with the aim of helping their own projects.

The final project tasks you with using your network data and related data model to carry out an analysis. The project should take the form of an R Markdown notebook, which you'll learn how to create over the next few weeks. An R Markdown document is a format which allows you to combine text, chunks of code, and the output of those chunks. You'll write up this document and then turn it into a HTML page - a process known as 'knitting'.

When you're finished, upload to the course Moodle area before deadline.

1.3 Reading

Each week will have one piece of set reading, usually an article or book chapter, to be discussed in-class the following week.

Week 1: Ruth Ahnert, Sebastian E Ahnert, Metadata, Surveillance and the Tudor State, *History Workshop Journal*, Volume 87, Spring 2019, Pages 27–51, https://doi.org/10.1093/hwj/dby033

(http://www.scottbot.net/HIAL/index.html@p=6279.html will also be helpful)

Week 2: Beheim, B., Atkinson, Q.D., Bulbulia, J. *et al.* Treatment of missing data determined conclusions regarding moralizing gods. *Nature* **595**, E29–E34 (2021). https://doi.org/10.1038/s41586-021-03655-4

Week 3: Sections from Jonathan Blaney, "Introduction to the Principles of Linked Open Data," Programming Historian 6 (2017), https://doi.org/10.46430/phen0068.

Week 4: Ahnert et. al. (2021). *The Network Turn: Changing Perspectives in the Humanities* (Elements in Publishing and Book Culture). **Chapter 5, 'Quantifying Culture' (https://www.cambridge.org/core/elements/network-turn/CC38F2EA9F51A6D1AFCB7E005218BBE5)**

Week 5: Venturini, T., Jacomy, M., & Jensen, P. (2021). What do we see when we look at networks: Visual network analysis, relational ambiguity, and force-directed layouts. *Big Data & Society*, 8(1). https://doi.org/10.1177/20539517211018488

Week 6: Silvia Donker, 'Networking Data. A Network Analysis of Spotify's Socio-Technical Related Artist Network: Vienna Music Business Research Days', International Journal of Music Business Research 8, no. 1 (1 April 2019): 67–101. https://pure.rug.nl/ws/portalfiles/portal/96957258/volume_8_no_1_april_2019_donker_end.pdf

(http://www.scottbot.net/HIAL/index.html@p=41158.html will also be helpful)

Week 7: Mushon Zer-Aviv, 'If Everything Is a Network, Nothing Is a Network', *Visualising Information for Advocacy*, accessed 17 October 2022, https://visualisingadvocacy.org/blog/if-everything-network-nothing-network.

(https://raley.english.ucsb.edu/wp-content/Engl800/Galloway-some-things-unrepresentable.pdf also interesting)

There is no set textbook, but the following are good general introductions to networks:

Ahnert, R., Ahnert, S., Coleman, C., & Weingart, S. (2021). *The Network Turn: Changing Perspectives in the Humanities* (Elements in Publishing and Book Culture). Cambridge: Cambridge University Press. doi:10.1017/9781108866804

Short, very readable volume on networks, specifically focused on humanities applications. Free, open access copy available through the above link.

Easley, D., and Kleinberg,J. (2010). *Networks, Crowds, and Markets: Reasoning about a Highly Connected World.* Cambridge University Press.

Very comprehensive textbook on networks, mostly relating to economics, sociology, computing. Pre-publication draft is available for free on the book website.

Barabási, A.-L. (2002). *Linked: The New Science of Networks.* Perseus Pub.

Popular science book on networks, very influential in bringing the science of networks to a popular audience.

Newman, M. E. J. (2018). *Networks* (Second edition). Oxford University Press.

Comprehensive textbook of network theory, recommended if you want to understand algorithms etc. in more detail.

Also worth checking out is the extensive bibliography and journal by the Historical Network Research Community.

1.4 Slides

There are also a set of slides for each week, which you can access here:

- Week 1: class 1, class 2
- Week 2: class 1, class 2
- Week 3: class 1, class 2
- Week 4: class 1, class 2
- Week 5: class 1, class 2
- Week 6: class 1, class 2
- Week 7: class 1

Chapter 2

Week 1, class 1: Course Introduction

Slides for this week

2.1 Introduction

This course will teach you how to use networks to ask (and hopefully answer) questions relating to humanities. More specifically than this, you'll use the science of networks to model entities, and the relationships between them. To do this, you'll use your knowledge of a subject to build a *data model*: a way to conceptualise the way your data all fits together, in a way that allows you to extract network data from it.

2.2 How to use this book and follow the course.

Each week, you will need to complete a short assignment. This takes the form of an editable R notebook, which can be found on your CSC notebooks workspace. Soon, you'll log into the CSC notebooks server, and open a *source* copy of this section, which is interactive, allowing you to edit and run code. After each week, you'll open the interactive notebook, complete the assignment, and upload the file.

As we haven't introduced R yet, for this week, your only task is to make a copy of the exercise in your personal folder, 'knit' it, and send the resulting file to the course leader. In the following weeks, this is the method you'll need to use to submit the assignments, so this is an opportunity to familiarise yourself with it, and to iron out any problems.

2.3 What do we mean by a 'network'?

The word 'network' is ubiquitous in our daily lives. The Oxford English Dictionary tells us the term itself originally was first defined as `Work (esp. manufactured work) in which threads, wires, etc., are crossed or interlaced in the fashion of a net.` (think patch work), but now we commonly use it for any complex system of interrelated things: we all use social networking, and the telephone, railway, and road networks; an area of a city might be described as having networks of alleyways, or a historian might write about a network of trading posts.

These are all in some way metaphorical. The networks we'll learn about and use on this course have a more specific definition. In mathematics, a network is an object made up of *things* and *connections*. To use the standard language, the *things* are called **nodes** (or sometimes vertices), and the *connections* called **edges**.

These things (and connections) could be almost anything. Some typical examples include:

- Nodes are people, and the edges their friendships or followers, as in a typical social network.
- The nodes are books and authors, and the connection is 'written by'.
- Nodes are web pages on the internet

In this final example, what might the connections be?

The connections between web pages are generally the hyperlinks: the system of links known as the World Wide Web.

2.4 Networks in the Humanities

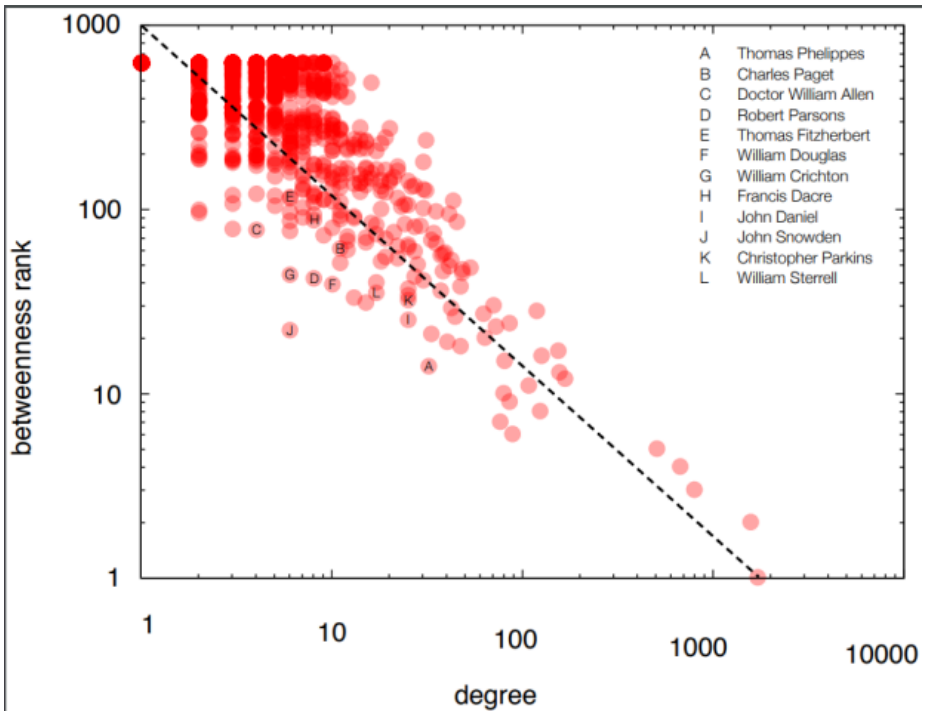
Networks have long been used to understand aspects of historical or cultural change. Some typical uses of complex networks in humanities subjects include spatial networks, line of sight or pottery similarity networks in archaeology, historic road networks, and character networks in plays. In this course, we'll mostly work with networks as tools for **descriptive** and **exploratory** data analysis. But network science also has wide uses in **statistical and predictive modelling**.

2.4.1 Some Examples

Before we get in to some of the details, it's worth going through a few examples of interesting projects which have combined humanities data with network science.

2.4.1.1 Tudor networks

As found in your reading this week, Ahnert and Ahnert used a number of network measurements to make a 'network fingerprint' for individuals in an archive of Tudor state papers. Using this measure, they were able to identify or confirm spies and collaborators. Specifically, they looked at individuals with a high betweenness (a bridging measurement) in comparison to their degree (measuring their overall connections): represented by the points underneath the trendline in the graph below. The biggest outlier is Jon Snowden: an alias of John Cecil, a Catholic priest in exile in Elizabeth I's regime. Looking at others with a similar profile revealed other conspirators.



2.4.1.2 Networking Cultural History

In below video, accompanying a paper in *Science*, researchers graphed places of birth and death—taken from Wikidata—as nodes and edges, using it as a way to trace the movement of culture from one area of the world to another, and how this changed over time.

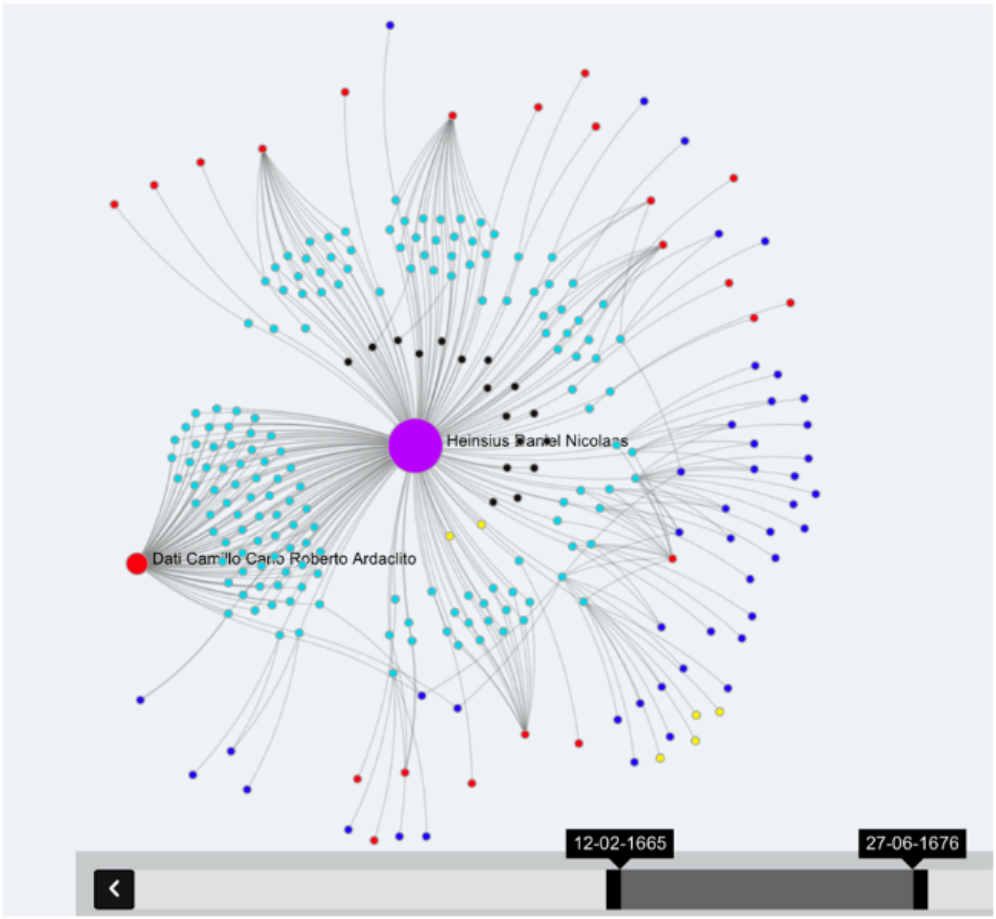
Given that the information is taken from Wikidata, what problems could you envisage with this study?

While Wikidata is a large data source, it is obviously very partial and has a

particular focus, for example in favour of white, male, Western Europeans. The study may just replicate the biases of Wikidata itself, hiding the important cultural influences on the west, from other parts of the world.

2.4.1.3 Using Multi-Layered Networks to Disclose Books in the Republic of Letters

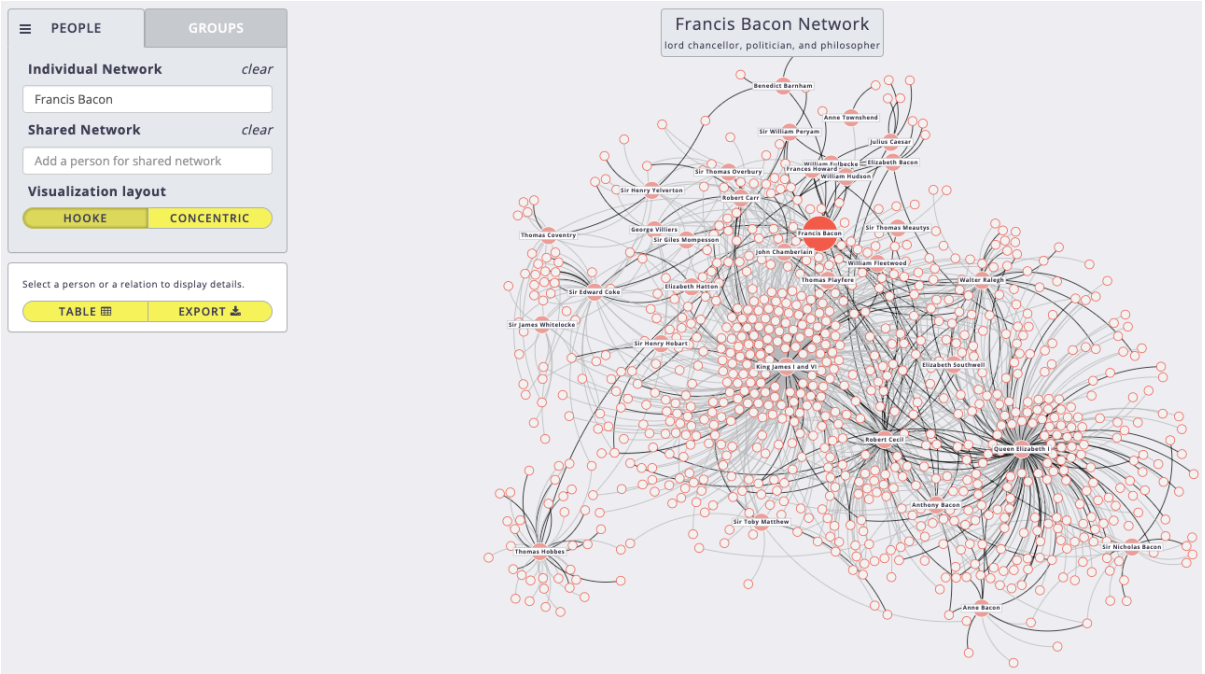
In this project, Ingeborg van Vugt made a 'multi-layer network' by combining a network of letters with a network of book mentions. The paper compared the networks of the Dutch philologist, Nicolaas Heinsius, and the Florentine librarian Antonio Magliabechi. Dynamic visualisations were made using the Nodegoat software. One key finding was that having access to books turned these people into experts - scholars would send their books to them, and then in turn they would use this information to inform their correspondents. Van Vugt showed that these two separate spheres (books and correspondence) were related to each other, that a book dedication could influence whether two individuals would write to each other, and so forth.



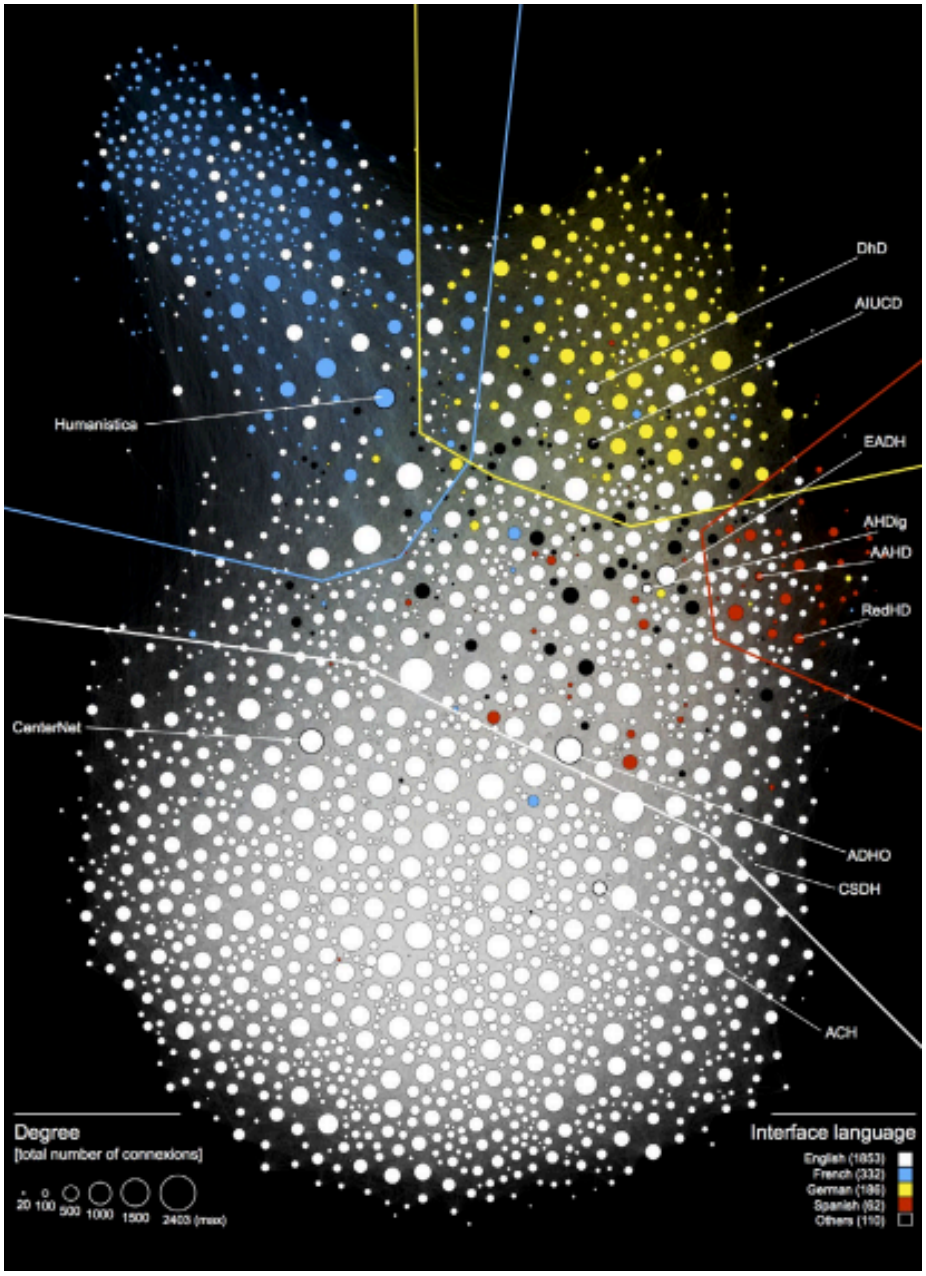
2.4.1.4 Six Degrees of Francis Bacon

The Six Degrees of Francis Bacon project used co-occurrence networks (a technique we'll learn about later) to infer links between early modern individuals. Specifically, they looked for co-occurrences of individuals in the Oxford Dictionary of National Biography, inferring links if two individuals were mentioned together more than was statistically significant. This data was then turned into an interactive social network.

Rather than answering a specific question, the Six Degrees project is a good example of how we can use networks as exploratory tools. The interactive site can be used to explore patterns and groups who were likely connected, and can be used as a starting-point to understanding the social relations of a figure of interest.


PEOPLE
GROUPS
Individual Network
clear
Francis Bacon
Shared Network
clear
Add a person for shared network
Visualization layout
HOOKE
CONCENTRIC
Select a person or a relation to display details.
TABLE
EXPORT
Francis Bacon Network
lord chancellor, politician, and philosopher
Benedict Barnham
Anne Townshend
Sir William Peryam
Julius Caesar
Elizabeth Bacon
William Fulbecke
Frances Howard
William Hudson
Sir Thomas Overbury
Sir Henry Yelverton
Robert Carr
George Villiers
Thomas Coventry
Sir Giles Mompesson
Francis Bacon
Sir Thomas Meautys
John Chamberlain
William Fleetwood
Thomas Playfere
Walter Raleigh
Elizabeth Hatton
Sir Edward Coke
Sir James Whitelocke
Sir Henry Hobart
King James I and VI
Elizabeth Southwell
Robert Cecil
Queen Elizabeth I
Anthony Bacon
Sir Toby Matthew
Thomas Hobbes
Sir Nicholas Bacon
Anne Bacon

2.4.1.5 Mapping the digital humanities community of Twitter



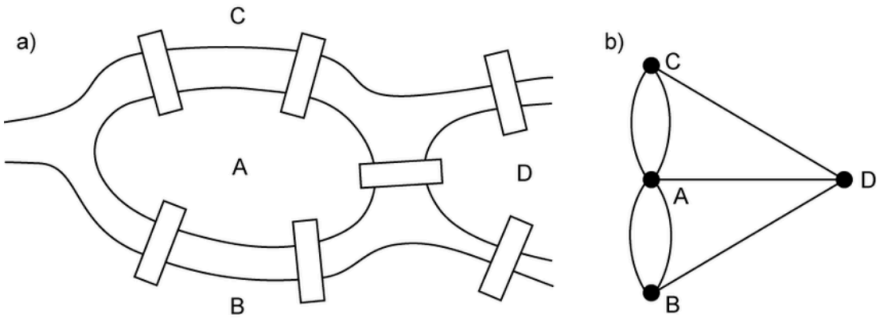
Network analysis has naturally been used extensively to understand the structure of social networks and sites like Reddit. In this study, the authors measured the centrality of a group of 2,500 Twitter users self-identified as digital humani-

ties practioners. They found that there were separate communities of languages, and that French was particularly isolated.

2.5 Graph Theory

Representing data in this way allows us to use the mathematics of *graph theory* to analyse and explore it. The origins of graph theory date back to the 18th century, and the mathematician Leonard Euler. The city of Königsberg (now Kaliningrad) was built on a river with two islands and a system (a network, perhaps?) of seven bridges connecting them. The inhabitants of the city had long wondered if there it was possible to devise a route which would cross each bridge exactly once.

Euler solved this problem by abstracting the bridge system into what we now call a graph: a structure with nodes and edges. He used this approach to prove that the path was only possible if the graph had exactly zero or two nodes with a *degree* (a count of the total connections) with an odd number.



This process: abstracting a system to its connections, and devising mathematical rules to understand it, forms the basis of modern graph theory. Graph theory has methods to perform all sorts of calculations on the network: both on the individual nodes (for example to find the most important nodes by various measurements), and on the network as a whole. Rather than think about each individual relationship separately, a network model means that we can understand some more about how each part is interrelated, how the structure itself is important, and how one node might affect another in complex ways.

2.6 Why use networks in humanities?

Network analysis has become one of the most popular (and perhaps one of the most overused) techniques in the digital humanities toolbox. Over this course, we'll ask you to think critically about the methods you use and when they might not always be the best way to think about your data.

There are plenty of circumstances when a network model is not the best was of representing a dataset, as we'll learn. However, representing data as a network has a number of unique benefits:

- Networks allow us to reduce or understand **complexity:** we can reduce complicated data to its overall structure. Network visualisations, when used well, allow us to **spot patterns** and make inferences on the structure of our data.
- Rather than consider each individual data point as isolated, networks allow us to consider how **data is interrelated**, and how the relationships between nodes effect each other and the overall structure.
- Network techniques allow us to move **beyond pairs** to putting entities and their relationships in a much larger context, showing how they bridge the local and the global. They allow us to consider **groups** within data, particularly in communication networks.
- Networks can point to data subsets worth investigating in more detail: allowing us to move from **close to distant reading**, and in-between.

2.7 Network Basics

This section introduces the fundamental ideas behind network analysis, including its key components, and various network types.

2.7.1 Nodes and Edges

As already mentioned, a network is a graph consisting of nodes and edges (connections). Typically (though not always), the edges are *pairwise,* meaning they run between a pair of nodes. These are often represented visually as points (the nodes) and lines (the edges), like this:

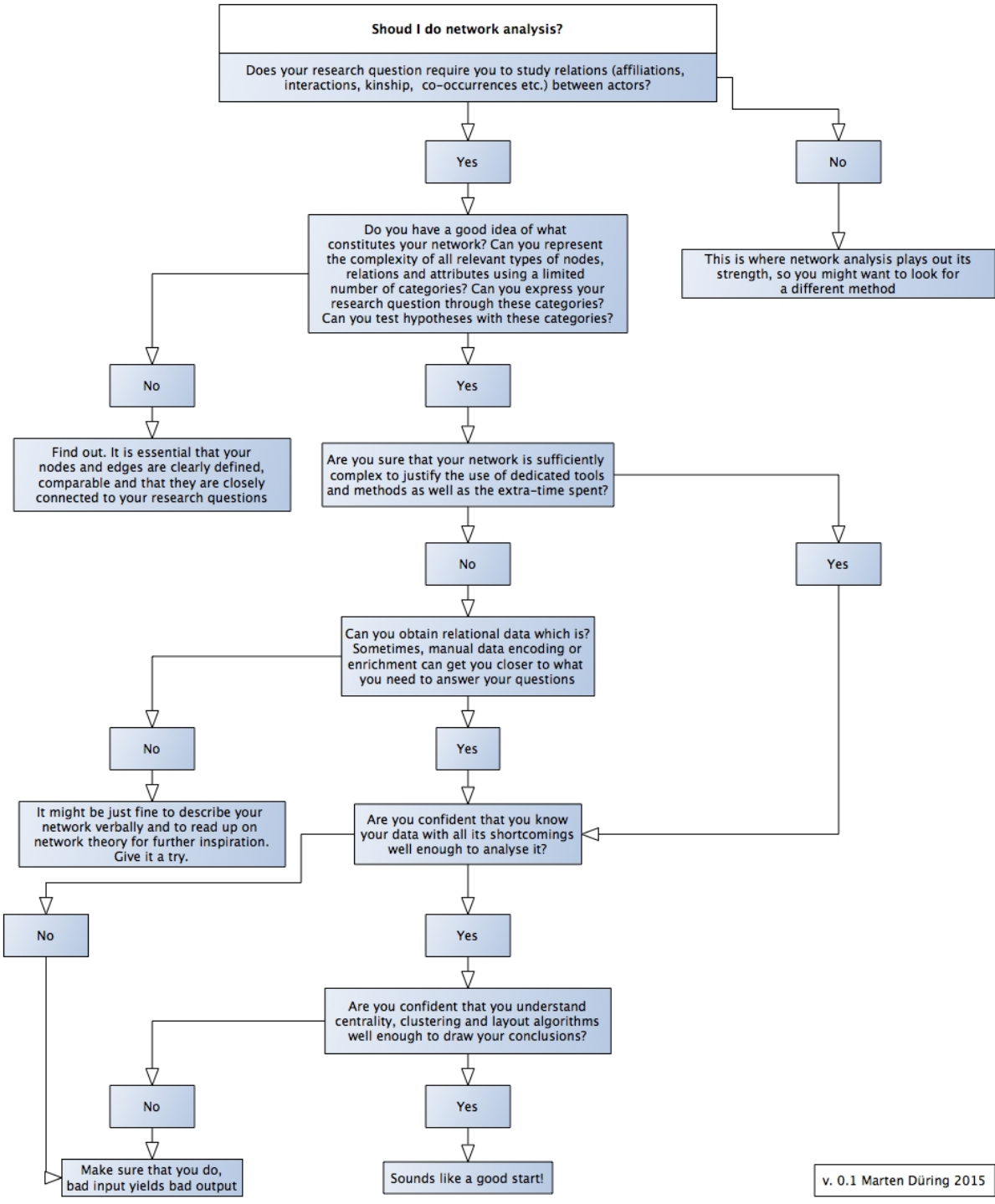
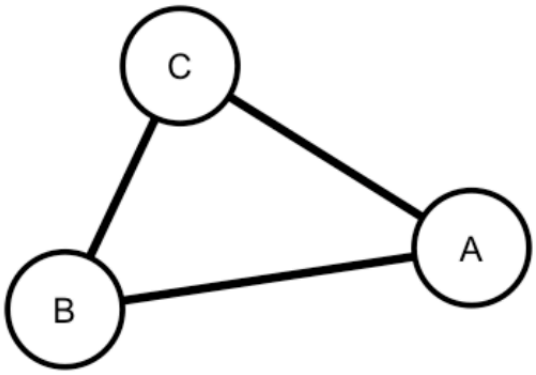
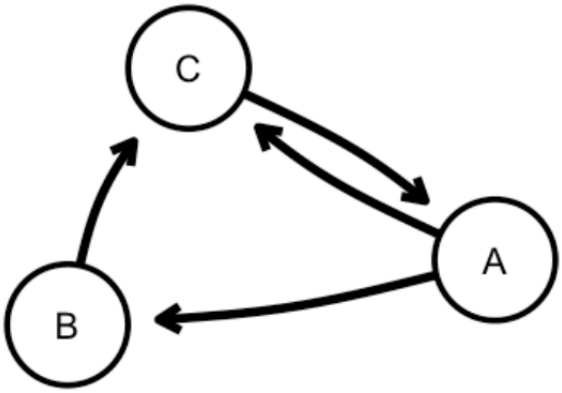


Figure 2.1: Almost anything can be a network, but should you use network analysis? Here is a handy flowchart which might help you decide.



2.7.2 Edge weights

These connections can often have a *weight* attached, for example the number of letters exchanged between two people, or the number of times two actors appear in a scene in a movie or play together. A weight might also be a measurement of similarity or difference between two nodes, such as a linguistic similarity between two books (a network can also be between inanimate objects!), or the distance between two cities on a map. These weights can be used in the calculations.



2.7.3 Edge direction

Edges can also have *directions,* meaning that the incoming and outgoing links are counted separately: for example we might count incoming and outgoing

letters separately.

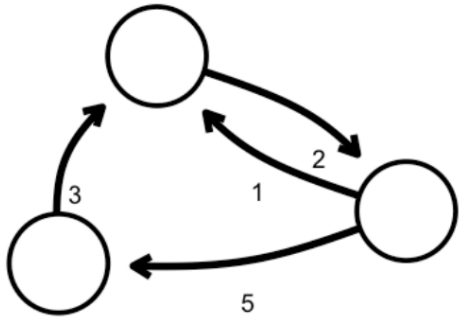
What might this letter count tell us about a relationship in a network? What potential danger is there is reading something from this?

It might be worth considering how the letter network was constructed. Often they are reconstructed from personal letter archives, which tend to be collections of mostly incoming letters to a single person or family. In that case, it's likely the difference between incoming and outgoing letters is not statistically significant, but simpy a product of the method of data collection.

Another typical illustration of directed and undirected networks is the difference between Twitter and Facebook. Facebook friends must be reciprocated - both people have to accept the friend request, meaning that a Facebook network is undirected. Twitter, on the other hand, is a directed network, because an individual can have separate *follower* and *following* numbers.

2.7.4 Weighted and directed

These directed edges can also have separate weights attached to them:



2.8 Network paths

One of the central concepts behind networks is that they allow information to travel along the edges, moving from node to node. In a metaphorical sense, nodes with less 'hops' between them have an easier route to this information, and may be said to be close together or influential on each other. If we add more than three nodes to the network, these paths begin to emerge. A network path is simply a route, travelling along edges, from one node to another in a network. Some network metrics use these paths to estimate structural importance, for example. Paths work differently in *directed* networks: information can only move in the direction of the edges.

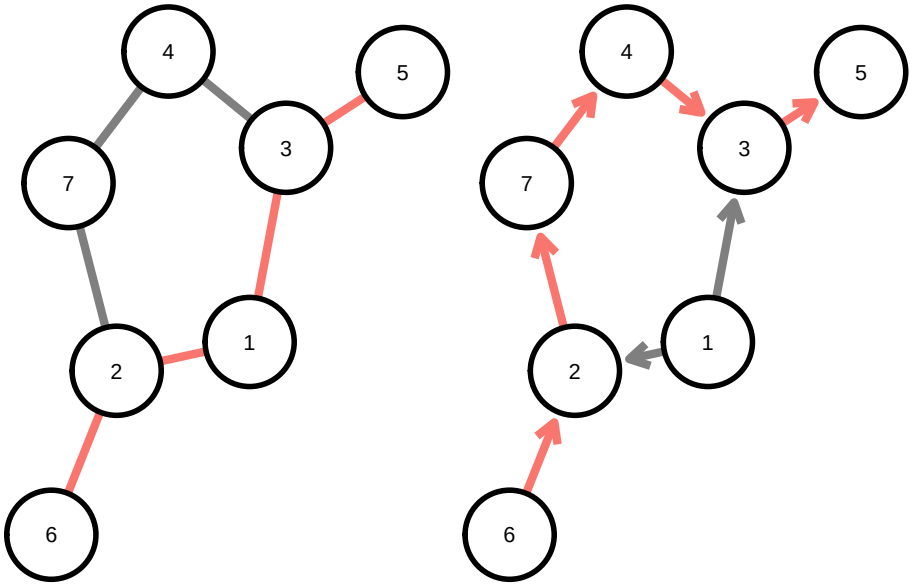


Figure 2.2: Network diagram showing the shortest path between node 5 and node 6. Left-hand network is undirected, meaning the path can travel along any edge. Network on the right is directed, meaning a path only exists in the direction of the edge.

2.9 Different Network Types

2.9.1 Bipartite networks

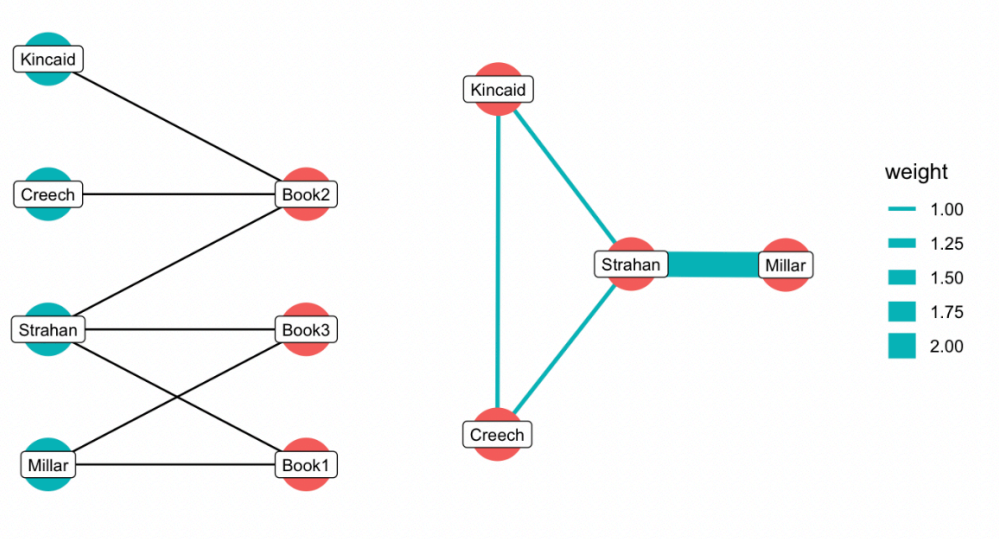
In the above examples, the nodes and connections have been very straightforward: two things of the same *type* (people in a social network for example) connected to each other. These are known as *one-mode* or *unipartite* networks. However, many networks you'll encounter will be of things of different types. These are known as *bipartite networks,* and we'll return to them later in the course. They are very common in humanities data. Some examples include:

- A network of characters in a play connected to scenes they appear in.
- A network of company directors connected to companies
- A network of publishers connected to the books they financed.
- A network of people connected to membership of certain organisations

The diagram below is an illustration of how this looks, with a network of publishers connected to books. Each individual is listed as the publisher of a number of books, which become the two node types, connected as shown in the figure on the left. A publisher can be connected to many books, but the two types of nodes cannot be connected to each other (a publisher obviously can't publisher another publisher…).

Many network measurements and algorithms are designed to be used on regular, *one-mode* networks. Often, when working with bipartite networks, we *collapse* the network into one of the node types, meaning that we directly connect one of the types to each other, based on shared connections to the other type.

In the figure on the left below, we have collapsed the network so that now, publishers are directly connected to each other, based on shared appearances on books. If publishers shared multiple books, this can be added to the new edge as weight.



What would a projected network of the other type (books connected to books) look like?

All three books would be connected to each other, because they all have one publisher in common (Strahan). Book 1 and book 3 would have an edge weight of 2, because they have two publishers in common (Millar and Strahan).

2.9.2 Multigraphs

There can also be multiple *kinds* of edges, in the same network. For example in a network of historical correspondents, you might have 'met in person' as well as 'sent a letter to/from'. If these edges are not merged, these graphs are known as **multigraphs**.

Multigraphs are also known as multi-layer networks: you could imagine each separate set of edges as a separate layer within a network. Conceptually, we might learn interesting things by understanding how the various layers overlap and interact. R and other programming languages have packages to plot and analyse them.

2.9.3 Hypergraphs

The final type of network is even more exotic: in an ordinary network, an edge always connects two nodes together. A network where each edge can connect to any number of nodes is called a **hypergraph**. It's best explained with a diagram: in the figure below, an edge is no longer a line but a coloured area, and each of the points which fall within them are connected nodes. Good

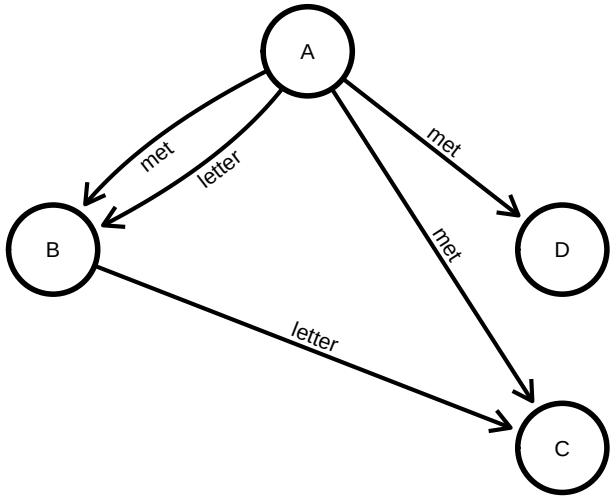


Figure 2.3: A diagram of a multigraph: a network with edges of more than one type.

real-world examples are WhatsApp or Facebook groups: each person can be a member of multiple groups.

What other way could we model the network of Facebook or WhatsApp groups?

These could also be considered bipartite networks (individuals connected to groups).

These networks—multigraphs and hypergraphs—require an additional set of algorithms, and generally off-the-shelf tools have not been developed for them in the same way as exist for normal graphs. However, there are packages available for R which have been developed to deal with them.

2.9.4 Spatial networks

A final type of network is a spatial network: one where the nodes and edges have real-world spatial characteristics. A good example of this is a road network: edges can be represented as geometric lines, and cities and other points of interest as geographic points. These can have historical uses, for example the early modern road network of some of north-western Europe has been mapped by the Viabundus project as a spatial network: using simple network shortest-path algorithms and adding the road length as a weight, the data can be used

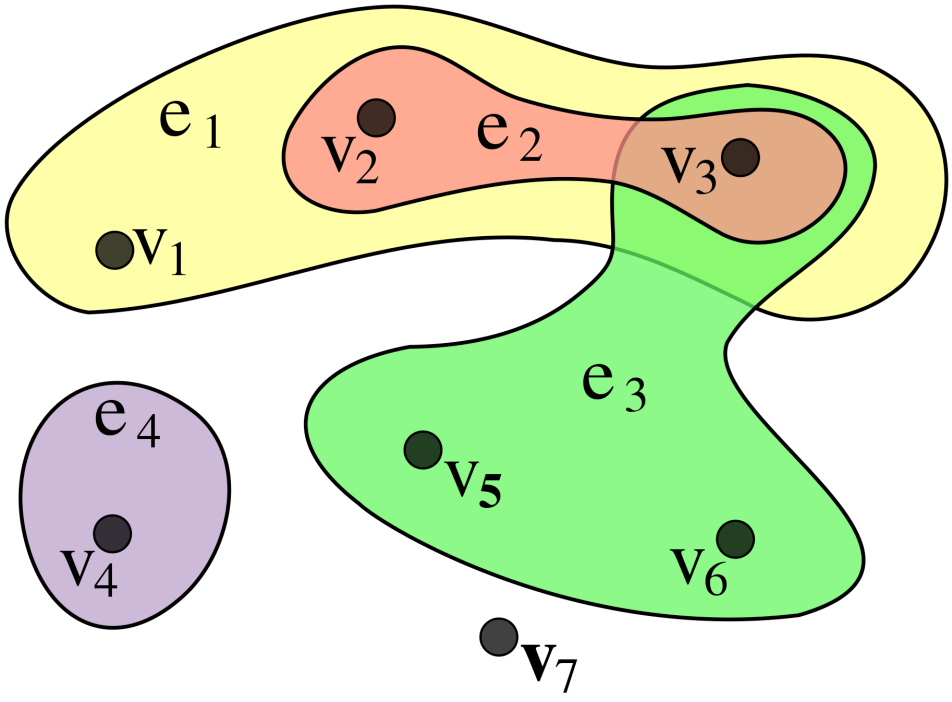


Figure 2.4: (By Hypergraph.svg: Kilom691derivative work: Pgdx (talk) - Hypergraph.svg, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=10687664)

to plot likely itineraries from one point to another.

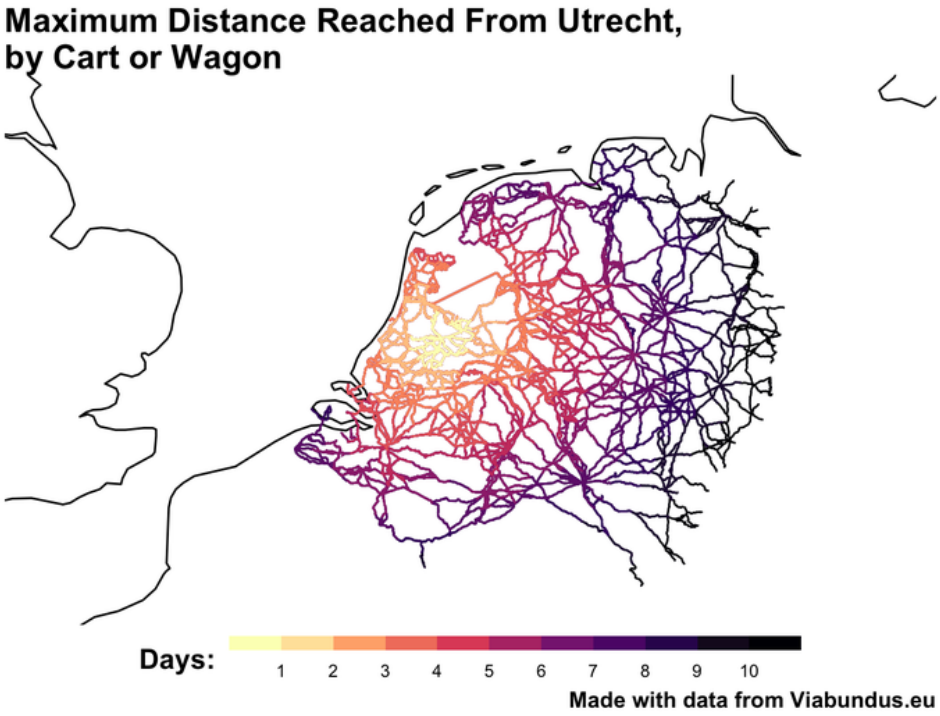


Figure 2.5: A map of the early modern road network made with data from the Viabundus project. Each color represents the 'neighbourhood' of roads which can be reached from Urecht in a single day.

2.10 From Bridges to Social Networks...

In the twentieth century, this graph theory began to be used by the new discipline of sociology now applied to human relationships, to understand the processes behind business, family, and friendship ties. This gave birth to the field of 'social network analysis', which over the past half a century or so has developed a whole range of theories governing the ways networks of people are formed, and what implications this has for the way they act. Human nodes naturally behave quite differently to an island, after all.

In what specific way might the connection between two people have different attributes to a connection between two islands?

Bridges and islands, clearly, don't have any choices as to whom they will connect to: it is determined by their geographic position. So many network analysis

techniques may not be appropriate. But it is a good example of how graph theory should be considered firstly as a set of mathematical tools.

2.10.1 The Strength of Weak Ties

We won't deal with all of these theories here, but will mention a few key ones which are good demonstrations of the way in which graph theory has been applied to social networks, and the kinds of things it has determined.. One of the pioneers of social network analysis was Mark Granovetter. In a 1973 paper, Mark Granovetter argued that the most important ties in a network are often those which connect two separate social groups together. These ties, according to Granovatter, occupy a 'brokerage' position, and can be key in certain situations.

For example, paradoxically, job seekers are more likely to find the most useful leads through their distant acquaintances rather than their close friends…

Why might this be the case?

It's because, according to Granovetter, these distant acquaintances are more likely to be able to provide 'new' information on opportunities: a close friend, on the other hand, will probably have access to the same information as you.

This 'brokerage' position can be deduced mathematically using a metric known as *betweenness centrality* (we'll come back to that in a later class).

2.10.2 It's a Small World

Another important early finding of social network analysis came from a series of experiments by the social psychologist Stanley Milgram. In 1967 Milgram devised an experiment where a series of postcards were mailed out to random people in US cities. These postcards contained basic information about a 'target' person in another, geographically-distant, city. The participants were asked to send the postcard to that person if they knew them, and if not, send it to the contact they thought might be most likely to know that person. The details of each person were recorded on the postcard at every step.

What problems can you imagine with this experiment?

It's worth considering how the experiment may be biased. Are all groups of people equally likely to answer (or have the time and money to carry out) a request from a random postcard? How might this have distorted the findings?

When (or if) a postcard made it to the target person, Milgram could see how many 'hops' it had taken in order to get there. The average number of hops was between five and a half and six: this information was later used to claim that everyone in the US was connected by 'six degrees of separation'. In network

terms, this is known as the 'average path length' of the network. This fact is known as the 'small world' effect.

It is also connected to the 'strength of weak ties' theory by Granovetter. This surprisingly-small number is possible because of the structure of social networks: if you want to reach someone in a distant city, are you more likely to have success if you send it to a close neighbour, or a distant acquaintance who lives there?

2.11 The 'New' Science of Networks

The most common use of networks in academic research much of the 20th century was looking at these small, sociological networks of relationships between people. This changed in the late 1990s, when a group of scientists began to use network research to understand the structures governing many kinds of complex systems, initially using the approach to map out the structure of the World Wide Web. This research showed that many of these complex networks had a similar structure: a small number of nodes with a very large number of connections, known as *hubs,* and a large number of nodes with very few connections. In fact, they argued, the structure followed what is known as a *power-law*: essentially many nodes have a very small number of connections, an exponentially smaller number of nodes have an exponentially larger number of connections, and so forth until a tiny of number of nodes have very many connections.

These researchers, notably Albert Barabasi, argued that this process was guided by *preferential attachment*, meaning that these networks were created because nodes in a network were much more likely to attach themselves to nodes which already had many connections, leading to a 'rich get richer' effect. Perhaps most importantly, they demonstrated empirically that this particular structure could be found across a range of networks, from computers, to people, to biological networks or the structure of neurons in the brain. And they have wide-ranging implications: a scale-free network means it is easy for a disease to spread, because once it reaches a hub node it can easily move throughout the network.

These ideas, published at a time when the internet (and later social networking) moved into the mainstream, drove a huge interest in thinking about the world in a 'networked' way. They spawned a range of popular science and psychology books. Barabasi wrote *Linked,* one of the defining popular science books of the 2000s, and used networks to argue that this scale-free network structure could be used to explain a whole range of human behaviours, from epidemics, to economics, to politics.

2.12 Conclusion

This outline has hopefully got you thinking about this network approach to data. A word of warning: almost everything can be represented as some kind

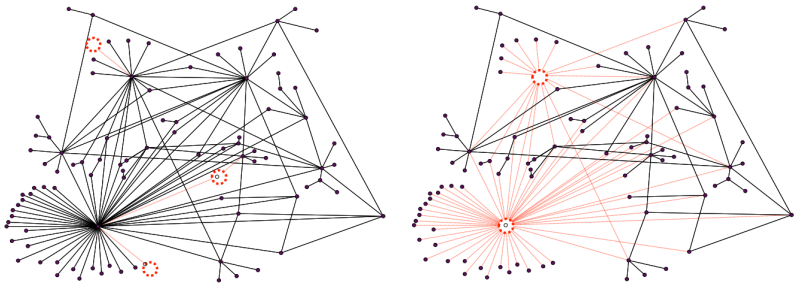


Figure 2.6: These diagrams show how a scale-free network is susceptible to to targeted attack but not to random. Removing any two nodes at random in a network of hubs makes little difference to the overall structure or robustness. However, removing two hub or bridging nodes, as on the right, results in a network in which many of the paths are now severed, meaning the network structure has begun to fall apart. Images from https://cs.brynmawr.edu/Courses/cs380/spring2013/section02/slides/01Introduction.pdf

of network. However, many of the findings from them require some careful thought: do they tell us something interesting, such as why a group of contacts formed in a particular formed in this way, or do they just reflect the data that we have collected or have available? What does a complex 'hairball' network diagram really tell us? Have the claims of *Linked* really helped us to understand business, economics, and the spread of disease?

Between now and the next class, try to consider the ways some of the data you have used in your studies might be thought of as a network, and what benefits (and pitfalls) that approach might bring.

Lastly, remember that a network is a *model:* a set of proposals to explain something about the real world. It is an artificial contruction, or a metaphor, rather than the thing itself. As such, even a very 'accurate' network is likely to be only one of many such models which could be used as an explanation for a particular phenomenon or observation. Throughout this course, you should use your critical tools to keep this in mind, and assess the usefulness of a particular model to your data explorations.

Chapter 3

Week 1, Class 2: Introduction to R and the Tidyverse

3.1 Exercises

This lesson has a corresponding editable notebook containing some exercises. These are stored as a separate notebook in the CSC notebook workspace, with the name `1-2-intro-to-r.Rmd`. You'll need to move a copy to your 'my-work' folder in your CSC notebook Workspace, complete the exercises, and knit them, using the method we learned in the previous class. Once this is done, send a copy of the HTML file to the course leaders.

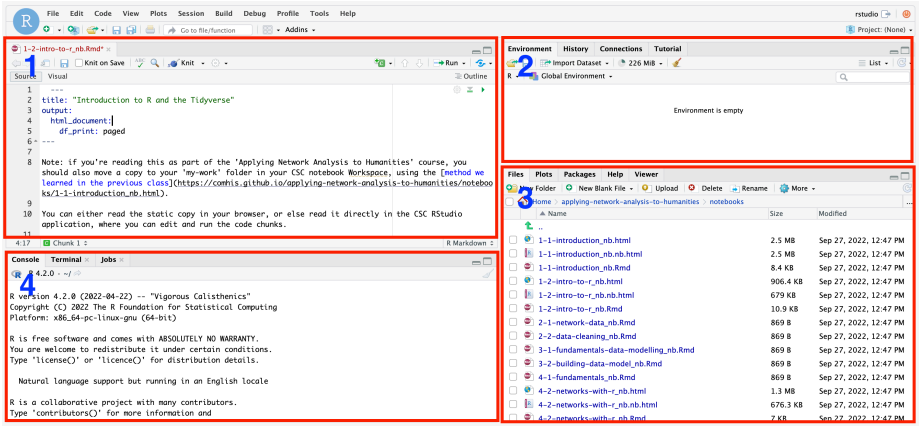
3.2 R and R-Studio

Throughout this course, we'll mostly work on networks and data using the programming lanugage R and a popular extension known as 'the tidyverse'. This will be done using R-Studio, an interface designed to make R easier to work with, known as an IDE.

For this course, the data, files, and interface are all already set up for you in the CSC Notebooks workspace. In most cases, you will want to install R and R-Studio on your local machine. See here for instructions on how to do this.

3.2.1 Logging into CSC notebooks and opening a notebook.

The first thing you should do is log in to CSC Notebooks, and start the RStudio application, [as explained in the previous chapter]. Once you've done this, and opened the relevant notebook, you'll see this screen (I've overlaid squares and numbers to refer to different parts).



R-Studio is divided into four different sections, or *panes*. Each of these also has multiple tabs. Starting from the top-left (numbered 1):

1. The source editor. Here is where you can edit R files such as RMarkdown or scripts.

2. The environment pane will display any objects you create or import here, along with basic information on their type and size.

3. This pane has a number of tabs. The default is files, which will show all the files in the current folder. You can use this to import or export additional files to R-Studio from your local machine.

4. The console allows you to type and execute R commands directly: do this by typing here and pressing return.

All four of these panes are important and worth it's worth exploring more of the buttons and menu items. Throughout this course, you'll complete exercises by using the source editor to edit notebooks. As you execute code in these notebooks, you'll see objects pop into the environment pane. The console can be useful to test code that you don't want to keep in a document. Lastly, getting to know how to use and navigate the directory structure using the files pane is essential.

3.3 Using R

3.3.1 'Base' R.

Commands using R without needing any additional packages are often called 'base' R. Here are some important ones to know:

You can assign a value to an object using `=` or `<-`:

```
x = 1

y <- 4
```

Entering the name of a variable in the console and pressing return will return that value in the console. The same will happen if you enter it in a notebook cell (like here below), and run the cell. This is also true of any R object, such as a dataframe, vector, or list.

```
y
```

```
## [1] 4
```

You can do basic calculations with `+`, `-`, `*` and `/`:

```
x = 1+1

y = 4 - 2

z = x * y

z
```

```
## [1] 4
```

You can compare numbers or variables using `==` (equals), `>` (greater than), `<`, (less than) `!=` (not equal to). These return either `TRUE` or `FALSE`:

```
1 == 1
```

```
## [1] TRUE
```

```
x > y
```

```
## [1] FALSE
```

```
x != z
```

```
## [1] TRUE
```

3.3.2 Basic R data structures

It is worth understanding the main types of data that you'll come across, in your environment window.

A variable is a piece of data stored with a name, which can then be used for various purposes. The simplest of these are single **elements**, such as a number:

```
x = 1

x
```

```
## [1] 1
```

Next is a vector. A vector is a list of **elements**. A vector is created with the command `c()`, with each item in the vector placed between the brackets, and followed by a comma. If your vector is a vector of words, the words need to be in inverted commas or quotation marks.

```
fruit = c("apples", "bananas", "oranges", "apples")
colour = c("green", "yellow", "orange", "red")
amount = c(2,5,10,8)
```

Next are dataframes. These are the spreadsheet-like objects, with rows and columns, which you'll use in most analyses.

You can create a dataframe using the `data.frame()` command. You just need to pass the function each of your vectors, which will become your columns.

We can also use the `glimpse()` or `str()` commands to view some basic information on the dataframe (particularly useful with longer data).

```
fruit_data = data.frame(fruit, colour, amount, stringsAsFactors = FALSE)

glimpse(fruit_data)
```

```
## Rows: 4
## Columns: 3
## $ fruit  <chr> "apples", "bananas", "oranges", "apples"
## $ colour <chr> "green", "yellow", "orange", "red"
## $ amount <dbl> 2, 5, 10, 8
```

3.3.3 Data types

Notice that to the right of the third column, the amount, has `<dbl>`under it, whereas the other two have ‘. That’s because R is treating the third as a number and others as a string of characters. It’s often important to know which data type your data is in: you can’t do arithmetic on characters, for example. R has 6 data types:

- character
- numeric (real or decimal)
- integer
- logical
- complex
- Raw

The most commonly-used ones you’ll come across are `character`, `numeric`, and `logical`. `logical` is data which is either `TRUE` or `FALSE`. In R, all the items in a vector are *coerced* to the same type. So if you try to make a vector with a combination of numbers and strings, the numbers will be converted to strings, as in the example below:

```
fruit = c("apples", 5, "oranges", 3)

glimpse(fruit)
```

```
##  chr [1:4] "apples" "5" "oranges" "3"
```

3.3.4 Installing and loading packages:

R is extended through the use of ‘packages’: pre-made sets of functions, usually with a particular task or theme in mind. To work with networks, for example, we’ll use a set of third-party packages. If you complete the exercises using the CSC cloud notebooks, these are already installed for you in most cases. To install a package, use the command `install.packages()`, and include the package name within quotation marks:

```
install.packages('igraph')
```

To load a package, use the command `library()`. This time, the package name is not within quotation marks

```
library(igraph)
```

3.4 Tidyverse

Most of the work in these notebooks is done using a set of packages developed for R called the 'tidyverse'. These enhance and improve a large range of R functions, with a more intuitive nicer syntax. It's really a bunch of individual packages for sorting, filtering and plotting data frames. They can be divided into a number of different categories.

All these functions work in the same way. The first argument is the thing you want to operate on. This is nearly always a data frame. After come other arguments, which are often specific columns, or certain variables you want to do something with.

```
library(tidyverse)
```

Here are a couple of the most important ones

3.4.1 select(), pull()

`select()` allows you to select columns. You can use names or numbers to pick the columns, and you can use a `-` sign to select everything *but* a given column.

Using the fruit data frame we created above: We can select just the fruit and colour columns:

```
select(fruit_data, fruit, colour)
```

```
##     fruit colour
## 1  apples  green
## 2 bananas yellow
## 3 oranges orange
## 4  apples    red
```

Select everything but the colour column:

```
select(fruit_data, -colour)
```

```
##     fruit amount
## 1  apples      2
## 2 bananas      5
## 3 oranges     10
## 4  apples      8
```

Select the first two columns:

```
select(fruit_data, 1:2)
```

```
##     fruit colour
## 1  apples  green
## 2 bananas yellow
## 3 oranges orange
## 4  apples    red
```

3.4.2 group_by(), tally(), summarise()

The next group of functions group things together and count them. Sounds boring but you would be amazed by how much of data science just seems to be doing those two things in various combinations.

`group_by()` puts rows with the same value in a column of your dataframe into a group. Once they're in a group, you can count them or summarise them by another variable.

First you need to create a new dataframe with the grouped fruit.

```
grouped_fruit = group_by(fruit_data, fruit)
```

Next we use `tally()`. This counts all the instances of each fruit group.

```
tally(grouped_fruit)
```

```
## # A tibble: 3 x 2
##   fruit       n
##   <chr>   <int>
## 1 apples      2
## 2 bananas     1
## 3 oranges     1
```

See? Now the apples are grouped together rather than being two separate rows, and there's a new column called `n`, which contains the result of the count.

If we specify that we want to count by something else, we can add that in as a 'weight', by adding `wt =` as an argument in the function.

```
tally(grouped_fruit, wt = amount)
```

```
## # A tibble: 3 x 2
##   fruit       n
##   <chr>   <dbl>
## 1 apples     10
## 2 bananas     5
## 3 oranges    10
```

That counts the amounts of each fruit, ignoring the colour.

3.4.3 filter()

Another quite obviously useful function. This filters the dataframe based on a condition which you set within the function. The first argument is the data to be filtered. The second is a condition (or multiple condition). The function will return every row where that condition is true.

Just red fruit:

```
filter(fruit_data, colour == 'red')
```

```
##    fruit colour amount
## 1 apples    red      8
```

Just fruit with at least 5 pieces:

```
filter(fruit_data, amount >=5)
```

```
##     fruit colour amount
## 1 bananas yellow      5
## 2 oranges orange     10
## 3  apples    red      8
```

You can also filter with multiple terms by using a vector (as above), and the special command `%in%`:

```
filter(fruit_data, colour %in% c('red', 'green'))
```

```
##    fruit colour amount
## 1 apples  green      2
## 2 apples    red      8
```

3.4.4 slice_max(), slice_min()

These functions return the top or bottom number of rows, ordered by the data in a particular column.

```
fruit_data %>% slice_max(order_by = amount, n = 1)
```

```
##     fruit colour amount
## 1 oranges orange     10
```

```
fruit_data %>% slice_min(order_by = amount, n = 1)
```

```
##    fruit colour amount
## 1 apples  green      2
```

These can also be used with `group_by()`, to give the top rows for each group:

```
fruit_data %>% group_by(fruit) %>% slice_max(order_by = amount, n = 1)
```

```
## # A tibble: 3 x 3
## # Groups:   fruit [3]
##   fruit   colour amount
##   <chr>   <chr>   <dbl>
## 1 apples  red         8
## 2 bananas yellow      5
## 3 oranges orange     10
```

Notice it has kept only one row per fruit type, meaning it has kept only the apple row with the highest amount?

3.4.5 sort(), arrange()

Another useful set of functions, often you want to sort things. The function `arrange()` does this very nicely. You specify the data frame, and the variable you would like to sort by.

```
arrange(fruit_data, amount)
```

```
##     fruit colour amount
## 1  apples  green      2
## 2 bananas yellow      5
## 3  apples    red      8
## 4 oranges orange     10
```

Sorting is ascending by default, but you can specify descending using `desc()`:

```
arrange(fruit_data, desc(amount))
```

```
##     fruit colour amount
## 1 oranges orange     10
## 2  apples    red      8
## 3 bananas yellow      5
## 4  apples  green      2
```

If you 'sort`arrange()` by a list of characters, you'll get alphabetical order:

```
arrange(fruit_data, fruit)
```

```
##     fruit colour amount
## 1  apples  green      2
## 2  apples    red      8
## 3 bananas yellow      5
## 4 oranges orange     10
```

You can sort by multiple things:

```
arrange(fruit_data, fruit, desc(amount))
```

```
##     fruit colour amount
## 1  apples    red      8
## 2  apples  green      2
## 3 bananas yellow      5
## 4 oranges orange     10
```

Notice that now red apples are first.

3.4.6 left_join(), inner_join(), anti_join()

Another set of commands we'll use quite often in this course are the `join()` 'family'. Joins are a very powerful but simple way of selecting certain subsets of data, and adding information from multiple tables together.

Let's make a second table of information giving the delivery day for each fruit type:

```
fruit_type = c('apples', 'bananas','oranges')
weekday = c('Monday', 'Wednesday', 'Friday')

fruit_days = data.frame(fruit_type, weekday, stringsAsFactors = FALSE)

fruit_days
```

```
##   fruit_type   weekday
## 1     apples    Monday
## 2    bananas Wednesday
## 3    oranges    Friday
```

This can be 'joined' to the fruit information, to add the new data on the delivery day, without having to edit the original table (or repeat the information for apples twice). This is done using `left_join`.

Joins need a common `key`, a column which allows the join to match the data tables up. It's important that these are unique (a person's name makes a bad key by itself, for example, because it's likely more than one person will share the same name). Usually, we use codes as the join keys. If the columns containing the join keys have different names (as ours do), specify them using the syntax below:

```
joined_fruit = left_join(fruit_data, fruit_days, by = c("fruit" = "fruit_type"))

joined_fruit
```

```
##     fruit colour amount   weekday
## 1  apples  green      2    Monday
## 2 bananas yellow      5 Wednesday
## 3 oranges orange     10    Friday
## 4  apples    red      8    Monday
```

In this new dataframe, the correct weekday is now listed beside the relevant fruit type.

3.4.7 Piping

Another useful feature of the tidyverse is that you can 'pipe' commands through a bunch of functions, making it easier to follow the logical order of the code. This means that you can do one operation, and pass the result to another operation. The previous dataframe is passed as the first argument of the next function by using the pipe `%>%` command. It works like this:

```
fruit_data %>%
  filter(colour != 'yellow') %>% # remove any yellow colour fruit
  group_by(fruit) %>% # group the fruit by type
  tally(amount) %>% # count each group
  arrange(desc(n)) # arrange in descending order of the count
```

```
## # A tibble: 2 x 2
##   fruit       n
##   <chr>   <dbl>
## 1 apples     10
## 2 oranges    10
```

That code block, written in prose: "take fruit data, remove any yellow colour fruit, count the fruits by type and amount, and arrange in descending order of the total"

3.4.8 Plotting using ggplot()

The tidyverse includes a plotting library called `ggplot2`. To use it, first use the function `ggplot()` and specify the dataset you wish to graph using `data =`. Next, add what is known as a 'geom': a function which tells the package to represent the data using a particular geometric form (such as a bar, or a line). These functions begin with the standard form `geom_`.

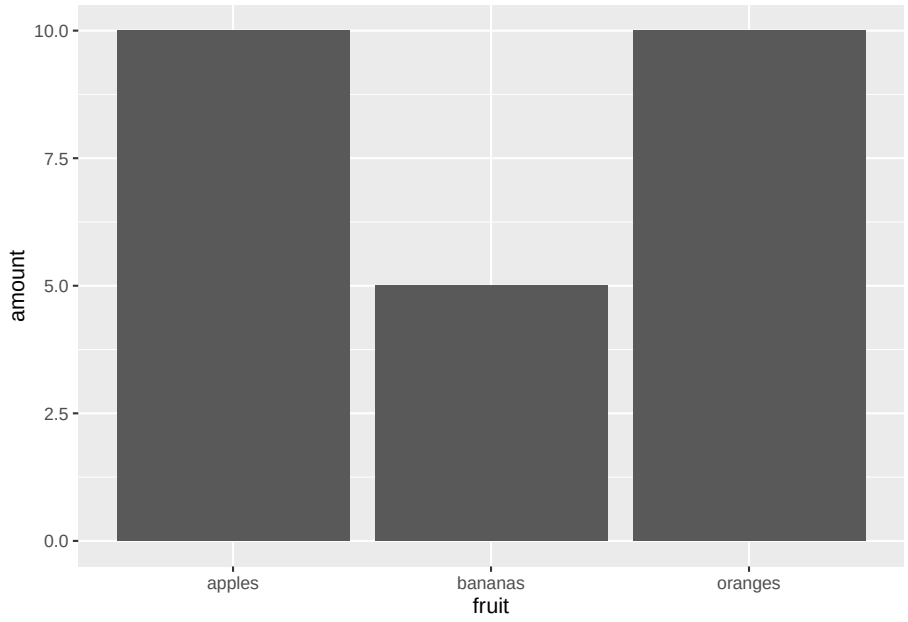
Within this geom, you'll add 'aesthetics', which specify to the package which part of the data needs to be mapped to which particular element of the geom. The most common ones include `x` and `y` for the x and y axes, `color` or `fill` to map colors in your plot to particular data.

`ggplot` is an advanced package with many options and extensions, which cannot be covered here.

Some examples using the fruit data:

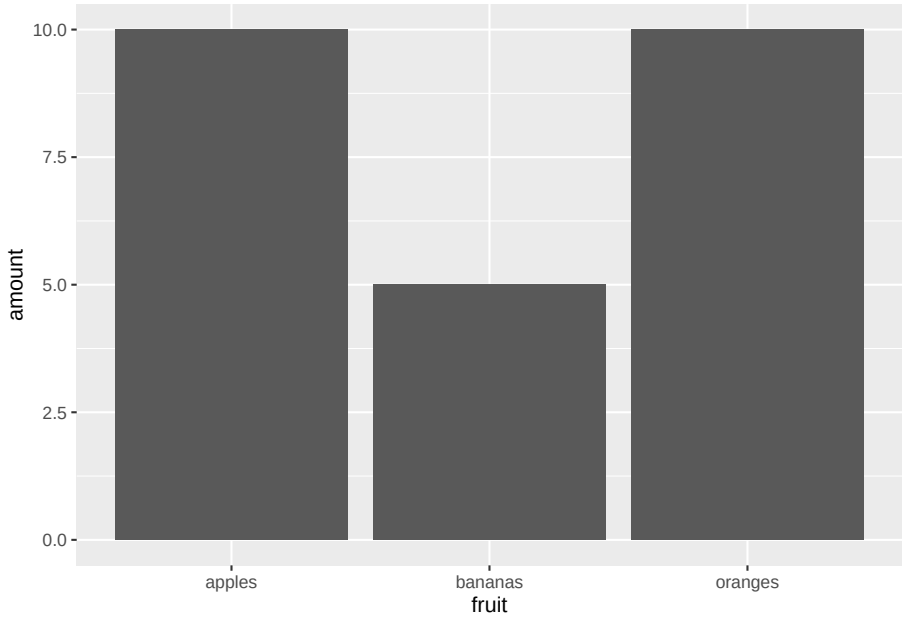
Bar chart of different types of fruit (one each of bananas and oranges, two types of apple)

```
ggplot(data = fruit_data) + geom_col(aes(x = fruit, y = amount))
```



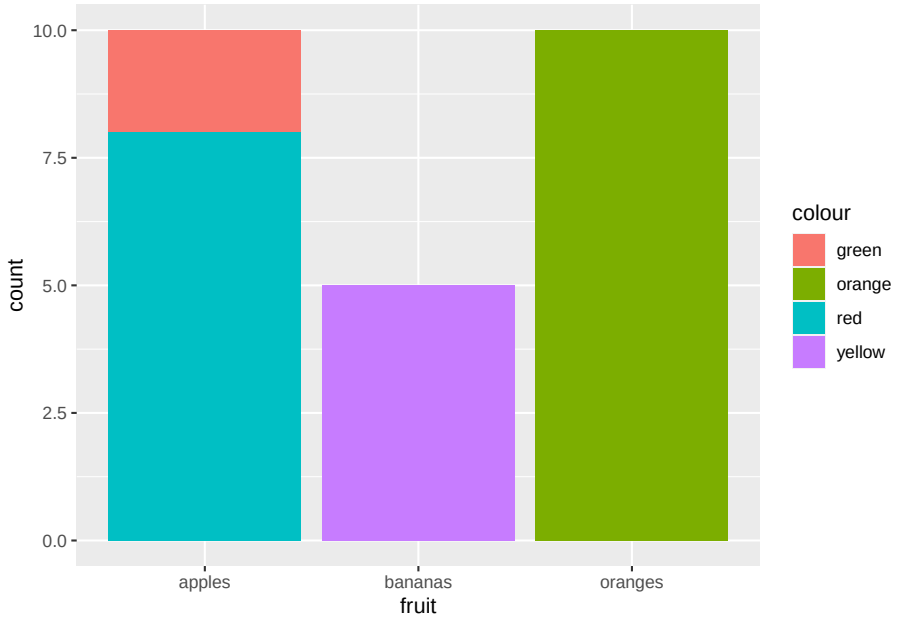
Counting the total amount of fruit:

```
ggplot(fruit_data) + geom_col(aes(x = fruit, y = amount))
```



Charting amounts and fruit colours:

```
ggplot(data = fruit_data) + geom_bar(aes(x = fruit, weight = amount, fill = colour))
```



3.5 Reading in external data

Most of the time, you'll be working with external data sources. These most commonly come in the form of comma separated values (.csv) or tab separated values (.tsv). The tidyverse commands to read these are `read_csv()` and `read_tsv`. You can also use `read_delim()`, and specify the type of delimited using `delim = ','` or `delim = '/t`. The path to the file is given as a string to the argument `file=`.

```
df = read_csv(file = 'sample_network.csv') # Read a .csv file as a network, specify the path to t
```

```
## Rows: 7 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): from, to
## dbl (1): weight
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
df
```

```
## # A tibble: 7 x 3
##   from  to    weight
##   <chr> <chr>  <dbl>
## 1 A     B          5
## 2 A     C          2
## 3 B     C          4
## 4 A     D          1
## 5 A     E          8
## 6 E     D          2
## 7 C     D          3
```

Notice that each column has a data type beside it, either for text or for numbers. This is important if you want to sort or run calculations on the data.

3.6 Further resources

This has been a very quick introduction to R, covering the basics. There are lots of places to learn more, including:

R-studio cheat sheets

The Pirate's Guide to R, a good beginners guide to base R

R for data science, which teaches the tidyverse in detail

Learn how to make a book like this using Bookdown

3.7 Reading For Next Week

For next week, we want you to read an example of network analysis applied specifically to a humanities dataset. The article to read is the following:

Ahnert, Ruth, and Sebastian E. Ahnert. 'Metadata, Surveillance and the Tudor State'. History Workshop Journal, vol. 87, Apr. 2019, pp. 27–51. DOI.org (Crossref), https://doi.org/10.1093/hwj/dby033.

An openly-available pre-publication proof is available through this repository.

We will discuss the article during next Wednesday's class. In preparation, please be prepared to answer the following four questions - we'll choose four people at random to answer one each.

1) What do the authors mean by 'not-reading'?

2) Describe the distribution of the degree scores (degree is the number of connections). What is the significance of this?

3) What are 'bridges' in this context? How do the authors find them in the network?

4) What is the process by which they make the network 'fingerprint'?

As well as this, we'd like you to think about the article as a whole. What does it do well? What can you say about the 'balance' and connections between the humanities and computational parts?

Chapter 4

Week 2, Class 1: Acquiring and Working With Network Data

4.1 Introduction

The topic of the second week is data. We will present the data sets available for the final project of the course, and give some practical examples of the kinds of networks that can be derived from them. We will also start discussing the data wrangling steps that are necessary to transform a given data set to a network data set.

4.2 Presenting Networks as Data

Before jumping into discussing data, we quickly introduce few terms terms that are helpful for understanding how data sets of various sorts can be converted to networks.

Network is made of nodes and edges, so both need to be defined in machine-readable form. Although not necessary in all presentations of network data, node list is a good starting point. It is a table on N rows and C columns, where N is the number of nodes in your data. Each row, then has information about one node. One of the columns is reserved for the identifier of your nodes (a value that defines the node uniquely, think of it as the social security number of the node), the rest are optional. For example, a node list for a social network of humans would have an identifier node at one column of each row, and possible additional information on other rows (name, occupation etc).

Below, we have defined the nodes - persons - in a very small network of academics. In addition to their identifier, the node list includes their occupations.

id	occupation
person_1	humanist
person_2	digital humanist
person_3	computer scientist
person_4	post humanist

The most important element - which is in itself enough to define a network - is edge list. Edge list is a table with at least two columns. The columns can be called "from" and "to", and the values of these columns are unique identifiers of nodes. Each row of the edge list describes one edge in the network, from the node in the "from" column to the node in the "to" column. There can be additional columns in the edge list as well, like columns describing the "strength" of the edge. In the coming weeks, we will teach you how the edge list (and optionally also a node list) can be given as a input to R functions to create a network.

Now we can define the connections (let's say, number of shared publications) between the four researchers with an edge list. It looks like this when printed in R:

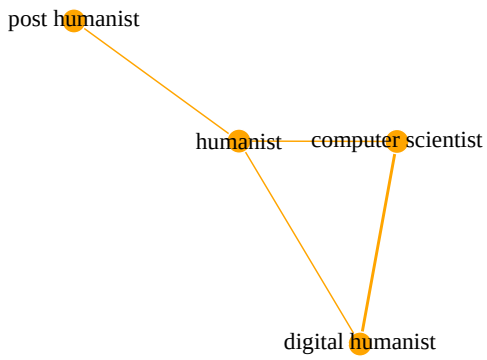
from	to	weight
person_1	person_4	1
person_2	person_3	2
person_1	person_3	1
person_1	person_2	1

Adjacency matrix is a N X N table where element depicts the similarity between two things. The values in the matrix go from 0 (no similarity at all) to any positive real number. In network analysis, adjacency matrix is sometimes used to store similarities between nodes. So if you have a node list that goes from 1. to Nth row (so N nodes in total), then the cell in row i ja and column j of the adjacency matrix tells how similar ith node is to the jth node. If the network is undirected, the cell in row i and column j has the same value as the cell in row j and column i, but this is not necessarily the case with directed network. Adjacency matrix is often the easiest way to store comparisons of nodes to each other, and R makes it easy to convert it to an edge list.

The adjacency matrix corresponding with the preceding edge list looks like this:

	person_1	person_2	person_3	person_4
person_1	0	0	1	1
person_2	0	0	2	0
person_3	1	2	0	0
person_4	1	0	0	0

Finally, the network defined by our example edge and node is presented below. Notice that one of the edges is slightly thicker than the others, can you figure out the reason for that based on the edge list?



Very often, it is the case that the data you are working with is not formatted to node and edge lists from the start, and some wrangling of data is needed to get there. We provide some examples of what the process of turning data to network data can look like in the coding exercises later this week. The questions, however, remain the same. What are the nodes of your data, what are the ids of the nodes, and what are the attributes of the nodes relevant for your analysis. Make a node list of these variables.

The second question is how the edges are defined in your data set. It is often first easier to make an adjacency matrix to compare the nodes to each other, and then turn the adjacency matrix to edge list, but in practice the process of creating an edge list can take many forms. The defining of edges and their strength can also require a great deal of substance understanding and thinking related to the research question.

The following weeks will make these steps more concrete, and the first step towards that is to get to questions about the actual data we will be working with.

4.3 ESTC

The English Short Title Catalogue (ESTC) is an union catalogue of early modern British Print Products. It is the largest collection of metadata about early modern books and pamphlets of the English speaking world. Metadata in general means data about data, and in the case of books (data, in some sense) data

about them. This metadata has hundreds of fields, but the big idea is to connect editions (single print runs of books) of print products to information that is relevant for contextualising them, like publication year, publication place, author publisher etc. The original intention of the ESTC was to help historians and other users of libraries with historical collections to find relevant research material, or to check basic facts about the source material they were working with.

The original ESTC is managed by the British Library and the University of California Riverside, but the version we will be using during the course is a worked on by researchers of the Computational History Group at the University of Helsinki for the last ten-ish years. The ESTC has been extensively harmonised (a process relevant in all data analysis that we will discuss more during next lecture) and enriched, adding information to it both by hand and with computational tools. The end result is a database of editions, authors publishers and other information found from the ESTC, which is much more suited for quantitative research than the original ESTC data.

Why to spend so much time working on data about books, one might ask? To put it simply, a data set like the ESTC allows the studying of all sorts of historical phenomena related to printing from vernacularisation (the process in which the language of power and knowledge shifts to the language spoken by the inhabitants of the country) to political crises, which tend to create upsurges in pamphleteering [Lahti et al., 2019]. Because the ESTC is machine readable, it can be used to study these and other questions with the methods of data analysis, network analysis included.

ESTC has been used to study all sorts of networks of actors, like authors, publishers and printers [Hill et al., 2019]. A very common type of network constructed from the ESTC data connects actors who have been involved in the production of the same edition, based mostly on the information from the imprint of the book. Interestingly from modern perspective, these networks are not always centered and might even exclude the authors. Publishers had significant influence as gatekeepers of what was being actually printed, analysis of their business arrangements are an important context for phenomenon like the Scottish Enlightenment [Sher, 2006].

One option for the data of the final project is a subset of the ESTC, from which a network of publishers/editions/authors or other entities - like networks of editions - is provided by the course organisers.

4.4 Tweetsets and hydrator

Another option for data of the final project is Twitter, and getting the data will be the exercise related to this lesson. Note that there is no additional work in

choosing Twitter data, as everyone will acquire a Twitter data set whether they are going to use it or not.

Twitter data has been studied with network analysis from many perspectives, from the spread of false information [Shahi et al., 2021] to the spread of news about the discovery of Higg's boson [Domenico et al., 2013]. If you are interested of contemporary questions about culture, politics or internet communities, there most likely a (network) perspective to Twitter that you can work on. Networks can be made, e.g. of the quoting, retweeting, following, mentioning or other interaction between users.

However, getting data from Twitter on your own could be non-trivial after only a week's introduction to R, and there are many other caveats as well. Because of Twitter's user policy, only tweet id's can be shared publicly. This means that you will not find Twitter data suitable from network analysis from openly available academic repositories of data. Collecting your own private data set also means plenty of data collection, which would take much time if started from scratch.

Fortunately, there is a service that allows you to get tweet ids related to your topic of interest with an user interface (no coding needed), and a software that then can attach relevant information about the tweets (like the user id, content etc.) to them automatically (no coding needed here either). You can get a customised data set from Twitter data about your topic of interest without any coding at all and without breaking the rules of the platform. The next section guides you through the steps of doing this with the Service Tweetsets and the software Hydrator. Quoting and retweeting networks can be constructed from the data obtained in this way.

4.5 Instructions for homework

Follow this excellent guide from the programming historian to the point where you have a data set from Twitter downloaded to your computer. Stop when you have the hydrated data at your computer, we do processing part differently from the tutorial: https://programminghistorian.org/en/lessons/beginners-guide-to-twitter-data

Some tips to make the process go smoothly:

- make a Twitter account in advance
- Do not create a subset of more than some hundreds of thousands of tweets (you can make a larger data set later if you want to, but hydrating the tweets takes some time).
- The file that is needed to install the Hydrator from Github varies by the operating system of your computer. It is (the part of the file name after .) exe for Windows, deb for ubuntu and dmg for macOS.

- Save the output from the Hydrator as a csv file.

After you have the data set, put it to your working directory in your CSC workspace. There is no coding exercise for this session, but we will use the data set you acquired in the exercises of the next session.

4.6 References

Chapter 5

Week 2, Class 2: Cleaning the Observed and Thinking About the Unobserved Data

5.1 Unharmonised and Missing data, or the Universals of Data Analysis

Although network analysis is the topic of this course, it shares many elements with other approaches to research that apply data, statistics and computation. Perhaps most universal of these similarities are the problems that need to be solved before (network) analysis can begin: those of unharmonised and missing data. In the worst case, jumping over or failing the steps to tackle them can lead to completely faulty results or interpretations. Fortunately, there are things you can do to make your data (or, very least, your interpretation of it) better.

5.2 Unharmonised Data - An Overview

Computers do not have an understanding of relevant and irrelevant differences. This means, that even the smallest of inconsistencies in denoting a thing can have big differences in data analysis. These inconsistencies stem from various sources. There were often various ways to write a name of a city or a person during the early modern period, so often even data that has been "accurately" collected ends up being unharmonised. Both modern and pre-modern data are

also subject to mistakes that happen when the data is recorded or transformed from one format to another. Regular users of research libraries will often find instances where mistakes have been made in the cataloging process, and the users of Twitter that make the contents of a social network data sets might not respect spelling rules to begin with. If these different variants of the same thing are not accounted for, the data will be inflated by pseudo persons, cities, books or other entities, that should be grouped together.

```
## [1] "Some name variants of the City of Aachen according to CERL Thesaurus:"
```

```
## [1] "Aach"            "Achen"           "Aix la Chapelle" "Aix-la-Chapelle"
```

This process of matching and standardising different ways of denoting the same things is called data harmonisation. There are various ways of doing data harmonisation. In fact, it is a field of research on its own right [Christen, 2012], and many algorithms and tools have been developed to help database developers and users to find and match instances of the same thing. Fortunately, it is not necessary to become an expert in all of these techniques to solve most data harmonisation tasks that you will encounter. Here, we introduce two entry-level tools, that will already solve significant number of harmonisation problems, and keep being useful even if you familiarise yourself with more advanced ones.

5.3 Harmonisation Table

The technically simplest solution is to construct a harmonisation table. The table has at least two columns, one for the unharmonised values and other for corresponding harmonised values. The number unharmonised and harmonised columns can also be greater. The pros of this approach are the technical ease and the possibility of doing such matching that would be hard to automate, like converting variations of latin place names to moden standardised names. You can then join the harmonised values of the variable to your table with the unharmonised values with the join-functions of R. However, this approach does not scale well and always requires manual labor. But for a small data set with intricacies of who maps to what (like is often the case with e.g. historical data) manual approach is sometimes the best choice.

```
## [1] "A simple harmonisation table"
```

```
##     name_variants harmonised_name
## 1            Aach          Aachen
## 2           Achen          Aachen
## 3 Aix la Chapelle          Aachen
## 4 Aix-la-Chapelle          Aachen
```

5.4 Regular Expressions and the Stringr Tools

5.4.1 Cleaning Data at Scale

Cleaning data often involves making the same kind of operation over many things. For example, a very common source of spelling variation is that at the very end of a string (e.g "City of London.") there is a white space ("City of London ."), and this same error occurs for tens or even hundreds of entities in our data ("Aachen .", "Paris ." etc.). A slightly more abstract problem of similar kind would be data that is consistent, but at the wrong level of granularity. If we wanted to analyse families instead of persons, but in our data people were named the following way:

```
## [1] "Thomas Hobbes" "Maria Hobbes"  "John Locke"    "Adam Locke"
```

Then we would need to extract the latter part of each name not to count individuals (in this case duplicate variants of the family), and to do this over all names.

Fortunately, there is a collection of R tools that allow us to detect textual patterns of both sorts and more than we demonstrated above, and, after finding the patterns, to apply different kinds of operations over them.

5.4.2 Using Stringr and Regular Experssions

Stringr is a R package [**?**] that is part of the Tidyverse, the collection of R tools that were introduced last week. It is not the only way to do text pattern detection and manipulation with R, but it being part of a bigger system (that we are already using) and covering the basic tools makes mastering it a good starting point for learning data harmonisation by coding.

There are seven main functions in the Str package, and all of them share a component called pattern. The pattern is written as a regular expression. Regular expression is a description of a string of text. It can be as concrete as description of the letter a (the corresponding pattern being "a"), number between 1 and 5 ([1-5]) or a letters-only piece of text at the end of the string ([a-z]+$) These descriptions follow a certain grammar, that we will discuss more soon. Regular expressions can be used to find spelling variations, data in the wrong format and other things that make data unharmonised for your purposes.

The functions of Stringr allow different kinds of operations over the patterns? We consider four of the functions in the String package (for those interested, all of the main 7 are shortly introduced here) that are especially useful for harmonisation and getting to know your data in general. In all cases, the "pattern" parameter is the regular expression, and string is a character vector.

5.4.2.1 str_detect

Returns TRUE or FALSE based on whether it found the pattern from the text. Very useful for exploring your data.

```
example <- c("Thomas Hobbes",c("Maria Hobbes"),c("John Locke"),c("Adam Locke"))
is_hobbes <- str_detect(example,pattern = "Hobbes")
print(is_hobbes)
```

```
## [1]  TRUE  TRUE FALSE FALSE
```

5.4.2.2 str_subset

Returns those instances where there is match with the pattern. Useful for exploring your data.

```
example <- c("Thomas Hobbes",c("Maria Hobbes"),c("John Locke"),c("Adam Locke"))
is_hobbes <- str_subset(example,pattern = "Hobbes")
print(is_hobbes)
```

```
## [1] "Thomas Hobbes" "Maria Hobbes"
```

5.4.2.3 str_replace

Replace the pattern with something else. Note that we used a slightly more complicated regular expression, that replaces the letters only part of the string at the end, and that notices that the first letter of the family name is a capital letter. The text that replaces the matched pattern goes to the replacement parameter.

```
example <- c("Thomas Hobbes",c("Maria Hobbes"),c("John Locke"),c("Adam Locke"))
family_name_replaced <- example %>%  str_replace(.,pattern = " [A-Z]{1}[a-z]+$",replac
print(family_name_replaced)
```

```
## [1] "Thomas insert_family_name_here" "Maria insert_family_name_here"
## [3] "John insert_family_name_here"   "Adam insert_family_name_here"
```

5.4.2.4 str_extract

Returns the first match of the pattern. For example, we could use the function to turn the names in our data set to family names. Notice also that we use the str_replace function to remove the whitespace after exracting the family name.

```
example <- c("Thomas Hobbes",c("Maria Hobbes"),c("John Locke"),c("Adam Locke"))
family_name_extracted <- example %>%  str_extract(.,pattern = " [A-Z]{1}[a-z]+$") %>% str_replace
print(family_name_extracted)
```

```
## [1] "Hobbes" "Hobbes" "Locke"  "Locke"
```

5.4.3 More About Regular Expressions and Stringr in Practice

For effective use of String and similar tools, certain familiarity with the language of regular expressions is needed. We don't aim to teach you all about regular expressions, but enough to get you started. Don't worry about memorising everything by heart, there is an official cheat cheet. It is more important to get a touch to how regular expressions work. The cheat sheet has both the Stringr and regular expressions basics condensed to it, so it is a good idea to bookmark or download to your computer. There are also more comprehensive Stringr tutorials like this one.

Next, we go through some examples that make use of different elements of the "grammar" of regular expressions. It might be a good idea to also look at the str_function examples again after these examples. We use str_replace, because it makes it easy to demonstrate what was the detected pattern (that is then replaced).

```
# Matching a singe element, the letter T
example <- c("Thomas Hobbes",c("Maria Hobbes"),c("John Locke"),c("Adam Locke"))
example_replaced <- str_replace(example,pattern = "T",replacement = " match was here ")
print(example_replaced)
```

5.4.3.0.1 Trivial patterns

```
## [1] " match was here homas Hobbes" "Maria Hobbes"
## [3] "John Locke"                   "Adam Locke"
```

```
# Matching collection of elements, e.g a name
example <- c("Thomas Hobbes",c("Maria Hobbes"),c("John Locke"),c("Adam Locke"))
example_replaced <- str_replace(example,pattern = "Thomas",replacement = " match was here ")
print(example_replaced)
```

```
## [1] " match was here  Hobbes" "Maria Hobbes"
## [3] "John Locke"              "Adam Locke"
```

5.4.3.0.2 Alternates The pattern might allow for different options, which is what the examples in this subsection are about.

```
# Matching either or (one of the two names in this case)
example <- c("Thomas Hobbes",c("Maria Hobbes"),c("John Locke"),c("Adam Locke"))
example_replaced <- str_replace(example,pattern = "Thomas|Maria",replacement = " match
print(example_replaced)
```

```
## [1] " match was here  Hobbes" " match was here  Hobbes"
## [3] "John Locke"              "Adam Locke"
```

```
# Matching from a range, here they are numbers from 0 to 9, but they could also be e.g
# In general, the brackets [] allow for alternative matches.
example <- c("Thomas Hobbes 1","Maria Hobbes 7","John Locke","Adam Locke")
example_replaced <- str_replace(example,pattern = "[1-9]",replacement = " match was her
print(example_replaced)
```

```
## [1] "Thomas Hobbes  match was here " "Maria Hobbes  match was here "
## [3] "John Locke"                     "Adam Locke"
```

```
# Matching from a collection of values, notice that we once again use brackets [] for
#without range.
example <- c("T","M","L")
example_replaced <- str_replace(example,pattern = "[TM]",replacement = " match was here
print(example_replaced)
```

```
## [1] " match was here " " match was here " "L"
```

```
# The alternatives can also be used to replace anything but what is specified within t
example <- c("A","7")
example_replaced <- str_replace(example,pattern = "[^1-9]",replacement = " match was he
print(example_replaced)
```

```
## [1] " match was here " "7"
```

5.4.3.1 Anchoring

The pattern can be specified to relate to certain parts of the string, namely the start or the end.

```
# The match can be anchored to the start of the string with ^
example <- c("this is not a number 9","7 is a number")
example_replaced <- str_replace(example,pattern = "^[1-9]",replacement = " match was here ")
print(example_replaced)
```

```
## [1] "this is not a number 9"       " match was here  is a number"
```

```
# The match can be anchored to the end of the string with $
example <- c("this is not a number 9","7 is a number")
example_replaced <- str_replace(example,pattern = "[1-9]$",replacement = " match was here ")
print(example_replaced)
```

```
## [1] "this is not a number  match was here "
## [2] "7 is a number"
```

5.4.3.2 Grouping

Elements in the pattern can be grouped with parentheses.

```
#Search for either Maria or Thomas, and Hobbes after that
example <- c("Thomas Hobbes","Maria Hobbes","John Locke","Adam Locke")
example_replaced <- str_replace(example,pattern = "(Thomas|Maria) Hobbes",replacement = " match w
print(example_replaced)
```

```
## [1] " match was here " " match was here " "John Locke"       "Adam Locke"
```

5.4.3.3 Quantification

The pattern can include a specification about the number of times something occurs in the string, this subsection provides examples of that.

```
#Replace Thomas when it occurs from two to three times.
example <- c("Thomas Hobbes","Thomas Thomas Hobbes","Thomas Thomas Thomas Hobbes")
example_replaced <- str_replace(example,pattern = "(Thomas ){2,3}",replacement = " match was here
print(example_replaced)
```

```
## [1] "Thomas Hobbes"          " match was here Hobbes" " match was here Hobbes"
```

```
#? * and + can also be used to control how many times something need to occur in a match. Here we
#+, that means that the pattern needs to be found at least once, but possibly more times.
example <- c("Thomas Hobbes","Thomas Thomas Hobbes","Thomas Thomas Thomas Hobbes")
example_replaced <- str_replace(example,pattern = "(Thomas )+",replacement = " match was here ")
print(example_replaced)
```

```
## [1] " match was here Hobbes" " match was here Hobbes" " match was here Hobbes"
```

5.4.3.4 Look arounds

```
# Match can be conditioned to the surrounding elements. Here, the regular expression l
#More ways to do this on the cheat sheet.
example <-  c("Thomas Hobbes","Maria Hobbes","John Locke","Adam Locke")
example_replaced <- str_replace(example,pattern = "Maria(?= Hobbes)",replacement = " ma
print(example_replaced)
```

```
## [1] "Thomas Hobbes"           " match was here  Hobbes"
## [3] "John Locke"              "Adam Locke"
```

5.4.3.5 Character matching

There are some characters or character combinations with special meaning in regular expressions. We demonstrate some examples, the cheat sheet has a more comprehensive list.

```
# . matches any character except a new line. Here, we use it with + to allow any strin
example <- c("Thomas Hobbes","Maria Hobbes","John Locke","Adam Locke")
example_replaced <- str_replace(example,pattern = ".+",replacement = " match was here "
print(example_replaced)
```

```
## [1] " match was here " " match was here " " match was here " " match was here "
```

```
# . is an example of a character, that needs to be written as \\., if the aims is to m
#meaning, in this case any character, of that character in regular expressions. The sa
example <- c("Thomas Hobbes.","Maria Hobbes","John Locke","Adam Locke")
example_replaced <- str_replace(example,pattern = "\\.",replacement = " match was here
print(example_replaced)
```

```
## [1] "Thomas Hobbes match was here " "Maria Hobbes"
## [3] "John Locke"                    "Adam Locke"
```

```
# [:digit:] is an example of command that matches a distinct set of characters, in thi
example <- c("Thomas Hobbes 12","Maria Hobbes","John Locke 7","Adam Locke")
example_replaced <- str_replace(example,pattern = "[:digit:]",replacement = " match was
print(example_replaced)
```

```
## [1] "Thomas Hobbes  match was here 2" "Maria Hobbes"
## [3] "John Locke  match was here "     "Adam Locke"
```

5.4.3.6 Combining elements

Some of the examples have already used many of the properties of regular expressions at once, and this is often extremely helpful, as it makes regular expressions much more expressive. The last example uses many of the things we have dicussed at the same time.

```
# Task, match the people who were born at the 17th century and have a de at their names
example <- c("Baruch de Spinoza 1632-1677","John Locke 1632-1704","François Fénelon 1651-1715")
example_replaced <- str_replace(example,pattern = ".* de .* 16[0-9]{2}-1[0-9]{3}",replacement = '
print(example_replaced)
```

```
## [1] " match was here "           "John Locke 1632-1704"
## [3] "François Fénelon 1651-1715"
```

Here we first allowed string of any length (first name), then we required de, then allowed for another string of any length or type, after which we required 16 followed by two other numbers (birth year in the 17th century), followed by - and the year of death. And this is by no means as expressive as a regular expression can get.

5.4.4 Regular Expressions and Stringr - a Summary

Now, you should have a some sort of understanding in how regular expressions can be used in the manipulation of strings. We will demonstrate the practical usefulness of these tools in the exercise set of this week, where we will use some of these tools to clean the Twitter data set you obtained last time.

5.5 Other Solutions To Harmonisation of Data

It should also be mentioned, that data harmonisation can be done without building a code workflow. Openrefine is an example of a harmonisation tool with an user interface. We do not prohibit the use of such tools for e.g. harmonising your data set for the final project, and it can a reasonable choice in many instances. However, it will serve you in the long run to get familiar at least with regular expressions, and possibly more. Building workflows of data harmonisation often takes as much or more time than the actual analysis in DH research, so building competence in these skills is useful.

5.6 Missing data

Another major theme is data that is lacking. It is very common in the humanities and the social sciences, that some of the data that should be part of the data set

is not actually there. This phenomenon comes in two major forms, as the data can be either completely or partially missing. Here, we take a quick overview to these two types of missingness, and to some of the ways the problems caused by them can be alleviated (or at least understood).

Partially missing data is a known unknown, you can observe just by looking at the data set. In R, you might have seen value NA in a cell of a table, where you would have expected a character or numeric value. In other words, partially missing data refers to instances where we are missing some of the values of variables for an observation. The reasons for missing data vary. People might not answer all the questions in a survey, the piece of data might be unavailable (e.g. books imprint might lack the name of the publisher), query used to get the data might miss values in the wrong format, or any other reason. The end result is still the same: a collection of missing elements in your data.

```
##            name  occupation
## 1 Thomas Hobbes philosopher
## 2  Steven Steel        <NA>
```

There are two reasons why the partially missing data matters. One is purely technical: many operations can not be performed over missing values. For example, lets assume that you want to take the average number of pages for three books, but for one of the books, the number of pages is missing. As "missing" or NA is not a number, you can't sum it with numbers, hence you can not take the average. Most of data analysis in the humanities and the social sciences would be impossible if problems like this stopped as from taking averages or from doing other sorts of analyses that can't tolerate missing values. One common solution is simply to ignore the missing values as if they did not exist in the data. In the example, this would mean that the average is taken over the number of pages of the two books with page number information available.

The other reason for why partially missing data is problematic, is that missing values might not be missing at random. For example, think of a survey that asks peoples opinions about a polarising topic: people who are not answering might do this because they think their opinion is controversial, and they don't trust the anonymity of the survey. This is one of the reasons why the simplest solution - just ignoring the missing values, more formally omission - is sometimes dangerous. Other solutions include imputation (filling of missing information with an estimate) and analyses that are designed to be less affected by missing data.

We do not teach imputation or analyses robust to missing data on this course, but we emphasize that they are motivated by a question anyone analysing their data should consider: are the missing values Missing Completely at Random (MCAR), or does data missing correlate with the properties of the data? If yes, is it possible that this affects the results of a (network) analysis done with the data? There is no out-of-the-box solutions to these problems, which makes

(among other reasons) the expertise that scholars from the humanities have about their subject very valuable to data analysis as well, as they often know about such potential pitfalls in the sources they are working with (even if they don't articulate these things in terms of statistics).

Completely missing data makes the unknown unknowns related to your data, those instances that are not there at all. Like partially missing data, completely missing data can have various causes. For example, some information in social media can be hidden, meaning that queries or scraping will miss it. Historians, archeologist and others working with pre-modern societies face missing data all the time, as many books, letters and other artifacts known to have existed have been completely lost.

Completely missing data does not create the same kind of technical problems as partially missing data, as they are not present in your database in any form. However, their implications for the meaningfulness and interpretation of, lets say, your network, can be as or even more severe than those cases that are partially missing. Entire observations can also be missing at random, or it might be related to the properties of the observations themselves. For example, there is strong evidence that the ESTC is heavily biased towards certain kinds of items before 1641, covering them extensively (e.g. bibles) while missing most examples of other types of items [Hill, 2016]. In some instances, it is possible to evaluate how much and what is missing, like with the ESTC, that can be compared to other sources. However, this is not always the case.

Fortunately, it is always possible to think what your (network) data needs to be to serve your research question. For example, is it necessary that your network covers all the persons and connections that could be involved, or is a good proportion or a sample (representative set of data from the population of your interest) enough? Do the relevant results change radically if you drop some of the network edges away at random to test how sensitive the results are to some variation of the data set? If so, is it possible that your results are very sensitive to missing data as well. Very often, the research process and closer analysis of the data reveals that the original question required too much from the data, or that it is much more suited to analyse something else that was revealed during the research process. But this is a normal part of research, and should not be feared.

5.7 For the Next Week

The treatment of missing data can lead to dramatic changes in results. The article to read for the next week is a critique of another article, published in a very prestigious journal. You should be able to access the article through Helka. The article is not the easiest, but try to understand the main ideas. We will discuss the article next week. Especially, everyone should be prepared to answer these four questions in front of the class. Who answers will be selected

randomly during the lecture (one person per question, no more than one question per student). The questions are:

1) Q: What is the forward bias in the Seshat data according to the authors?
2) Q: How the missing information about moralising gods was handled in the R script of the original article?
3) Q: How the predicted emergence of moralising gods differ between the revised regressions and the original Seshat data set?
4) Q: What has happened to the original article?

5.8 References

Chapter 6

Week 3, Class 1: Relational Data Part 1

6.1 Managing Data

The aim of this week is to teach skills that are useful in managing large collections of data. These skills are not particular to network analysis, but this makes them even more important, as a digital humanist (or researcher of any other sort with data in their hands) can not run away from questions of data management just by changing methods. On the other hand, one element of these skills is to see your data as a network.

What this managing means in practice? One way to illuminate this is to demonstrate what the absence of it looks like with (pseudo) data.

```
##   sender receiver occupation_sender occupation_receiver receiver_city
## 1  Dewey      Hue       philosopher          theologian     Frankfurt
## 2    Hue    Louie        theologian        lens grinder        Aachen
## 3  Louie    Dewey      lens grinder         philosopher     Frankfurt
##   receiver_location                                letter    letter_topic
## 1              Main               Absence of data management            data
## 2         Westfalen     Absense of tenures for lens grinders         economy
## 3              Oder Absence of pseudo name for letter titles meta commentary
```

As we can see, the table is a collection of at least of three sort of things, persons, cities and letters. But it is not a good a good way to represent the information in it for various reasons. First of all, it is confusing. Instead of each row representing information about a single thing (e.g information about an individual), each row has variables related to the place, the sender, the receiver and the

letter itself. If we were to add even more variables, the data set would become even more unwieldy.

Another problem is, that the data is a time bomb waiting for errors to happen. Senders, receivers and cities are identified by their names, but it is very likely that - if this data set was to grow larger - at some point distinct persons, cities and letters will share the same name. This has already happened with one of the cities, Frankfurt, as the data set has both Frankfurt am Main and Frankfurt am Oder in it. Without another way to handle identification, this data will most likely result in faulty analysis if it expands to more than three row. These problems that can already be seen with three participants of an imaginary letter exchange network become even more apparent when the data grows into realistic size.

The solution that we discuss is to transform your data into relational data, collection of linked data sets, each of which stores entities of one type. To do that, we need data modeling and primary keys. Data modeling might sound complicated, but for our purposes it is not. At the level of a conceptual data model, it is about identifying entities from your data set. At the level of logical data modeling, it is about adding more concrete elements to the conceptual data model. Primary keys identify entities uniquely, and make the data relational. The rest of this chapter is about making these steps and concepts that are useful for creating relational data (as well as the concept itself) more concrete.

6.2 Relational data

Instead of starting with a definition, we demonstrate what the data of the previous example would look like as relational data, after the data modeling and addition of primary keys and such. The data set above will be transformed into three separate tables:

```
## [1] "A table of persons:"
```

```
##   person_id person_name person_occupation
## 1    p_id_1         Hue        theologian
## 2    p_id_2       Louie      lens grinder
## 3    p_id_3       Dewey       philosopher
```

```
## [1] "A table of cities:"
```

```
##   city_id city_name city_location
## 1  c_id_1 Frankfurt          Main
## 2  c_id_2    Aachen     Westfalen
## 3  c_id_3 Frankfurt          Oder
```

```
## [1] "A table of letters:"

##   letter_id                                letter_name    letter_topic sender_id
## 1    l_id_1               Absence of data management            data    p_id_3
## 2    l_id_2     Absense of tenures for lens grinders         economy    p_id_1
## 3    l_id_3 Absence of pseudo name for letter titles meta commentary    p_id_2
##   receiver_id city_in_which_received_id
## 1      p_id_1                    c_id_1
## 2      p_id_2                    c_id_2
## 3      p_id_3                    c_id_3
```

It is the same data, but now stored in three separate tables of different kinds of entities, people, locations and letters. The connections between e.g. letters and their senders are now stored to the columns of these tables. For example, letters connect to receivers and senders by the columns "sender_id" and "receiver_id" and to cities by the column "city_in_which_received_id". It is also worth noting that the entities now have unique identifiers instead of being recognised by their names. When reading the next sections, it might be useful to do it by thinking how the steps help us to get closer to this point.

In general, the idea of relational data is to store data into a collection of separate tables connected to each other by unique identifiers. Each table stores information about entities of one type. For example, our relational version of the correspondence data has three tables. One for people, one for cities and one for letters. In each of these tables, one row corresponds with one and only one entity of that type, and it is identified by the unique identifier, also known as primary key, which is defined by one or more columns. In our case, the primary keys are the person, city and letter ids in the tables of people, cities and letters.

However, although unique identifiers are unique in the primary key column(s) of each table, they can appear in other columns more, equally or less often, and this makes relational data relational. For example, both letters 1 and 2 (l_id_1 and l_id_2) connect to person 1 (p_id_1, Hue), as he is a receiver of one letter and sender of another. This means, that by linking the table of people to the table of letters by matching the person_id with sender or receiver ids, we can connect letters to their senders and receivers. Connecting the tables is possible with the join functions that were discussed during the first week, and we are going to return to them later this week. The same applies to the connections between letters and cities, as those are also recoverable by linking the receiver_city_ids to city_ids.

6.3 Conceptual Data Model

Conceptual data model aims to divide data to different types of entities, and to elaborate if they are related to other entity types in the data. Here by entities

we mean the main categories of "things" in your data, so the conceptual data model can be thought of as a map that describes the structure of your data at a very general level. Conceptual data model is necessary (at least implicitly) to split your data into a collection of related tables, as doing that requires an understanding of what we are splitting to what.

Let's take a look at what a conceptual data model could look like for the example correspondence (pseudo) data set that we discussed at the start of this chapter. The data set is made up of of eight columns, so the immediate idea could be that the conceptual data model involves eight types of entities. However, when we start thinking about the data more closely, the number of distinct type of "things" in the data is much smaller. For example, both sender and receiver are people, and the occupation is an attribute that describes the person. Person, then, would be a good candidate for an entity. With similar reasoning, both the city of the receiver and the letter would make for good entities.

We are already halfway through making our conceptual data model, as we have identified the main things (entities) in the data set. But we still have to consider what the relationships between these entities to complete our conceptual data model. Letters have senders and receivers, and those are people, other types of entities. In the conceptual data model, we would mark this by connecting letters to people. Letters also connect to the places, in our example to the place in which they were received (although place from which they were sent might be more realistic). As cities are also entities in our conceptual data model, the cities and letters are also connected in our conceptual data model.

If drawn, this conceptual data model would look something like this. The details of visualisation can vary, but the important thing is that the entities and their connections can be read from the conceptual data model.

6.4 Logical Data Model

The next step in data modeling is to add details to the overview given by the conceptual data model. Logical data model always includes the attributes of entities listed under them. So, in our case, the logical data model would not only divide entities to people, cities and letters, but also specify that all of them have a name attribute, people have occupations and letters have a city of reception, sender and receiver.

Logical data model is also a good step to specify what is the unique identifier for each entity type, and how these unique identifiers link between the entities. For example, a logical data model made of our example data set would depict how the primary key of the person table (person_id) would connect with attributes sender (sender_id) and receiver (receiver_id) of the letters. As the logical data model already lists all the attributes, primary keys and relations between the entities, it can already be used to "see" what a data set would look like in

relational form. Logical data model (made for data set that is very close to our example) looks like this in practice. Once again, the details of visualisation are not important, but the listing of attributes,identifiers and relationships between the entities (People,Letter,Cities).

The logical data model is already a good enough conceptualisation of your data, that you can turn it into relational format. We will demonstrate this next, and we will practice it more in the exercises of this session.

6.5 Relational Data in Practice

Following the logical data model for our example data set, we see that three tables are needed: one for people, another for cities and third for letters. We will also define unique identifiers for all of these entity, as the names are not unique enough to ensure that we can always disambiguate between entities of the same type. The original table with the unique identifiers added looks like this:

```
##   sender receiver occupation_sender occupation_receiver receiver_city
## 1  Dewey      Hue       philosopher          theologian     Frankfurt
## 2    Hue    Louie         theologian        lens grinder        Aachen
## 3  Louie    Dewey      lens grinder         philosopher     Frankfurt
##   receiver_location                                  letter    letter_topic
## 1              Main               Absence of data management            data
## 2         Westfalen     Absense of tenures for lens grinders         economy
## 3              Oder Absence of pseudo name for letter titles meta commentary
##   receiver_id city_id letter_id sender_id
## 1      p_id_1  c_id_1    l_id_1    p_id_3
## 2      p_id_2  c_id_2    l_id_2    p_id_1
## 3      p_id_3  c_id_3    l_id_3    p_id_2
```

While adding the identifiers ensured that our Frankfurts will not get mixed up, the table has become even more unwieldy, as we now have 12 columns that connect to three different types of entities.

The next step of splitting the data in to the tables of the entities would normally require more data wrangling. Here we work with the simplifying assumption that each identifier maps only to one name, occupation etc. In reality, it would often take more effort to split a single data set to relational data. We will also assume that each person is both a receiver and sender of a letter, so we don't have to create a new table of both senders and receivers.

```
##                                   letter    letter_topic receiver_id city_id
## 1 Absence of data management            data      p_id_1  c_id_1
```

```
## 2     Absense of tenures for lens grinders         economy      p_id_2  c_id_2
## 3 Absence of pseudo name for letter titles meta commentary      p_id_3  c_id_3
##   letter_id sender_id
## 1    l_id_1    p_id_3
## 2    l_id_2    p_id_1
## 3    l_id_3    p_id_2
```

```
##   person_name person_occupation person_id
## 1         Hue        theologian    p_id_1
## 2       Louie      lens grinder    p_id_2
## 3       Dewey       philosopher    p_id_3
```

```
##   city_name city_location city_id
## 1 Frankfurt          Main  c_id_1
## 2    Aachen     Westfalen  c_id_2
## 3 Frankfurt          Oder  c_id_3
```

We have now walked through a simple example of recognising entities from a data set and mapping the attributes and relationships to and between these entities. These steps required conceptual and logical data modeling. Finally we transformed the data set to relational data by following the logical data model.

Very often, relational data is stored in to databases, and queried with SQL. However, while useful, databases and SQL are not necessary to work efficiently with relational data. Next chapter of this week will return to the join_operations (already introduced during the first week) and other functions in the tidyverse, that are needed to work with our relational data in practice.

6.6 Data Modeling and Networks

You might have noticed that the example visualisations of conceptual and logical data models seemed familiar not only in terms of content, but also structurally. This is not a coincidence, as the structure is that of a graph. Data models are like networks in the sense that they depict relationships. In both logical and conceptual data models, nodes can be thought of as entity types and edges as relationships between different types of entities. Logical data model can be thought of as adding attributes to nodes (like we discussed when we introduced the node list) and additional information about the edges. In fact, logical data model is already a quite complex network, as there can be different kinds of connections between the nodes. For example, in our example data set letters are connected to people both by senders and receivers.

From this perspective, the question is not only about what relational data can do to network analysis, but what networks (as a framework) can do to managing

data. The answer would be, that conceptual and logical data modeling are about making a network representation about the data, and this representation can then serve as basis for creating a relational data(base).

6.7 Note About Unique Identifiers

The primary keys of a relational data set can be specific to your data collection, but there is often a better alternative. Internet is full of databases that identify people, cities and other entities, and these databases often have useful information in them. Choosing e.g. a VIAF identifier for authors, or a identifier of a historical place name from Getty Thesaurus of Geographic Names might allow linking of your data set to useful information. What is even better, that many of these internet resources are linked data. Linked data is interlinked with other internet data resources, which means that an identifier obtained for a "thing" from one database can often be linked with its identifier in another database. Linked data, then, can in some cases allow you to significantly enrich your data set, if you can match the entities in your data set with corresponding identifiers in a linked data set.

Linked data is also uniformly structured, and it can be queried with SPARQL. We do not teach SPARQL in this course, but we greatly recommend getting familiar with it, as it can be used to query data from an enormous collection of interlinked data sets on the internet. This data, then, can be used to enrich your data, if you can get your hands on it, for which SPARQL is the easiest solution. Both linked data and SPARQL will be discussed in more detail in the reading for this week.

6.8 References

Chapter 7

Week 3, Class 2: Relational Data Part 2

7.1 Working With Relational Data

Last week, we argued that arranging your data according to the principles of relational data is beneficial. It allows for efficient storage of data (we do not repeat the same information) and is unambiguous in distinguishing entities from each other (the requirement of primary key for each table). Each table is also easier to understand. All of them store information about entities of one type, one entity per row.

However, working with multiple tables instead of one also means we need to be able to join, intersect and otherwise manipulate a collection of tables instead of one. Fortunately, the main tools to do that have already been introduced during the first week. By using the joins, working with multiple tables is easy. The main idea is that the primary keys of each table are foreign keys in other tables (or even in the same table in another column). By matching primary keys of one table to the foreign keys, we can connect entities of different type to each other by their relations. Combined with other tidyverse functions, joins make relational data easy to manage and "query". By querying, we mean selecting subsets of the data.

Next, we will go through joins by explaining in more detail what they do and provide examples of the different operations. ## Joins and relational data The second chapter of the first week introduced the basic idea of joins and demonstrated the use of inner_join. We recommend reading that section about joins again first. The other joins also work by connecting tables by keys, but they differ in what they do after matching the keye in the tables. left_join, right_join and full_join are like inner_join in the sense that they return variables from

both tables. left_join and right_join add the variables from the right (left) table to the left (right) table based on key matches, whereas full_join returns rows from both tables. Usage of these functions can create NA values to the data.

anti_join and semi_join only return variables of the first argument. First returns only those rows for which the key(s) used in matching can only be found from the first argument. Later returns all rows from the first argument with mach in the second.

To demonstrate the use of joins with relational data, we return to the example (pseudo) correspondence network from the last chapter.

```
print("A table of persons:")
```

```
## [1] "A table of persons:"
```

```
print(pseudo_data_person)
```

```
##   person_id person_name person_occupation
## 1    p_id_1         Hue        theologian
## 2    p_id_2       Louie      lens grinder
## 3    p_id_3       Dewey       philosopher
```

```
print("A table of cities:")
```

```
## [1] "A table of cities:"
```

```
print(pseudo_data_city)
```

```
##   city_id city_name city_location
## 1  c_id_1 Frankfurt          Main
## 2  c_id_2    Aachen     Westfalen
## 3  c_id_3 Frankfurt          Oder
```

```
print("A table of letters:")
```

```
## [1] "A table of letters:"
```

```
print(pseudo_data_letter)
```

```
##   letter_id                              letter_name    letter_topic sender_id
## 1    l_id_1              Absence of data management            data    p_id_3
## 2    l_id_2    Absence of tenures for lens grinders         economy    p_id_1
## 3    l_id_3 Absence of pseudo name for letter titles meta commentary    p_id_2
##   receiver_id city_in_which_received_id
## 1      p_id_1                    c_id_1
## 2      p_id_2                    c_id_2
## 3      p_id_3                    c_id_3
```

We can use left_join or right_join to connect correspondents to letters.

```
sender_to_letter <- left_join(pseudo_data_person,pseudo_data_letter,by=c("person_id"="sender_id")
print(sender_to_letter)
```

```
##   person_id person_name person_occupation letter_id
## 1    p_id_1         Hue        theologian    l_id_2
## 2    p_id_2       Louie      lens grinder    l_id_3
## 3    p_id_3       Dewey       philosopher    l_id_1
##                              letter_name    letter_topic receiver_id
## 1     Absence of tenures for lens grinders         economy      p_id_2
## 2 Absence of pseudo name for letter titles meta commentary      p_id_3
## 3               Absence of data management            data      p_id_1
##   city_in_which_received_id
## 1                    c_id_2
## 2                    c_id_3
## 3                    c_id_1
```

We could also combine the joining of tables with other operations. For example, we could first filter the occupations of the people sending the letter to be academic ones, and then join join the table. This would result in a table of letters sent by people with academic occupation:

```
sender_to_letter_academic <- pseudo_data_person %>% filter(person_occupation!="lens grinder") %>%
print(sender_to_letter_academic)
```

```
##   person_id person_name person_occupation letter_id
## 1    p_id_1         Hue        theologian    l_id_2
## 2    p_id_3       Dewey       philosopher    l_id_1
##                           letter_name letter_topic receiver_id
## 1 Absence of tenures for lens grinders      economy      p_id_2
## 2           Absence of data management         data      p_id_1
##   city_in_which_received_id
## 1                    c_id_2
## 2                    c_id_1
```

The other joins can be used to link and subset tables as well. If we wanted to compare our data set to another set of people with an additional column birth year:

```
print(pseudo_data_person_new_table)
```

```
##   person_id person_name   person_occupation person_birth_year
## 1    p_id_3       Dewey         philosopher              1600
## 2    p_id_4       Agnus          theologian              1575
## 3    p_id_5     Crassus natural philosopher              1590
```

Then we could use full_join to get people in both tables:

```
full_list_of_people <- full_join(pseudo_data_person,pseudo_data_person_new_table,by=c('
print(full_list_of_people)
```

```
##   person_id person_name.x person_occupation.x person_name.y person_occupation.y
## 1    p_id_1           Hue          theologian          <NA>                <NA>
## 2    p_id_2         Louie        lens grinder          <NA>                <NA>
## 3    p_id_3         Dewey         philosopher         Dewey         philosopher
## 4    p_id_4          <NA>                <NA>         Agnus          theologian
## 5    p_id_5          <NA>                <NA>       Crassus natural philosopher
##   person_birth_year
## 1                NA
## 2                NA
## 3              1600
## 4              1575
## 5              1590
```

Or inner_join for those in both tables

```
exclusive_list_of_people <- inner_join(pseudo_data_person,pseudo_data_person_new_table
print(exclusive_list_of_people)
```

```
##   person_id person_name.x person_occupation.x person_name.y person_occupation.y
## 1    p_id_3         Dewey         philosopher         Dewey         philosopher
##   person_birth_year
## 1              1600
```

Notice that both full_join and inner_join (and also left_join and right_join) return variables from both tables, which sometimes returns NA's like in the full_join example. The x and y at the end of the names of the table are the result of us doing the joining by the unique identifier, other variables are treated

as separate even if they have the same name. Without the specification of the variable by which the matching is done, variables with the same name (and only those) are treated as same.

anti_join and semi_join only keep the variables from the first argument. anti_join could be used to keep only the people from the first table of people with the following:

```
full_list_of_people <- anti_join(pseudo_data_person,pseudo_data_person_new_table,by=c("person_id"
print(full_list_of_people)
```

```
##   person_id person_name person_occupation
## 1    p_id_1         Hue        theologian
## 2    p_id_2       Louie      lens grinder
```

Semi join would return only those instances of the first table that are present in the second one, in this case the one shared person:

```
full_list_of_people <- semi_join(pseudo_data_person,pseudo_data_person_new_table,by=c("person_id"
print(full_list_of_people)
```

```
##   person_id person_name person_occupation
## 1    p_id_3       Dewey       philosopher
```

Joins, filters and other functions can also be used to subset data in more complex ways. For example, we could want to study only letters sent by philosophers, born in the 17th century and present in both lists of people (perhaps the lists are from two different scholars, and we only want to study individuals they both have listed). We would get exactly this information in multiple ways, here's one.

```
philosophers_both_lists_17th_century <- pseudo_data_person_new_table %>% filter(person_birth_year
print(philosophers_both_lists_17th_century)
```

```
##   letter_id               letter_name letter_topic sender_id receiver_id
## 1    l_id_1 Absence of data management         data    p_id_3      p_id_1
##   city_in_which_received_id
## 1                    c_id_1
```

First, we required the occupation and the birth year in the filter. After that, we used semi_join to find those people who are present in both tables (that now must also be philosophers born in the 17th century, as one of the tables was filtered so before the join). Having now the right people, we used semi_join again to connect these individuals to the letters sent by them (we matched the

tables by matching the primary key person_id of one table to the foreign key sender_id of another), and as semi_join only returns the variables from the first argument, we only get the letters from these individuals.

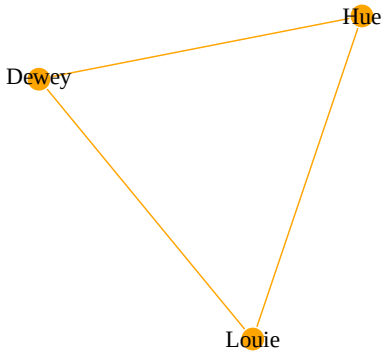
We have no discussed how to work with relational data at a general level. To not forger that the main theme of this course are networks, we next demonstrate how this example data can be used to construct one.

7.2 Creating A Network From Relational Data

We have now spent some time speaking about data in general, and less about networks specifically. However, relational data can be the basis for constructing a network as well, and we demonstrate it here by using the tools and (relational) data we are already using.

Relational data often makes creating the node list a trivial thing. If the nodes are one of the entities in your data, you already have it. For example, in our correspondence network the table of people is a perfectly fine node list as such, no additional work needed. Because the correspondence network depicts connections formed by exchange of letters, the edge list should have at least all the id's of senders in the from column and all the id's of the receiver in the to column. In our case, we also get the edge list directly from the relational data, as the table of letters has the id's of the receivers in one variable and the id's of senders on another.

```
node_list <- pseudo_data_person %>% distinct(person_id,person_name)
edge_list <- pseudo_data_letter %>% distinct(sender_id,receiver_id) %>% rename(.,from=s
graph <-  igraph::graph_from_data_frame(d=edge_list, vertices=node_list, directed = FA
plot(graph, edge.arrow.size=.2, edge.color="orange",
     vertex.color="orange", vertex.frame.color="#ffffff",
     vertex.label=V(graph)$person_name, vertex.label.color="black",edge.width=E(graph)$
```



We could have constructed the network without the steps of data modeling and turning the data into relational data. On the other hand, in our example doing the data modeling and splitting first might have made "seeing" the elements of the network easier than it would have been in one table with all the information related to all types of entities. As conceptual and logical data models can be visualised as graphs, we can turn the idea other way around: by taking a network perspective on data, we can see it structure as edges (relations between attributes) and nodes (entity types).

7.3 Other Practical Questions With Relational Data

Here we go through some other aspects of working with (relational) data.

How to store it? Like we discussed last week, relational data is often stored to databases. However, even a collection of csv tables - each storing one of the entity tables - will do in many instances. For work that you do by yourself or in a small group, this is often enough.

What to do if there are no unique identifiers in the data, how do I get primary keys? At least three solutions exists, creating your own identifiers, getting them from some external resource, or using some external tool to create unique identifiers for you. If you have a fixed amount of data that has already been harmonised (e.g. there are no longer name variants or such as different entities),

then something as simple as giving an unique id for each row in each entity table can be enough. There are also R functions that create ids out of unique combinations of fields if some set of attributes define the entities uniquely or closely enough. For example, we could deduce that people with the same name, same birth year and same occupation would most likely be the same person, but whether or not to do this is good enough depends on the context.

In the simple case in which you have data that has been already harmonised and some attributes uniquely define a person, the creation of primary keys can look something like this:

```
pseudo_data_person_new_table_without_ids <- pseudo_data_person_new_table[,c(2:4)]
#This is the second table of people without primary keyes
print(pseudo_data_person_new_table_without_ids)
```

```
##   person_name   person_occupation person_birth_year
## 1       Dewey         philosopher              1600
## 2       Agnus          theologian              1575
## 3     Crassus natural philosopher              1590
```

```
#Now we add them
pseudo_data_person_new_table_with_ids <- pseudo_data_person_new_table_without_ids %>% m
print(pseudo_data_person_new_table_with_ids)
```

```
##   person_name   person_occupation person_birth_year
## 1       Dewey         philosopher              1600
## 2       Agnus          theologian              1575
## 3     Crassus natural philosopher              1590
##                 unique_combinations
## 1           Dewey-philosopher-1600
## 2            Agnus-theologian-1575
## 3 Crassus-natural philosopher-1590
```

```
#And if we want to have more concise identifier, then we could do something like this
pseudo_data_person_new_table_with_ids_concise <- pseudo_data_person_new_table_with_ids
print(pseudo_data_person_new_table_with_ids_concise)
```

```
##   person_name   person_occupation person_birth_year
## 1       Dewey         philosopher              1600
## 2       Agnus          theologian              1575
## 3     Crassus natural philosopher              1590
##                 unique_combinations person_id
## 1           Dewey-philosopher-1600    p_id_3
## 2            Agnus-theologian-1575    p_id_1
## 3 Crassus-natural philosopher-1590    p_id_2
```

The generation of the person id might seem a little confusing, but it is simple when broken down into pieces. Each of the unique combinations of name,occupation and birth year is treated as a factor (a data type in R), each factor has a number assigned to it, there are as many as there are levels in the factor. In our case, the factor has three levels, one for each unique name-occupation-birth year combination. The factor is converted to the number of its level with as.numeric, and this number is attached at the end of the person_id to define the combination of information uniquely.

This is an idealised situation, as the data was already harmonised and we could be certain of the attributes that uniquely define a thing. This is one of the reasons why harmonisation is so important in practical work with (relational) data.

Like we discussed at the end of the last week, it is also possible to use external resources to get unique identifiers. As this approach has many potential benefits, like the potential to significantly enrich your data with new information, the reading from this week will be an introduction to the ideas of open linked data and SPARQL.

7.4 For the Next Week

The reading for the next week is part of a Programming Historian tutorial about linked open data [**?**]. Read the following sections from it: Introduction and Lesson Scope,Linked open data: what is it?,The role of the Uniform Resource Identifier URI and How LOD organises knowledge:ontologies. We will discuss the tutorial next week. Especially, everyone should be prepared to answer these four questions in front of the class. Who answers will be selected randomly during the lecture (one person per question, no more than one question per student). The questions are:

The questions are:

1) Q: What are the parts of a triple?
2) Q: What does ontology mean (in this context)?
3) Q: What can URI (unique resource identifier) describe?
4) Q: What can Linked Open Data do that a Relational database can't?

7.5 References

Chapter 8

Week 4, Class 1: Network Concepts and Metrics

8.1 Exercises

This lesson has a corresponding editable notebook containing some exercises. These are stored as a separate notebook in the CSC notebook workspace, with the name `4-1-fundamentals_nb.Rmd`. You'll need to move a copy to your 'my-work' folder in your CSC notebook Workspace, complete the exercises, and knit them, using the method we learned in the first class. Once this is done, send a copy of the HTML file to the course leaders.

8.2 Introduction

This lesson will cover a number of concepts. First, it outlines the concept and theory behind a number of key network metrics. These are generally divided into node-level and global metrics. Many node-level metrics are interested in *centrality*, that is, working out the most important and most influential nodes in a network. In a social network, this may point to individuals with a particular influence or role in the network, but it can also have other uses: Google's PageRank algorithm uses a version of centrality to work out which web pages are likely of the highest quality or usefulness for a particular search, for example.

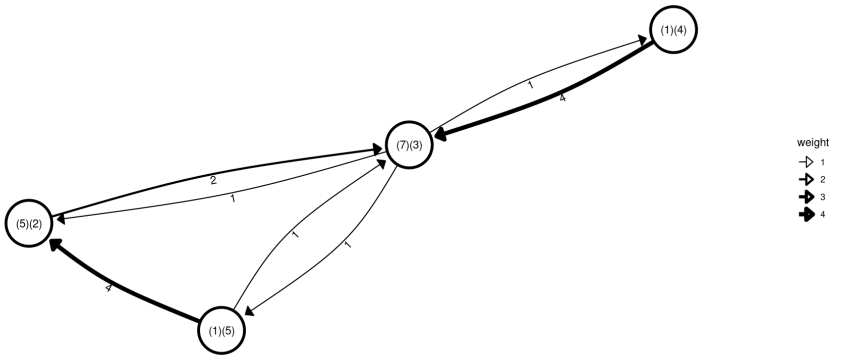
Second, we'll talk briefly about some theories specific to *social network analysis* (or SNA). Social networks (that is, networks of people, where the connections and structure is governed by social factors) have their own set of concepts, which do not apply to all networks. We won't go into detail (that could be a whole other course), but it's useful to understand.

8.3 Node-level metrics

8.3.1 Degree

The degree of a node is a count of its connections. Degree can be weighted and directed. In a directed network, the separate degree counts are called in-degree and out-degree. The total degree is the sum of the two.

In this graph, the ego is connected to each alter by an incoming and outgoing link. Weights can be added to the links: degree is a sum of the weights of the links, in that case. The edge weights are labelled in the diagram below, and the numbers inside each node give the in and out degree measurements.



Depending on the network, these in and out-degree measurements may signify different types of nodes. For example, imagine a network of Twitter accounts: each follower is counted as an *incoming* link, and each person you follow is counted as an *outgoing* link, for the purposes of your degree score. If you have a high *in-degree*, you might have a high influence in this network, because you can broadcast ideas or whatever to a large number of followers. Having only a high *out-degree* in this network, on the other hand, may not make you very influential at all, because it means you follow many accounts but few follow you back.

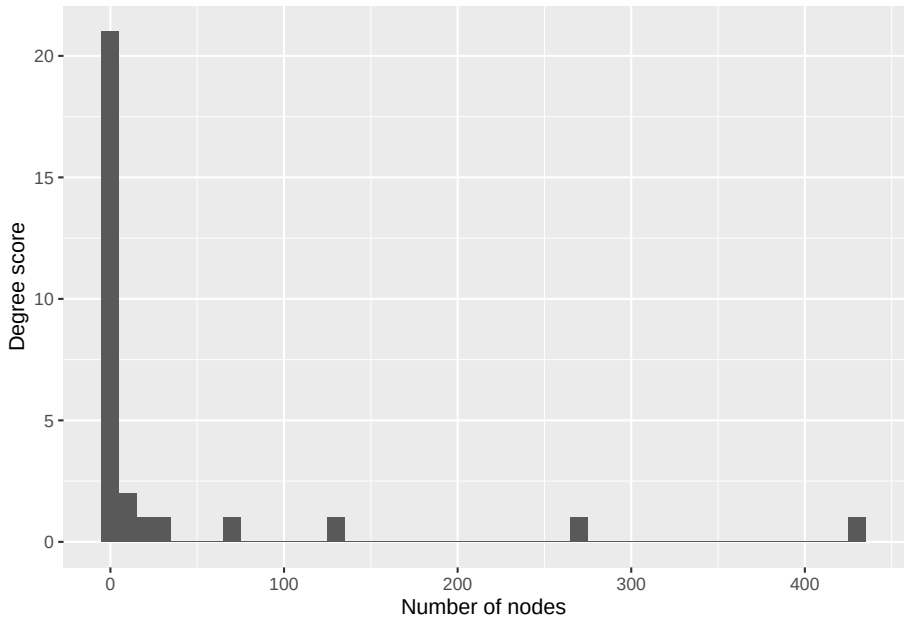
Degree is the most common measurement of network *centrality,* or importance. A node with a high degree is likely to have more influence in the network, to have more paths pass through it, and so forth.. However, it doesn't always point to the most 'important' nodes.

First, degree measures *quantity* over *quality* of links. The node with the highest absolute number of links will always be ranked highest, no matter which nodes those links point to.

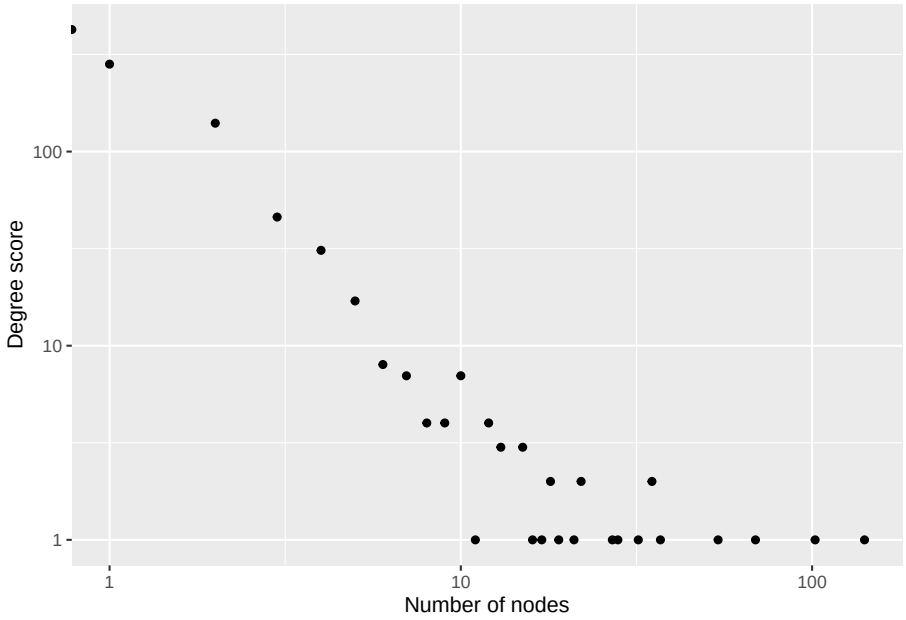
8.3.1.1 Degree distribution

Degree is used to measure structural properties of the network. We can do this using the **degree distribution**. A distribution counts how many nodes have a degree of 1, a degree of two, three, and so forth, or sometimes a range, such as a degree of between 1 and 5, 6 and 10, and so on. As mentioned in a previous chapter, this degree distribution tells us some valuable structural properties about the network, such as the extent to which it is a collection of hubs and less important nodes (known as 'scale-free').

To illustrate the degree distribution, below is a histogram of the degree distribution for a randomly-created scale-free network with 1000 nodes and 1000 edges. It shows that 0 to 10 nodes have a degree score of 50, 11 - 20 have a degree score of 4, and at the other end, more than 400 nodes have a degree score of just 1.



Another way to illustrate this is with a log-log plot. A log-log plot shows the same data, except the scales are logarithmic, making them easier to read.

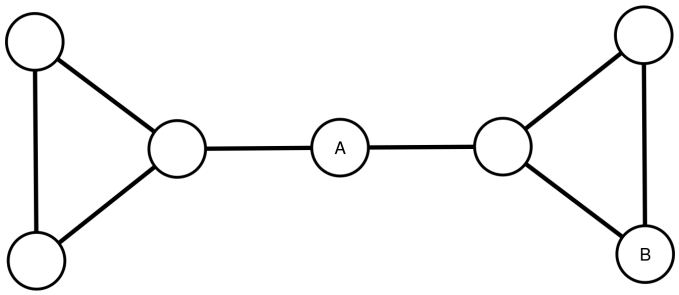


Networks with distributions which look like this on a histogram or log-log plot are likely to be scale-free, meaning their structure consists of a number of important hubs with high degree, followed by a larger number of less-important nodes.

Network science has shown that these network types are very common in all sorts of real-world situations, from biological, to social, to computer networks. They've also been shown to have a number of particular properties: for example, it may make a computer network resistant to random 'attack', because its only when hubs are removed that the network begins to become disconnected. Removing unimportant nodes doesn't affect the overall system. At the same time, this can make them vulnerable, if hub nodes are specifically targeted.

8.3.2 Betweenness Centrality

Betweenness centrality measures all the paths between every pair of nodes in a network. A node has a high betweenness centrality if it is used as a 'hop' between many of these paths. In the below diagram, A has a higher betweenness centrality than B, because it is needed to traverse from the left to the right side of the network.



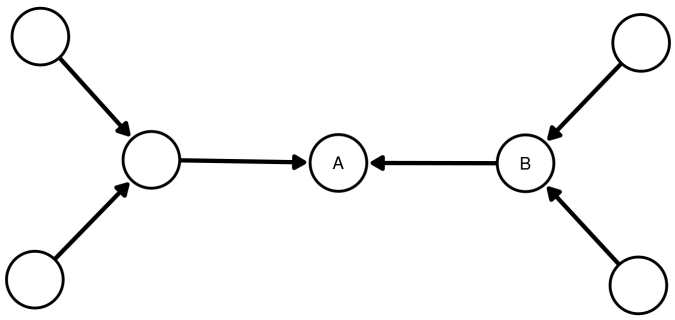
Betweenness centrality for a given node v is calculated by measuring the shortest path between every pair of nodes, and then counting the total number of these paths which pass through v. It can be directed, meaning that paths are only considered if they exist in the correct direction.

As mentioned above, degree is not always the best way to measure the importance of a node. It can also be important to consider the node's particular position, and the role it occupies within a larger system. In the above diagram, node A is the only way through which any of the nodes on the left can reach any on the right.

In a social network, nodes with high betweenness are often thought of as important 'brokers', as we talked about in a previous lesson. However in practice, note that high degree nodes are often the ones which have the highest betweenness values too, and there are often multiple paths between the same pair of nodes, which can limit the use of this metric.

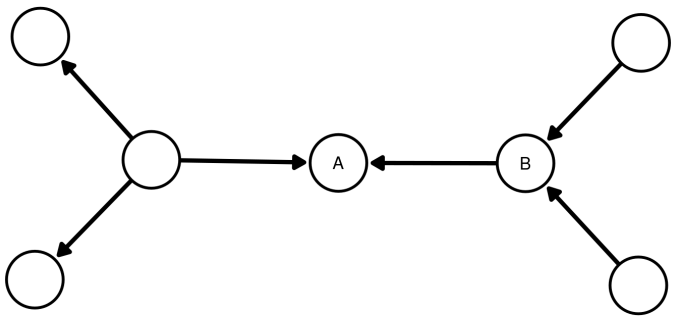
8.3.3 Eigenvector Centrality

A node that has a high eigenvector score is one that is adjacent to nodes that are themselves high scorers. Google's original PageRank algorithm used a version of this concept: ranking web pages higher in search results if they themselves were linked to by other high-scoring pages. In the example network above, both A and B have two incoming connections each, but A has a higher eigenvector centrality score because it is connected to more well-connected nodes.



8.3.4 Closeness centrality

Closeness centrality measures each node's path to every other node in the network: a node is declared more central if its paths to other nodes tend to be short by this measurement (the formula is 1 divided by the average of all the shortest paths from a given node to all other nodes). Closeness centrality might be thought of as a way of measuring how quickly something flowing through the network will arrive. In the diagram below, A has a higher closeness centrality than B, even though both have the same number of connections, because A is 'less far' to all other nodes, on average.

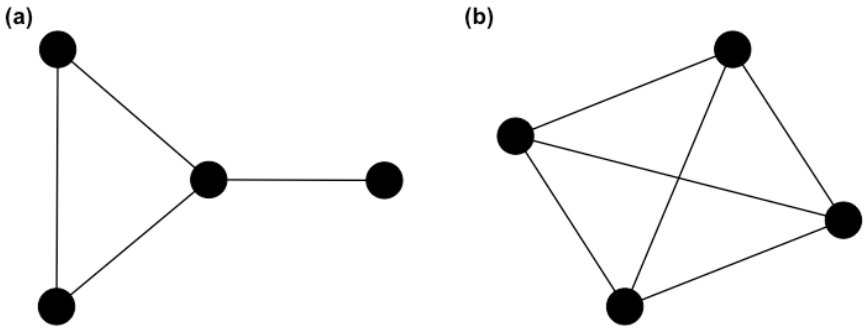


8.4 Edge-level metrics

8.5 Global Metrics

8.5.1 Density

The density of a network is defined as the fraction of edges which are present out of the total possible number of connections. Consider the undirected network below (a), which has four edges in total. The 'full graph' (when every node is connected to every other) of an undirected network with four nodes has six edges (b). The density of (a) therefore is 4/6, or .6666… In practice, most real-world networks are much less dense.



What implication does this have on the 'small world' networks we talked about in an earlier class?

The 'small world' effect works in low-density networks precisely because of the existence of hubs and weak ties. If there are very few ties in a network altogether, but there are a number of very well-connected hubs, then most nodes will be able to reach any other node by going through these hubs.

In a directed network, the possible number of nodes in the full graph is doubled.

On its own, density may not tell us much about the structure of a graph. Two graphs with 20 edges and 10 nodes may look very different: in one, each node may have two connections, and in another, 9 nodes may have one connection each, and the final one 10 connections. However, in combination with some other metrics, most importantly degree distribution, it may be a clue as to the underlying structure of the network.

8.5.2 Average path length

The average path length is the average distance in 'hops' from one node to another in the network. In the Milgram experiments, the average path length

through the network was found to be between 5 and 6. This measurement, therefore, can tell us how long it may take on average for information to get from one side of a network to another. A network with a small average path length is said to be a 'small world'.

Consider a letter network with an average path length of 4. Does this mean that information needs to go through on average 2 further people (because the start and end nodes are counted in the path) to get from A to B?

Well, not necessarily. This is a good example of why we should consider each network and set of connections carefully before using these metrics. Common sense tells us that in reality, if person A really wanted to communicate with B, there may have been no reason why they couldn't send a letter directly, without using the 'shortest path' through the other two nodes.

8.5.3 Clustering coefficient

The clustering coefficient is also known as transitivity, and it is defined as the ratio of completed triangles and connected triples in the graph. The more completed triangles a network contains compared to the overall triples, the more clustered it is said to be.

Clustering is a complex topic which forms a large part of the field of Social Network Analysis. On a local level, transitivity means the probability node B will be connected to node C, *if both nodes are connected to a common neighbour, A.* The higher this probability, the more the graph is said to display clustering tendencies: if you are more likely to connect to your common neighbours, it follows that it is more likely that the graph will be divided into dense cliques and sub-groups, rather than a free-for-all where everyone is connected to everyone else.

8.5.4 Groups and Community Structure in Networks

8.5.5 Components

8.5.6 Cliques

Networks and communities are often thought of as analogous, but in network analysis, the latter are considered a subset of the former. Discovering discrete communities—often called sub-graphs—within a wider network has long been one of the fundamental applications of network science. In epidemiological networks, for example, the extent to which a network tends to break into sub-graphs and the constitution of those parts has important implications for understanding the spread of disease through a system. While the ego networks of the previous chapters are communities in their own right, working with a large set

of merged, overlapping ego networks invites us to think about the whole, and how this whole might be redivided into clusters not necessarily corresponding to the ego networks which constitute its parts.

In a network sense, a community might be generally defined as a sub-grouping within a network which is in some way more densely connected to its internal group members than to those outside the group. The problem, of course, is that if the graph is connected (i.e. if all nodes can reach all other nodes eventually), then the boundaries of any detected community that goes beyond a simple ego network are necessarily arbitrary, flexible, and subjective. It is true that many individuals will sit firmly in the core of a given cluster, but there will be those on the margins who might be considered as members of more than one community—or none.

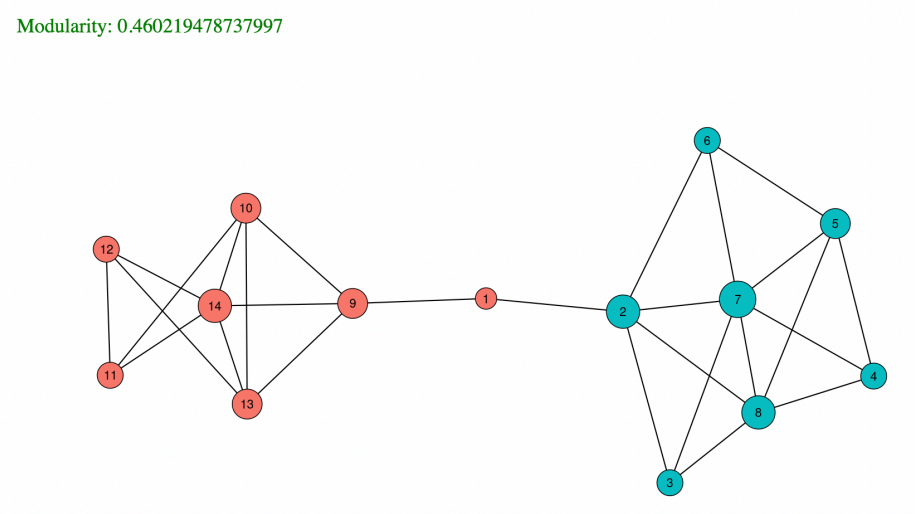
In spite of these problems, this rigid way of thinking about communities can serve as a useful tool, to which we can bring our existing knowledge of historical communities and compare them to an algorithm's results. By forcing us to over-simplify, they may not be able to capture the full extent of multiple overlapping communities of identity, but they do help us to reflect on how we think of individuals operating within communities of correspondence. For example, though we might think of person X primarily as a member of group Y, if we look just at their patterns of correspondence, they seem more firmly in group Z. This is not to say that we should necessarily re-evaluate the scholarship and declare that they were in group Z all along, but it might be an indication that we should re-assess that person's patterns of correspondence—after all, belonging to a community and being more likely to correspond with its members are not the same thing.

8.5.7 Community Detection Methods

Network science has a large of algorithsm for finding optimal communities in graphs. Many rely on a metric called modularity, explained in this next section.

8.5.7.1 Modularity

Modularity is a score evaluating how well a given set of group labels describe the groups found in a network. It's calculated as the number of edges falling within groups minus the expected number in an equivalent network with edges placed at random (link). In the below diagram, the modularity is high, because the groups (coloured as either red or green nodes) correctly describe the two separate 'communities' found in the network. Community detection algorithms work by trying to maximise this value.



As with the clustering coefficient, a high modularity indicates the extent to which the network might be described as a collection of densely-connected subgroups, or whether the links are spread out evenly across the network.

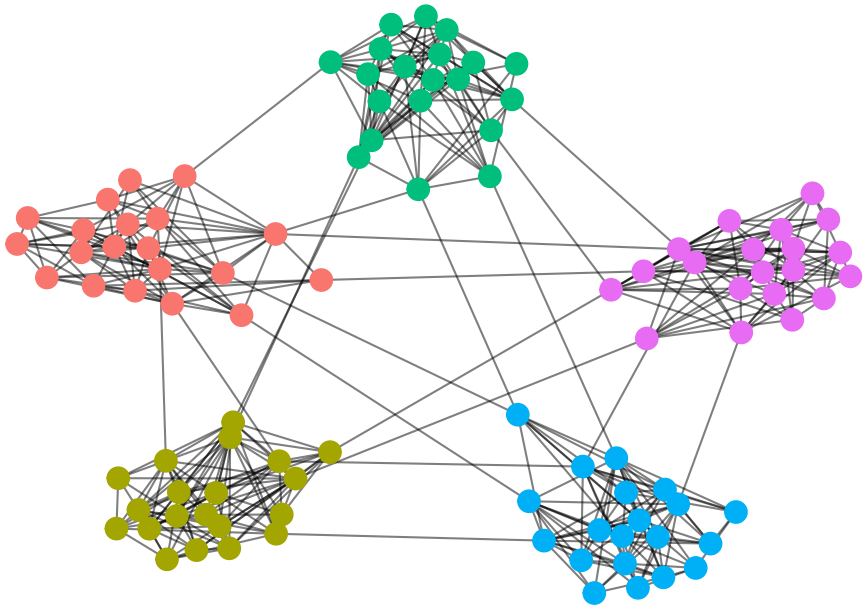


Figure 8.1: A network with high modularity. The given partitions (separate colours) successfully divide the network into groups with many more connections between the groups than to other groups.

8.5.7.2 Louvain Community Detection

The Louvain algorithm is one algorithm which uses modularity, developed specifically for use in large graphs. At first, each node in the network is assigned to a community of just itself. Then, one at a time, each node is temporarily moved to the community of each of its neighbours, and the modularity of the whole graph is re-checked using the method above. Each node is kept in the community which results in the highest change in overall modularity.

At this point, most nodes are now in a community of two (if there's no change in modularity, they stay in their own community). The next step treats these mini-communities as nodes in a network in their own right, and runs the same process again (putting each mini-community into the mini-community of its neighbour, calculating overall modularity, and moving to that which gives it the highest gain). This whole process then repeats until modularity stops increasing and communities stop merging with other communities—or a pre-supplied threshold is reached.

In very large graphs, the process is more difficult and likely to give different results depending on the random starting point. Also, while the method may be a useful way of finding communities without preconceived starting points, it does require us to think of the network as being divided into communities of some kind. Every node must be placed in a community - none can be placed outside the communities, and, perhaps more controversially, with this method, no node can be a member of more than one community.

Important to note is that communities are to some degree subjective, and 'densely-connected' can have different meanings depending on the algorithm. And needless to say, detecting communities on a graph alone does not make a research question!

8.6 Social network concepts: triadic closure, transitivity, brokerage

8.6.1 Triads

There are a number of different triad types, each of which, according to social theorists, tell us something about the particular role of that set of people. The proportion of various triad types in a network can tell us about its structure, and, in the case of a complex network, something of the process by which connections are formed. In this way, we connect the *micro* structure (the information on individual pairs or triads) with the *macro* (what this tells us about the overall formation and structure of a network).

In an undirected network, there are four possble triad types, as seen in the diagram below. The first, a **closed triad**, is when each node is connected to

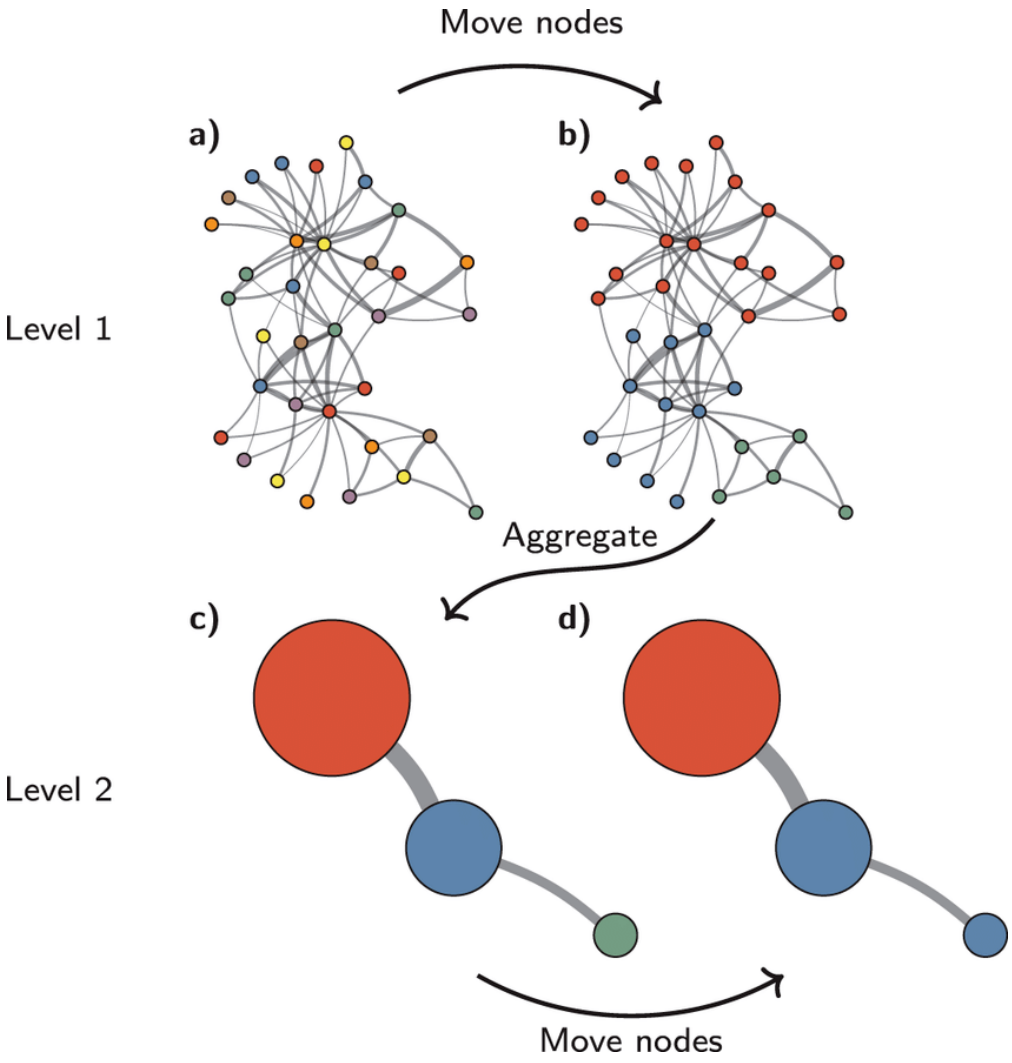


Figure 8.2: From Traag, V.A., Waltman, L. & van Eck, N.J. From Louvain to Leiden: guaranteeing well-connected communities. Sci Rep 9, 5233 (2019). https://doi.org/10.1038/s41598-019-41695-z

the other two. The other possibilities are **open triad**, where one of the three possible connections is not present, a **closed pair**, where just one pair of the three is connected, and a **disconnected triad**.

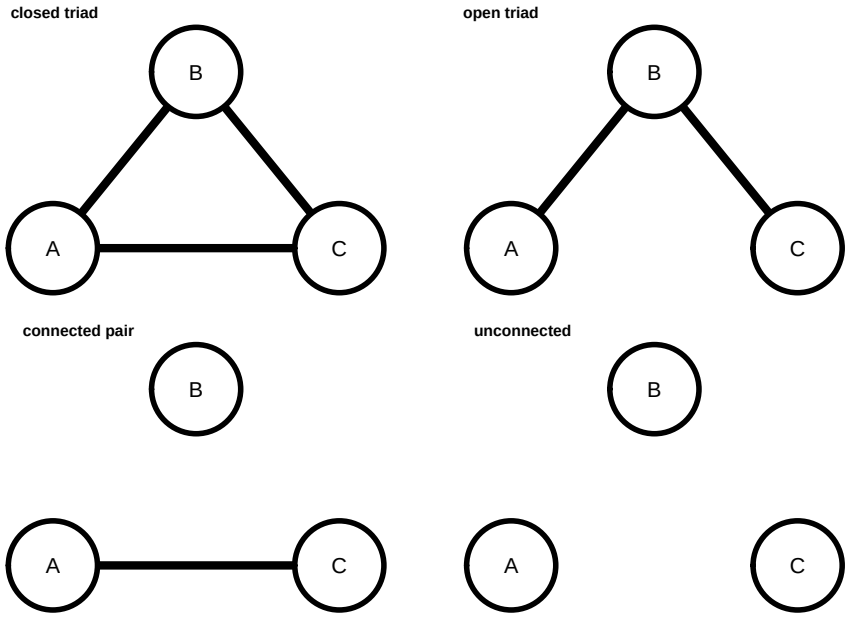


Figure 8.3: Four possible triad types in an undirected network

In a directed network, the situation is more complex. Each of the connections can be incoming or outgoing, meaning there are 16 possible triad connections. Don't worry, if you want to use these, network analysis software has methods for taking a 'census' of the various triads and their counts in a particular network.

8.6.1.1 Triadic Closure

Triadic closure is the simple idea that people in a social network are more likely to connect to each other if they share another common connection. In social network theory, this is very common, and thought to drive the process of network formation. It's called triadic closure, because it measures the relation between triads - sets of three nodes. It's easy to measure in a network: we can do so simply by measuring the number of 'complete' triangles.

Why is this interesting? As well as helping to explain network formation, we can think about how various triads work on a micro-level. Let's start with a simple example: an undirected friendship network between three people. A and B know each other, as do B and C, but A and C have never met:

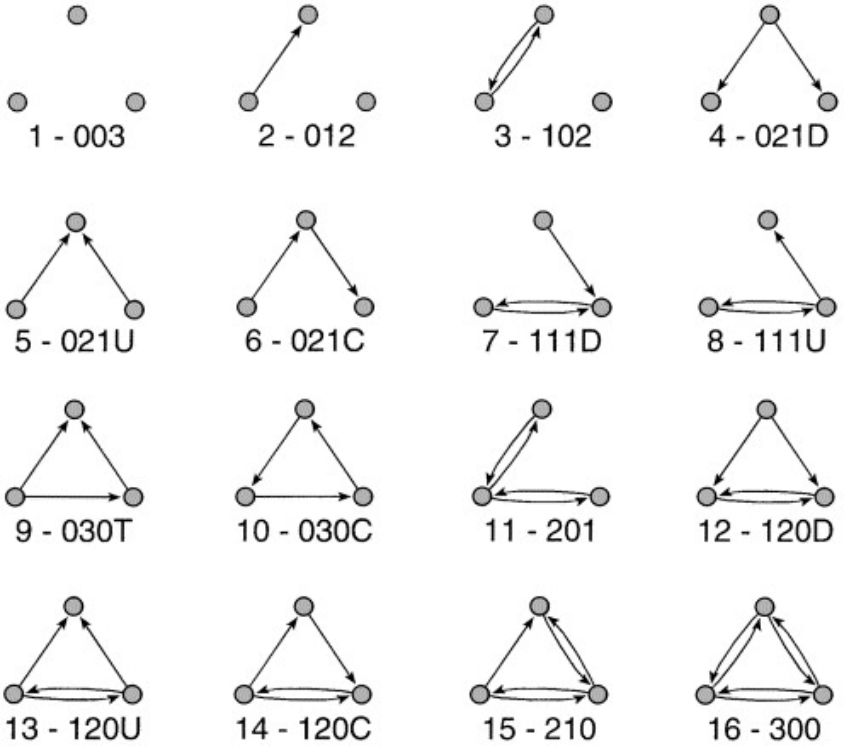


Figure 8.4: The 16 possible triad combinations in a directed network. The code underneath each triad refers to the numbers of mutual, asymmetric, and null dyads, with a further identifying letter: Up, Down, Cyclical, Transitive. E.g., 120D has 1 mutual, 2 asymmetric, 0 null dyads, and the Down orientation.

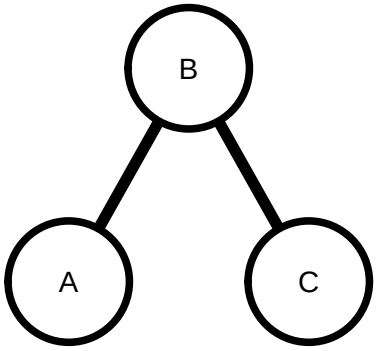


Figure 8.5: An open triad, meaning one with a missing third connection.

Mark Granovetter's paper 'The Strength of Weak Ties' looked at the formation of these triads their role in a network. For Granovetter, edges in a network are formed of either 'strong' or 'weak' ties. He writes that in an unclosed triad, if the two existing edges are strong ties, a third edge will always eventually appear between the final two nodes, forming a closed triad. An unclosed triad containing two strong edges is impossible and he calls this the 'forbidden triad'.

This gap, from A to C, is sometimes called a 'structural hole'. In this triad, B is particularly important because they are needed for information to be shared between A and C. B may even work to keep A and C from meeting and 'closing' the triad. The sociologist Ronald Burt, in his book *Structural Holes: The Social Structure of Competition* theorised that these gaps could be beneficial to individuals, and that people with many of these kinds of relationships were often successful in business.

Another influential paper, by Gould and Fernandez, used triads in their theory of *brokerage*. This is a process, according them, where individuals act as intermediaries in a network, helping to share information. They identify five types of brokerage roles, based on specific triad types, as seen in the image below:

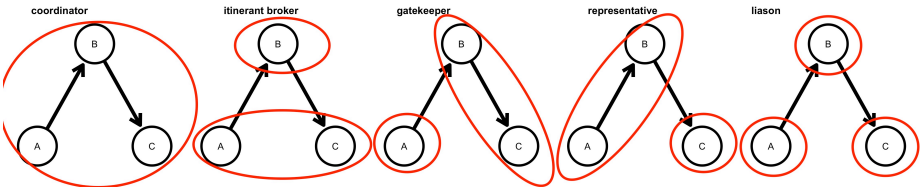


Figure 8.6: The five different brokerage roles, according to Gould and Fernandez (1989). In each case, the ellipses represent different 'sub-groups' of the wider network that each node belongs to.

8.6.1.2 A Word of Warning...

It's easy to see the appeal of these methods applied to historical or cultural sources, which we might use to uncover the specific social relations within a network of historical actors, understand the formation by which a particular correspondence network came about, or maybe measure the 'success' of a particular historical figure by the existence of 'structural holes' in their networks. I encourage you to think about and use these theories. However, do bear in mind that these theories were often developed using modern, 'complete' social network data, where every interaction was carefully recorded, often for the purposes of a particular test. As these methods rely heavily on these 'micro' structures (such as the presence or absence of a particular tie), they may be particularly sensitive to the kinds of missing data often found in historical sources, which are not usually collected systematically.

8.7 Conclusions

This tour of network metrics has just introduced some of the more common ways a network can be measured. All of these metrics can be applied to any network, but bear in mind that many have been developed with the specific aim of measuring relations between people in some sort of set of social interactions, and the conclusions drawn are based on the consideration that the individuals are in some sense 'free agents', and can freely connect with other nodes as they wish.

In many other network types (books, or cities for example), there may be hidden factors and other constraints guiding the structure of the network. This is not to say that the metrics cannot be used, but just that the conclusions from them may not be the same as those in SNA literature.

8.8 Class Assignment

Before we get to coding our own networks on Friday, first we'll use some ready-made tools to generate a set of network metrics and get familiar with them.

Your task for this assignment is to load a network into an online tool called Network Navigator, available here: https://networknavigator.jrladd.com/, and report on its basic network statistics.

You can use any network data you like. We have prepared a sample dataset called `sample_publisher_network.csv`, available in the `applying_networks_to_humanities/data` folder, but you are welcome to use any other of the sample datasets or another dataset you have found online.

The instructions to upload the network data are available within the web application. In short, you should drag and drop the relevant .csv file into the appropriate box. Note that you can choose whether or not your csv has a header row. If it does, click the correct radio button. The application expects headers called `source`, `target`, and (optionally), `weight`. The `sample_publisher_network.csv` file has a header in this format, but if you are using other data you may need to edit the file first.

Based on the information in today's lesson, write a short (about 10 bullet points) report on the structure of this network. Some information you could include:

- Global network metrics. Is it dense or sparse? Is it highly clustered?
- The distribution of degree and other network metrics. Do a small number of nodes have most of the links, or is it evenly spread out?

- A description of the network graph itself.

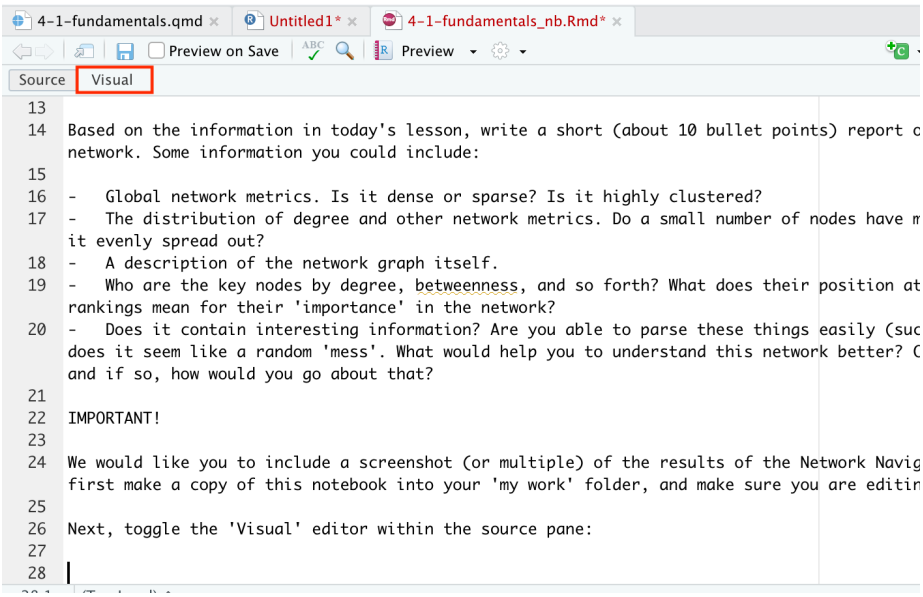
- Who are the key nodes by degree, betweenness, and so forth? What does their position at the top of these rankings mean for their 'importance' in the network?
- Does it contain interesting information? Are you able to parse these things easily (such as the structure), or does it seem like a random 'mess'. What would help you to understand this network better? Could it be filtered, and if so, how would you go about that?

IMPORTANT!

We would like you to include a screenshot (or multiple) of your results from the Network Navigator tool. To do this, first make a copy of this notebook into your 'my work' folder, and make sure you are editing this copy.

Take and save a screenshot and save it to your local machine. Upload it to the 'my work' folder on CSC Notebooks, using the file browser.

Next, switch to the R studio 'Visual editor' interface using the button highlighted below



The visual editor looks like a simple word processor, and has an option for inserting an image. Click this and insert the screenshot(s) you uploaded.

Once you're finished, save and send your work as usual.

Chapter 9

Week 4, Class 2: Network Analysis with R

9.1 Introduction

In this class, we'll talk through creating and analysing a network object in R. The network we'll work with is a sample of correspondence data taken from the British State Papers in the seventeenth century.

9.2 Network data structures

Network data comes in a number of forms. Two common ones are adjacency matrices and edge lists. An adjacency matrix is a matrix of rows and columns, one for each node. If there is a link between two nodes, a 1 is put in at that point. If it is a weighted network, the weight can be entered in the correct space. In some cases, your data may be in a format where it is easy to use it as an adjacency matrix.

Adjacency matrices can be easily read into R using `igraph` and `graph_from_adjacency_matrix()`.

The other common type is what we will work with in this lesson: an edge list.

9.3 Creating a Network Object in R from an Edge List

One of the easiest data formats to construct a network is an edge list: a simple dataframe with two columns, representing the connections between two nodes,

one per row. If the network is directed, generally the 'Source' node is on the left side, and the 'Target' node on the right. It makes particular sense with correspondence data, which is often stored as records of letters with a 'from' and a 'to'—more or less a ready-made edge list. In a correspondence dataset you might also have multiple sets of each of the edges (multiple letters between the same pair of individuals). This will be added to the edges as a 'weight'.

We will use three R network libraries to do almost everything network-related, from analysis to visualisation: `igraph`, `tidygraph` and `ggraph`. The goal is to port everything to a format which is easy to work with using existing an established data analysis workflow. That format is known as 'tidy data', and it is a way of working with data which is easily transferable across a range of uses. It also means you need to learn very little new programming to do network analysis if you stay within this 'ecosystem'.

9.3.1 Import Network Data

The workflow uses a number of R packages. In the CSC Notebooks environment, these have already been installed and can be loaded using the commands below. If you are doing this on a local machine, you may have to install them first using the command `install.packages()`, with the package name specified as a string, for example `install.packages('igraph')`.

```
library(tidyverse)
library(igraph)
library(tidygraph)
```

In this class, we'll use a dataset derived from the English State Papers during the Stuart era (1603-1714). It takes the form of a .csv containing the information on the author, recipient, and date of sending for a small sample of state letters received between 1670 and 1672.

Read the file into R with the `read_csv()` function from a previous lesson:

```
letters = read_csv("letter_data.csv", col_types = cols(.default = "c"))

letters
```

```
## # A tibble: 1,374 x 6
##    letter_id from_id                from_name               to_id to_name date
##    <chr>     <chr>                  <chr>                   <chr> <chr>   <chr>
##  1 1         E006019-S012734-T000000 Dr. Ralph Cudworth, Ma~ E022~ Willia~ 1670~
##  2 2         E006019-S012734-T000000 Dr. Ralph Cudworth, Ma~ E022~ Willia~ 1670~
##  3 3         E011771-S014372-T000000 Hyde, Edward            E015~ Sir Ed~ 1670~
##  4 4         E015103-S043299-T000000 Sir Philip Musgrave     E022~ Willia~ 1670~
```

```
##  5 5         E020134-S040615-T000000 Taylor, Silas           E022~ Willia~ 1670~
##  6 6         E014743-S041280-T000000 Sir Edward Misselden    E022~ Willia~ 1670~
##  7 7         E015103-S043299-T000000 Sir Philip Musgrave     E022~ Willia~ 1670~
##  8 8         E020485-S043979-T000000 Sir Thomas Allin        E022~ Willia~ 1670~
##  9 9         E020485-S043979-T000000 Sir Thomas Allin        E022~ Willia~ 1670~
## 10 10        E004654-S006979-T000000 Charles II, King of En~ E012~ James ~ 1670~
## # ... with 1,364 more rows
```

The letters dataset is a simple dataframe. Each row represents a letter record, and has a unique ID. Essentially, each row is a record of who the author and recipient of the letter. Each of these senders and recipients also have both a unique ID, and the original name of the letter writer and sender. The unique ID is used because it's quite likely that the names are not unique, and the network could combine two nodes with the same name together, for example.

9.3.2 Make an edge list

This is used to construct an edge list. If you have multiple letters between individuals, you can count them and use as a weight in the network, or you can ignore it. This is done with `tidyverse` commands we learned previously: `group_by()` and `tally()`, changing the name of the new column to 'weight'.

```
edge_list = letters %>%
  group_by(from_id, to_id) %>%
  tally(name = 'weight')

edge_list
```

```
## # A tibble: 364 x 3
## # Groups:   from_id [198]
##    from_id                 to_id                   weight
##    <chr>                   <chr>                    <int>
##  1 E000145-S012650-T000000 E022443-S042999-T000000      1
##  2 E000189-S014115-T000000 E004830-S019312-T000000      4
##  3 E000189-S014115-T000000 E006871-S022202-T000000      1
##  4 E000189-S014115-T000000 E022443-S042999-T000000      1
##  5 E000312-S012741-T000000 E022443-S042999-T000000      2
##  6 E000393-S036175-T000000 E001993-S018912-T000000      1
##  7 E000520-S001769-T000000 E003476-S051213-T000000      1
##  8 E000520-S001769-T000000 E022443-S042999-T000000      1
##  9 E000799-S042396-T000000 E022443-S042999-T000000      1
## 10 E001346-S047404-T000000 E022443-S042999-T000000      1
## # ... with 354 more rows
```

Now you see each unique combination of sender and recipient. If there are multiple letters, this is now signified by a weight of more than one in the weight column. You'll also notice that the other information (letter IDs and actual names) has disappeared. This is not needed to make the network, but we can bring the name information back later.

9.3.3 Turn the edge list into a tbl_graph

Next transform the edge list into a network object called a tbl_graph, using tidygraph. A tbl_graph is a graph object which can be manipulated using tidyverse grammar. This means you can create a network and then use a range of standard data analysis functions on it as needed.

Use `as_tbl_graph()` to turn the edge list into a network. The first two columns will be taken as the from and to data, and any additional columns added as attributes. An important option is the `directed =` argument. This will specify whether the network is directed (the path goes from the first column to the second) or undirected. Because this network is inherently directed (a letter is sent from one person to another), we use `directed = TRUE`. In many cases, the network will be undirected, and this should be specified using `directed = FALSE`.

```
sample_tbl_graph = edge_list %>%
  as_tbl_graph(directed = T)

sample_tbl_graph
```

```
## # A tbl_graph: 248 nodes and 364 edges
## #
## # A directed simple graph with 6 components
## #
## # Node Data: 248 x 1 (active)
##   name
##   <chr>
## 1 E000145-S012650-T000000
## 2 E000189-S014115-T000000
## 3 E000312-S012741-T000000
## 4 E000393-S036175-T000000
## 5 E000520-S001769-T000000
## 6 E000799-S042396-T000000
## # ... with 242 more rows
## #
## # Edge Data: 364 x 3
##    from    to weight
##   <int> <int>  <int>
```

```
## 1     1    89      1
## 2     2    22      4
## 3     2   199      1
## # ... with 361 more rows
```

The `tbl_graph` is an object containing two linked dataframes, one for the edges and one for the nodes. The node table is currently a table containing a row for each unique node in the dataframe, with one column, **name**, which is that node's ID. When we calculate additional node-level metrics, they will be added as additional columns to this node table.

The edges dataframe currently contains three columns: **from**, **to**, and **weight**. The **from** and **to** columns contain the edge information: the first row tells us that there is an edge running from node 1 to node 89. These node numbers are not the node IDs but rather correspond to the order of the node table (or the numbers just to the left of the name column). So this tells us that there is an edge going from the node in position 1 (E00145-S012650-T000000) to that in position 89.

You can access each of the tables using the function `activate(nodes)` or `activate(edges)`. The active table is listed first and has the word 'active' in the description. Any commands you do (filtering, joining and so forth) will happen on the *active* table.

```
sample_tbl_graph %>%
  activate(edges)
```

```
## # A tbl_graph: 248 nodes and 364 edges
## #
## # A directed simple graph with 6 components
## #
## # Edge Data: 364 x 3 (active)
##    from    to weight
##   <int> <int>  <int>
## 1     1    89      1
## 2     2    22      4
## 3     2   199      1
## 4     2    89      1
## 5     3    89      2
## 6     4    10      1
## # ... with 358 more rows
## #
## # Node Data: 248 x 1
##   name
##   <chr>
## 1 E000145-S012650-T000000
```

```
## 2 E000189-S014115-T000000
## 3 E000312-S012741-T000000
## # ... with 245 more rows
```

You can use many of the tidyverse commands we learned in the earlier lesson on this object, for example filtering to include only edges with a weight of more than 1:

```
sample_tbl_graph %>%
  activate(edges) %>%
  filter(weight>1)
```

```
## # A tbl_graph: 248 nodes and 165 edges
## #
## # A directed simple graph with 121 components
## #
## # Edge Data: 165 x 3 (active)
##    from    to weight
##   <int> <int>  <int>
## 1     2    22      4
## 2     3    89      2
## 3     9    74      2
## 4    10    89      6
## 5    12   154      2
## 6    13   202      2
## # ... with 159 more rows
## #
## # Node Data: 248 x 1
##   name
##   <chr>
## 1 E000145-S012650-T000000
## 2 E000189-S014115-T000000
## 3 E000312-S012741-T000000
## # ... with 245 more rows
```

9.4 Calculating Network Metrics

9.4.1 Global metrics

The first thing we want to do with this network is to calculate some global network statistics. Because the outputs to these are generally a single number, we don't need to worry about storing them in a table, as we'll do with the node-level metrics later. To calculate these metrics, generally just pass the network to

a relevant function. These metrics were covered in more detail in the previous class.

9.4.1.1 Density

(the number of links present out of all possible links):

```
sample_tbl_graph %>% igraph::graph.density()
```

```
## [1] 0.005942275
```

9.4.1.2 Average path length

(the average number of hops between every pair of nodes in the network):

```
sample_tbl_graph %>% igraph::average.path.length()
```

```
## [1] 11.04997
```

9.4.1.3 Clustering coefficient:

Because there are a number of ways to calculate clustering in a network, a method needs to be specified. The clustering coefficient is also known as *transitivity*, and it is defined as the ratio of completed triangles and connected triples in the graph. This measurement can be *global* (which counts the overall ratio) or *local* (which counts the individual ratio for each node). Because we want the global measurement, specific this with the `type =` argument.

```
sample_tbl_graph %>% igraph::transitivity(type = 'global')
```

```
## [1] 0.02453653
```

9.4.2 Node-level metrics.

There are a number of ways to calculate node-level metrics (these are things like degree, betweenness as explained in the previous class). For example, you can use igraph functions to calculate the degree of single node or group of nodes. The following code returns the degree for the node with the ID `E004654-S006979-T000000` (King Charles II of England). You can return to the original dataset to find the relevant IDs for a node of interest. The argument `mode =` specifies the type of degree: `in`, `out`, or `all`, as we learned in the previous lesson.

```
sample_tbl_graph %>% igraph::degree(v = 'E004654-S006979-T000000', mode = 'all')
```

```
## E004654-S006979-T000000
##                      52
```

You can also look up the degree of multiple IDs by passing them as a vector, using `c()`.

‘

```
sample_tbl_graph %>% igraph::degree(v = c('E004654-S006979-T000000','E006019-S012734-T
```

```
## E004654-S006979-T000000 E006019-S012734-T000000
##                      52                       1
```

9.4.2.1 Adding node-level metrics as a column with tidygraph.

Remember that our tidygraph object is made up of two dataframes, one of nodes and one of edges? We can use the format to add node-level metrics to the node dataframe as additional columns, making them easy to analyse using R later. To do this, we use a function called `mutate()`. Mutate creates a new column containing the value from some calculation, which is performed on each row in the dataset.

Assign the name `degree` to the new column with `degree =`. The column should contain the total degree score for each node. This is done using the function `centrality_degree()`. With the two additional arguments in this function, specify the mode (in, out, or all) and, if a weighted degree score is desired, the column to be used as weights.

```
sample_tbl_graph %>%
  activate(nodes) %>% # make sure the nodes table is active
  mutate(degree = centrality_degree(mode = 'all', weights = weight))
```

```
## # A tbl_graph: 248 nodes and 364 edges
## #
## # A directed simple graph with 6 components
## #
## # Node Data: 248 x 2 (active)
##   name                    degree
##   <chr>                    <dbl>
## 1 E000145-S012650-T000000      1
## 2 E000189-S014115-T000000     21
```

```
## 3 E000312-S012741-T000000      6
## 4 E000393-S036175-T000000      3
## 5 E000520-S001769-T000000      2
## 6 E000799-S042396-T000000      1
## # ... with 242 more rows
## #
## # Edge Data: 364 x 3
##    from    to weight
##   <int> <int>  <int>
## 1     1    89      1
## 2     2    22      4
## 3     2   199      1
## # ... with 361 more rows
```

The data format allows you to use dplyr pipes %>% to perform one calculation on the data, then pass that new dataframe along to the next function. Here we calculate the degree scores first, then filter to include only nodes with a degree score over two:

```
sample_tbl_graph %>%
  activate(nodes) %>%
  mutate(degree = centrality_degree(mode = 'total')) %>%
  filter(degree >2)
```

```
## # A tbl_graph: 51 nodes and 124 edges
## #
## # A directed simple graph with 2 components
## #
## # Node Data: 51 x 2 (active)
##   name                    degree
##   <chr>                    <dbl>
## 1 E000189-S014115-T000000      8
## 2 E001993-S018912-T000000     37
## 3 E002622-S014197-T000000     10
## 4 E003476-S051213-T000000      6
## 5 E003494                      3
## 6 E004454-S051301-T000000      3
## # ... with 45 more rows
## #
## # Edge Data: 124 x 3
##    from    to weight
##   <int> <int>  <int>
## 1     1     8      4
## 2     1    29      1
## 3     2     7      1
```

```
## # ... with 121 more rows
```

```
.infobox {
  padding: 1em 1em 1em 1em;
  margin-bottom: 2px;
  border: 2px solid orange;
  border-radius: 10px;
  background: #f5f5f5 5px center/3em no-repeat;
}
```

Note that the calculations are done one at a time. What difference would it make to your results if you filtered (for example based on a set of dates) and then calculated degree, rather than the other way around?

9.4.3 Summarising the network data

To work with your new network metrics, the data can be outputted to a standard R dataframe. Create a new dataframe by doing this, using the tidyverse function for creating dataframes, `as_tibble()`:

```
network_metrics_df = sample_tbl_graph %>%
  activate(nodes) %>% # make sure correct table is active
  mutate(degree = centrality_degree(weights = weight, mode = 'all')) %>% # calculate d
  mutate(between = centrality_betweenness(weights = weight,directed = F)) %>%  # calcu
  as_tibble() # turn the nodes table into a plain dataframe
```

```
## Warning in betweenness(graph = graph, v = V(graph), directed = directed, :
## 'nobigint' is deprecated since igraph 1.3 and will be removed in igraph 1.4
```

This new table can be sorted, totals counted, and so forth:

```
network_metrics_df %>% arrange(desc(degree))
```

```
## # A tibble: 248 x 3
##    name                    degree between
##    <chr>                    <dbl>   <dbl>
##  1 E004830-S019312-T000000    604   8048.
##  2 E022443-S042999-T000000    270  13404.
##  3 E600059-S012260-T000000    161   3223.
##  4 E013473                    150    235
##  5 E004654-S006979-T000000     91  11774.
##  6 E001993-S018912-T000000     72   6663.
##  7 E903376-S024822-T000000     57   1623
```

```
##  8 E008063                    51      0
##  9 E002620-S051125-T000000    47      0
## 10 E906453-S026312-T000000    47      0
## # ... with 238 more rows
```

9.5 Joining additional data

The value of working with a data model and tidygraph is that we can merge additional tables of data attributes to our nodes or edges. Earlier, we explained how important it is that every node in your data has a unique ID. The downside of this, particularly for humanities research, is that it's not easy to see who or what is behind a particular node. This can be solved by using a `join` command.

In a separate table, we have a dataset of attributes about this set of nodes, including place and dates of birth and death, and gender. Using the `join()` commands, we can merge this table to the network data, and use it to sort, filter (and later visualise) the data:

First, load the table of data using `read_csv`:

```
node_attributes = read_csv('node_attributes.csv')
```

```
## Rows: 248 Columns: 13
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (11): name, main_name, all_names, links, gender, roles_titles, wikidata_...
## dbl  (2): birth_year, death_year
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
node_attributes
```

```
## # A tibble: 248 x 13
##    name     main_~1 all_n~2 links birth~3 death~4 gender roles~5 wikid~6 occup~7
##    <chr>    <chr>   <chr>   <chr>   <dbl>   <dbl> <chr>  <chr>   <chr>   <chr>
##  1 E006019~ Dr. Ra~ Dr. Ra~ http~    1617    1688 male   theolo~ http:/~ theolo~
##  2 E011771~ Hyde, ~ Edward~ http~    1609    1674 male   judge;~ http:/~ judge;~
##  3 E015103~ Sir Ph~ Sir Ph~ http~    1607    1678 male   politi~ http:/~ politi~
##  4 E020134~ Taylor~ Silas ~ http~    1624    1678 male   compos~ http:/~ compos~
##  5 E014743~ Sir Ed~ Sir Ed~ http~    1608    1654 male   econom~ http:/~ econom~
##  6 E020485~ Sir Th~ Sir Th~ http~    1612    1685 male   naval ~ http:/~ naval ~
##  7 E004654~ Charle~ Charle~ http~    1630    1685 male   sovere~ http:/~ sovere~
##  8 E007992~ Heneag~ Heneag~ http~    1628    1689 male   diplom~ http:/~ diplom~
```

```
##  9 E007902~ John F~ John F~ http~    1625    1686 male   priest  http:/~ priest
## 10 S035587  Philad~ Philad~ http~    1612    1665 female politi~ http:/~ politi~
## # ... with 238 more rows, 3 more variables: place_of_birth <chr>,
## #   place_of_death <chr>, politician <chr>, and abbreviated variable names
## #   1: main_name, 2: all_names, 3: birth_year, 4: death_year, 5: roles_titles,
## #   6: wikidata_item, 7: occupations
```

This table contains further information about the nodes, each of which are identified by their unique ID. This can be joined to the network object using `join()` commands:

```
sample_tbl_graph %>%
  left_join(node_attributes, by = 'name')
```

```
## # A tbl_graph: 248 nodes and 364 edges
## #
## # A directed simple graph with 6 components
## #
## # Node Data: 248 x 13 (active)
##   name  main_n~ all_na~ links birth_~ death_~ gender roles_~ wikida~ occupa~
##   <chr> <chr>   <chr>   <chr>   <dbl>   <dbl> <chr>  <chr>   <chr>   <chr>
## 1 E000~ Dr. La~ Dr. La~ http~    1632    1703 male   chapla~ http:/~ chapla~
## 2 E000~ Edward~ Edward~ http~    1638    1697 male   astron~ http:/~ astron~
## 3 E000~ Dr. Ri~ Dr. Ri~ http~    1619    1681 male   priest  http:/~ priest
## 4 E000~ Solms-~ Prince~ http~    1602    1675 female art co~ http:/~ art co~
## 5 E000~ Arthur~ Arthur~ http~    1614    1686 male   politi~ http:/~ politi~
## 6 E000~ Sir Jo~ Sir Jo~ http~    1603    1671 male   politi~ http:/~ politi~
## # ... with 242 more rows, and 3 more variables: place_of_birth <chr>,
## #   place_of_death <chr>, politician <chr>
## #
## # Edge Data: 364 x 3
##    from    to weight
##   <int> <int>  <int>
## 1     1    89      1
## 2     2    22      4
## 3     2   199      1
## # ... with 361 more rows
```

Using this approach you can now make subsets of the network, and calculate global or node-level statistics for these. This example would return a network containing only individuals with politician listed as one of their occupations, for example:

```
sample_tbl_graph %>%
  left_join(node_attributes, by = 'name') %>% # first join the attributes table again
  filter(str_detect(occupations, "politician")) %>% # returns any row with the string 'politician
  mutate(degree = centrality_degree(weights = weight, mode = 'all')) # calculate network metrics
```

```
## # A tbl_graph: 94 nodes and 106 edges
## #
## # A directed simple graph with 18 components
## #
## # Node Data: 94 x 14 (active)
##   name  main_n~ all_na~ links birth_~ death_~ gender roles_~ wikida~ occupa~
##   <chr> <chr>   <chr>   <chr>   <dbl>   <dbl> <chr>  <chr>   <chr>   <chr>
## 1 E000~ Arthur~ Arthur~ http~    1614    1686 male   politi~ http:/~ politi~
## 2 E000~ Sir Jo~ Sir Jo~ http~    1603    1671 male   politi~ http:/~ politi~
## 3 E001~ Henry ~ Henry ~ http~    1618    1685 male   politi~ http:/~ politi~
## 4 E002~ Willia~ Willia~ http~    1649    1717 male   politi~ http:/~ politi~
## 5 E002~ Edward~ Edward~ http~    1623    1683 male   politi~ http:/~ politi~
## 6 E002~ Roger ~ Roger ~ http~    1621    1679 male   writer~ http:/~ writer~
## # ... with 88 more rows, and 4 more variables: place_of_birth <chr>,
## #   place_of_death <chr>, politician <chr>, degree <dbl>
## #
## # Edge Data: 106 x 3
##    from    to weight
##   <int> <int>  <int>
## 1     1    33      1
## 2     2    33      1
## 3     3    10      1
## # ... with 103 more rows
```

Node attributes could be added and used to filter at several steps: before the network is created, and before or after network metrics are calculated. What differences might these make?

Make sure you know at which stage you are calculating network metrics. If you calculate them after filtering, you'll get a set of metrics based on a new, subsetted network.

This new table can be outputted as a dataframe, as above. Here we use this to calculate the highest-degree nodes from the 'polticians network', keep their real names, and sort in descending order of degree:

```
sample_tbl_graph %>%
  left_join(node_attributes, by = 'name') %>% # first join the attributes table again
  filter(str_detect(occupations, "politician")) %>% # returns any row containing the string 'poli
  mutate(degree = centrality_degree(weights = weight, mode = 'all'))  %>%   # calculate network m
```

```
as_tibble() %>%
  arrange(desc(degree)) %>% select(name, main_name, degree)
```

```
## # A tibble: 94 x 3
##    name                   main_name                                      degree
##    <chr>                  <chr>                                           <dbl>
##  1 E022443-S042999-T000000 Williamson, Joseph (Sir)                         174
##  2 E001993-S018912-T000000 Henry Bennet, Earl of Arlington                   54
##  3 E004654-S006979-T000000 Charles II, King of England, Scotland, and Ir~    53
##  4 E002620-S051125-T000000 William Blathwaite                                47
##  5 E002622-S014197-T000000 Edward Conway, Earl of Conway                     30
##  6 S016294                 Francis, Lord Aungier                             28
##  7 E903376-S024822-T000000 Witt, Johan de                                    28
##  8 S047442                 Thomas Belasyse, Earl of Fauconberg               26
##  9 E921955-S041743-T000000 Sir George Rawdon                                 25
## 10 E021031-S049939-T000000 Van Beuningen                                     21
## # ... with 84 more rows
```

9.6 Social Network Analysis with R

This last part goes through some of the functions which are more related to the 'social network analysis'. Many of these are accessed directly through the package `igraph`.

9.6.1 Transitivity, Triads, structural balance

As we learned in the last lesson, counting the ratio of completed triangles in a network is a good way to understand its structure. This ratio is known as the **clustering coefficient**, and there are two types: **global**, which measures the ratio of complete and not complete triangles in the entire network, and **local**, which measures the complete triangles for each individual node. This measurement is also known as **transitivity**.

Use the following command to calculate the global clustering, or transitivity of the network. It returns a single number, which is just the ratio of completed triangles to the total possible number of triangles in the graph:

```
sample_tbl_graph %>% transitivity(type = 'global')
```

```
## [1] 0.02453653
```

Use the following to measure the local clustering coefficient for each node in the graph. It returns a vector of numbers, one for each node:

```
sample_tbl_graph %>% transitivity(type = 'local')
```

```
##   [1]         NaN 0.095238095 1.000000000 1.000000000 0.000000000         NaN
##   [7]         NaN         NaN         NaN 0.030252101         NaN 0.000000000
##  [13]         NaN         NaN         NaN 0.100000000 0.000000000 0.000000000
##  [19]         NaN 1.000000000 0.020390071 0.010796221         NaN 0.333333333
##  [25] 1.000000000 0.000000000 1.000000000         NaN         NaN         NaN
##  [31]         NaN 0.166666667         NaN         NaN 0.000000000         NaN
##  [37]         NaN 0.333333333 1.000000000         NaN         NaN         NaN
##  [43] 0.000000000         NaN 0.025000000         NaN         NaN 0.000000000
##  [49] 0.200000000         NaN 0.000000000 1.000000000         NaN 0.000000000
##  [55] 0.333333333         NaN         NaN         NaN 0.333333333         NaN
##  [61] 0.000000000         NaN 0.000000000         NaN         NaN         NaN
##  [67]         NaN         NaN         NaN         NaN         NaN 0.500000000
##  [73] 1.000000000 0.142857143 1.000000000         NaN 0.000000000         NaN
##  [79]         NaN         NaN         NaN         NaN         NaN         NaN
##  [85] 0.000000000 1.000000000 0.000000000 0.000000000 0.009937888         NaN
##  [91] 0.000000000         NaN 0.000000000 0.333333333 1.000000000         NaN
##  [97]         NaN         NaN         NaN         NaN         NaN 0.000000000
## [103]         NaN 1.000000000 0.000000000 0.090909091 0.333333333 1.000000000
## [109] 0.333333333 0.000000000         NaN 0.000000000         NaN         NaN
## [115]         NaN         NaN         NaN         NaN 0.000000000         NaN
## [121]         NaN         NaN         NaN         NaN         NaN         NaN
## [127]         NaN         NaN 0.000000000         NaN         NaN         NaN
## [133] 1.000000000         NaN         NaN         NaN 1.000000000         NaN
## [139]         NaN         NaN         NaN         NaN         NaN 1.000000000
## [145]         NaN         NaN         NaN         NaN         NaN         NaN
## [151] 0.000000000 0.000000000         NaN 0.000000000         NaN         NaN
## [157]         NaN         NaN 0.000000000         NaN 1.000000000         NaN
## [163]         NaN         NaN         NaN         NaN         NaN 1.000000000
## [169] 1.000000000         NaN         NaN         NaN         NaN         NaN
## [175]         NaN 1.000000000 0.333333333         NaN         NaN         NaN
## [181]         NaN         NaN 0.000000000         NaN         NaN 0.500000000
## [187] 1.000000000         NaN         NaN         NaN         NaN         NaN
## [193] 1.000000000 1.000000000 1.000000000         NaN         NaN         NaN
## [199]         NaN         NaN 0.035714286         NaN         NaN         NaN
## [205] 0.000000000         NaN         NaN         NaN         NaN 0.000000000
## [211]         NaN         NaN         NaN         NaN         NaN         NaN
## [217]         NaN         NaN         NaN         NaN         NaN         NaN
## [223] 0.000000000         NaN         NaN         NaN         NaN         NaN
## [229]         NaN         NaN         NaN         NaN         NaN         NaN
## [235]         NaN 0.333333333 0.000000000         NaN         NaN         NaN
## [241]         NaN         NaN         NaN         NaN         NaN         NaN
## [247]         NaN         NaN
```

Anoter related measurement is the average local clustering, calculated using the following:

```
sample_tbl_graph %>% transitivity(type = 'average')
```

```
## [1] 0.361597
```

We can also calculate the total reciprocity of a graph (the proportion of links which are reciprocated):

```
sample_tbl_graph %>% reciprocity()
```

```
## [1] 0.2912088
```

We can also calculate the 'triad census', the number of each type of triad in a directed network. First, run the `triad_census()` function, and turn it into a dataframe with one column. Next add the correct triad codes as a second column.

```
census = sample_tbl_graph %>% triad_census() %>% as_tibble()

census$type <- c("003","012","102","021D","021U","021C","111D","111U",
                 "030T","030C","201","120D","120U","120C","210","300")

census
```

```
## # A tibble: 16 x 2
##      value type
##      <dbl> <chr>
##  1 2440447 003
##  2   54274 012
##  3   11363 102
##  4     550 021D
##  5    2892 021U
##  6     653 021C
##  7     636 111D
##  8     255 111U
##  9      30 030T
## 10       3 030C
## 11     381 201
## 12       0 120D
## 13       4 120U
## 14       2 120C
## 15       2 210
## 16       4 300
```

You can now connect these codes to the triad diagrams from the last chapter, and use them to infer things about the structure and formation of the network.

Imagine that this is a complete network. Why, for example, might 021U (A and B both send a letter to C, but there's not reciprocation, and they are not connected to each other) be such a common triad?

The answer lies in the origin of this network, as a letter archive. Because a letter archive contains mostly incoming letters to an ego node, many of the triads are pairs of nodes who both have an incoming connection to one of these ego nodes, but are not themselves connected.

Another set of metrics are to do with groups found in a network. First, components. A component is a group of nodes which are connected together through a path in the network. The `components()` function returns a list which includes a named vector with a component number for each node, along with the number of nodes in each component, and the total number of components. Looking at this list with `glimpse` tells us that there are six components, the largest being 237 nodes and the smallest has 2.

```
sample_tbl_graph %>% igraph::components() %>% glimpse()
```

```
## List of 3
##  $ membership: Named num [1:248] 1 1 1 1 1 1 1 1 1 1 ...
##   ..- attr(*, "names")= chr [1:248] "E000145-S012650-T000000" "E000189-S014115-T000000" "E0003
##  $ csize     : num [1:6] 237 2 3 2 2 2
##  $ no        : int 6
```

Another global metric we can calculate is the number of **cliques** in a network. A clique is a fully connected set of nodes. We can calculate the total number of cliques with `clique_num`:

```
## [1] 4
```

9.7 Louvain community detection

Another way to look at groups in networks is to use community detection. As explained in the previous chapter, community detection attempts to find groups of nodes which are more connected to each other than they are to nodes outside the group. Unlike components or cliques, communities can be found using a number of methods. Here, we'll use the Louvain algorithm, which was introduced in the previous lesson.

To make it easy to use the results, we'll calculate the community for each node using tidygraph. This particular implementation of the algorithm only works

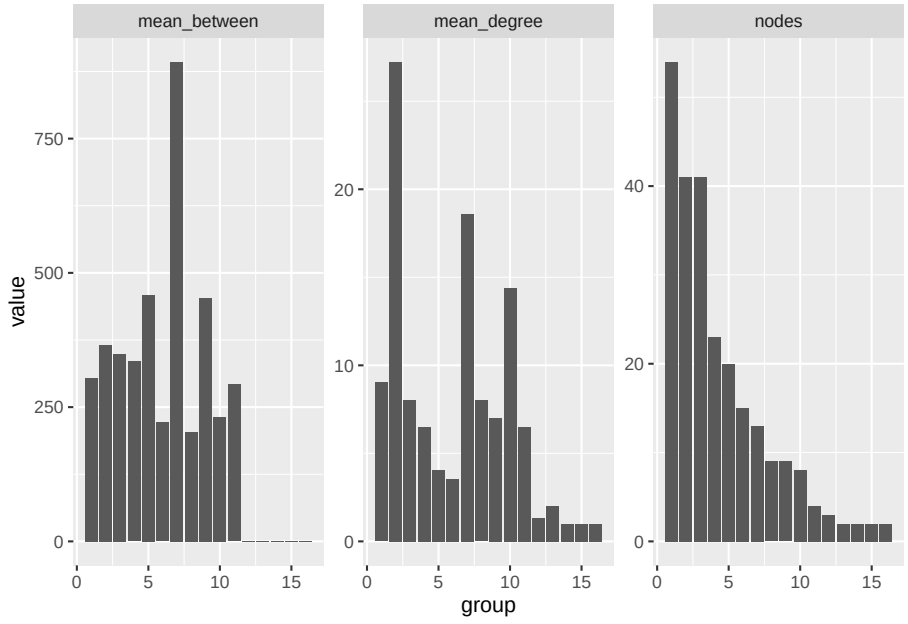
for undirected graphs, so we have to create our network again from scratch, this time specifying that it's **undirected**.

```
sample_tbl_graph_u = edge_list %>%
  as_tbl_graph(directed = FALSE)

sample_tbl_graph_u %>%
  mutate(group = group_louvain())
```

```
## # A tbl_graph: 248 nodes and 364 edges
## #
## # An undirected multigraph with 6 components
## #
## # Node Data: 248 x 2 (active)
##   name                    group
##   <chr>                   <int>
## 1 E000145-S012650-T000000     1
## 2 E000189-S014115-T000000     2
## 3 E000312-S012741-T000000     3
## 4 E000393-S036175-T000000     5
## 5 E000520-S001769-T000000     7
## 6 E000799-S042396-T000000     1
## # ... with 242 more rows
## #
## # Edge Data: 364 x 3
##    from    to weight
##   <int> <int>  <int>
## 1     1    89      1
## 2     2    22      4
## 3     2   199      1
## # ... with 361 more rows
```

By turning this information into a dataframe, we can get some basic statistics on the groups. By adding this information to the node information we collected earlier, we can get a summary of the average degree and betweenness scores for each of the detected communities:

```
sample_tbl_graph_u %>%
  mutate(group = group_louvain())%>%
  activate(nodes) %>% # make sure correct table is active
  mutate(degree = centrality_degree(weights = weight, mode = 'all')) %>% # calculate d
  mutate(between = centrality_betweenness(weights = weight,directed = F)) %>%  # calcu
  as_tibble()  %>% group_by(group) %>%
  summarise(mean_degree = mean(degree,na.rm = TRUE), mean_between = mean(between, na.r
  ggplot() + geom_col(aes(x = group, y = value)) + facet_wrap(~name, scales = 'free')
```



9.8 Reading for next week

Week 4: Ahnert et. al. (2021). *The Network Turn: Changing Perspectives in the Humanities* (Elements in Publishing and Book Culture). **Chapter 5, 'Quantifying Culture' (https://www.cambridge.org/core/elements/network-turn/CC38F2EA9F51A6D1AFCB7E005218BBE5)**

Questions:

- What do the authors mean by the 'abstraction' of humanities data? What are the advantages and disadvantages of doing so?
- What are the assumptions made by the authors in the 'toy example' on page 76? What other ways could they have treated this data?
- What is the 'trade-off' mentioned on page 78?
- What do the authors suggest might be the best way to 'solve' the skills gap between those with quantitative and those with traditional humanities skills?

Chapter 10

Week 5, Class 1: Visualising Networks with ggraph, I

10.1 Network visualisations

Many analyses of networks rely on visualisation. Graphing a network is particularly useful for *descriptive data analysis*, as a way of describing the overall structure of graph, and *exploratory data analysis,* where it's used as a sort of map to understand its various components, and to help spot patterns or interesting features by eye. These visualisations are most often the points and lines type diagrams which we have used throughout this book, but there are also a number of other ways they can be visualised (which often might be more useful).

It's important to note that there is nothing inherently spatial about a graph: it is simply a record of connections between nodes. When we choose to represent it visually, we have to make decisions as to its form and how precisely its nodes and edges are placed in 2D (or even 3D) space.

From early social network research, researchers tried to manually visualise these graphs in meaningful ways, for example by placing closely-connected clusters together (and away from other clusters), placing important or highly-connected nodes towards the centre, or minimising the number of edge (line) crossings. With large networks, today this process is usually carried out using algorithms to work out the node placements.

10.1.1 Force-directed network visualisations

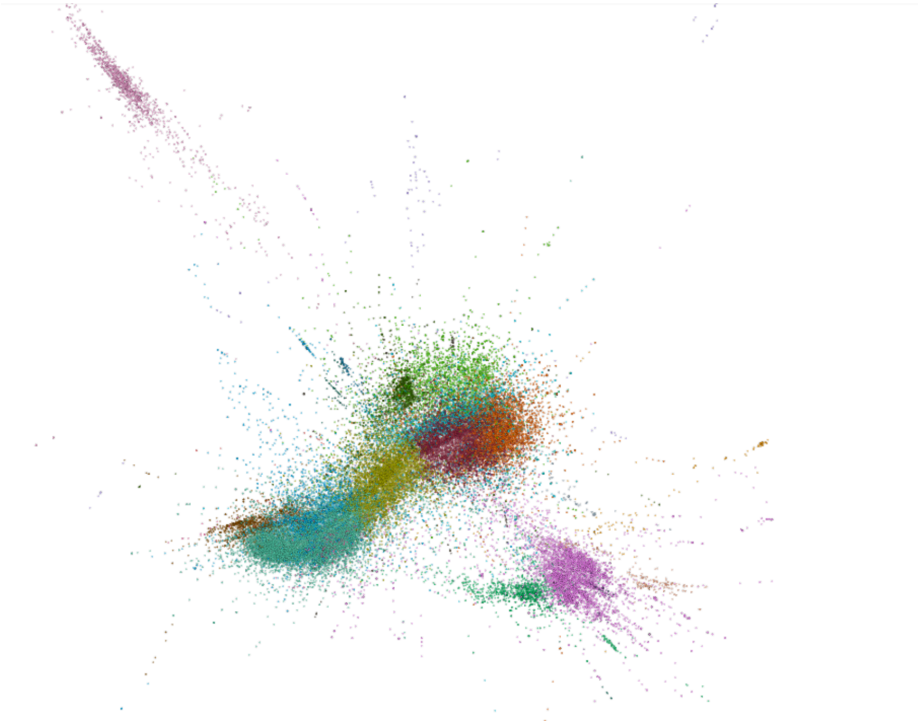
The most common family of these algorithms are 'force-directed', meaning they use a simulation of physical forces in order to create sensible placements of

nodes. One of the most common of these is the Fruchterman-Reingold layout, which treats edges like a spring. Nodes which share an edge are attracted to each other using a spring-like force; every pair of nodes in the system also has a repulsive force. The algorithm simulates this physical system and stops when the distances between the nodes means that the system is in equilibrium.

A good force-directed graph can actually convey a great deal of information about a network. The paper for this week's reading argues that the ambiguity of a force-directed graph can actually make them very useful for exploratory data analysis, when they are interpreted correctly.

10.1.1.1 Reading a network graph

Using a similar approach to that paper, consider this network of book publishers from the eighteenth century:



This is a large network, consisting of tens of thousands of nodes and millions of edges. The nodes are coloured by 'community', meaning tthat each colour of nodes is more densely connected to each other than to the nodes of other colours. Despite its size, some structure can be seen. We could describe it as the following:

- Spatially (ignoring the colours), there are three main sections: a large

central section, and two smaller sections, one to the bottom-right and another, even smaller, to the top-left of the main section.

- This main section is shaped a bit like a hairbrush: it has an elongated 'handle', and an attached 'brush' at the top.
- The colours are distinct, meaning that the force-directed graph did a good job in replicating the clusters found by the community detection algorithm.

To understand why it might have this shape, we looked into the nodes in each of its clusters.

The first thing we noted was that the almost separate 'islands' were groups of Dublin (larger and closer island) and US publishers.

The main section is made up of clusters of different time periods. Each time period is connected to the others mainly by a short edge, meaning that (for example) the 1700 - 1720 cluster is much more connected to the 1720 - 1740 cluster than to the 1780 - 1800 cluster, for example. This is typical in a long, multi-generational dataset like this.

The handle is mostly London-based publishers, and the 'brush' is a group of Scottish publishers. These Scottish publishers are mostly connected to the later London clusters (red/pink). Scottish publishers are much closer to the London core than either Dublin or US.

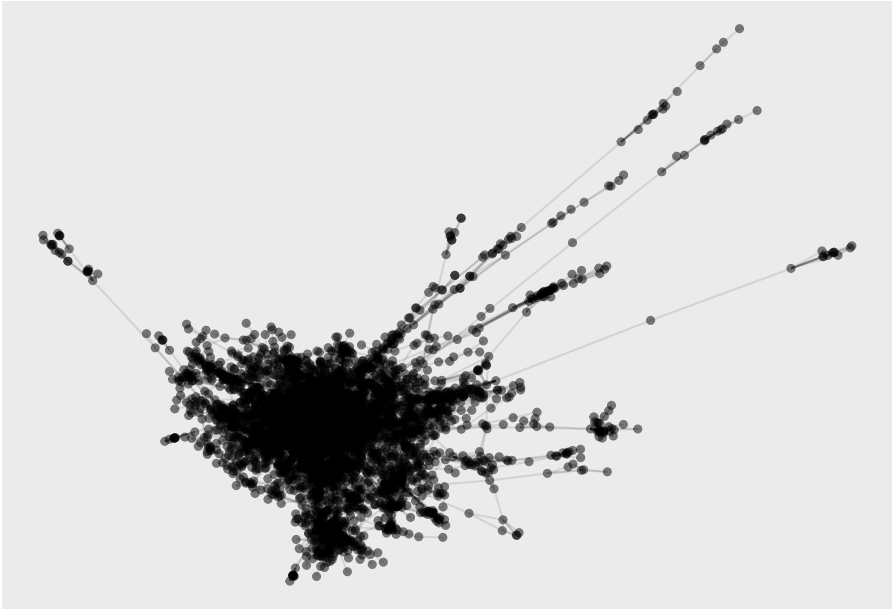
Some inferences we might make from this diagram.

- US and Dublin publishers were very separate from a London/Edinburgh publishing axis.
- For London publishers, the strongest pull is temporal rather than any other aspect.
- Over time, the closeness of the Edinburgh and London publisher networks grew.

When used correctly and with more knowledge about a network, force-directed diagrams like this can help to spot distinct clusters, structural 'holes', and other features of a network.

10.1.1.2 Avoiding the dreaded 'hairball'

Visualising large networks using these methods can often result in a large tangled mass of nodes and edges, known pejoratively as a 'hairball'. This is particularly true of large graphs without much of a tendency to cluster together, such as this graph of Facebook page networks[Rozemberczki et al., 2021].



These graphs have limited use, even as exploratory data. There are some ways to mitigate against them, however:

- Consider filtering the network, as in the previous steps. Make sure you're aware of the consequences of filtering before and after you calculate network metrics, however.
- For large graphs, software such as Gephi, because it gives a real-time feedback of a network visualisation, can be useful, rather than purely using a programming language such as R.
- Think about whether a network diagram (or a network model at all) is the best way to represent or display your data. Could you arrive at the same conclusions with a simpler data analysis and output, such as a bar chart?

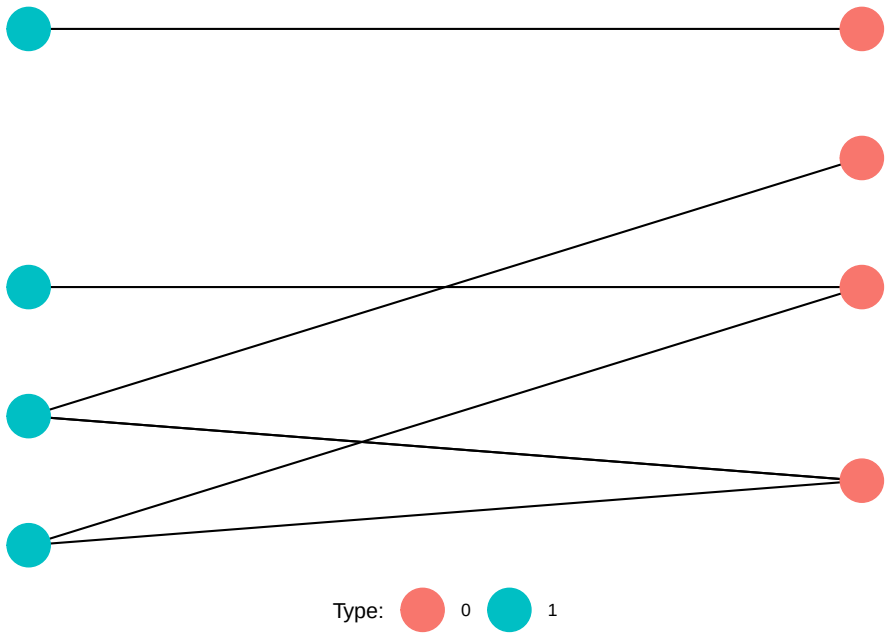
10.1.2 Other Network Visualisations

There are many other ways besides a 'force-directed' graph to visualise a network.

10.1.2.1 Bipartite graph

Some network types are particularly suited to other visualisation methods. Bipartite networks, for example, are often visualised so that the nodes are placed

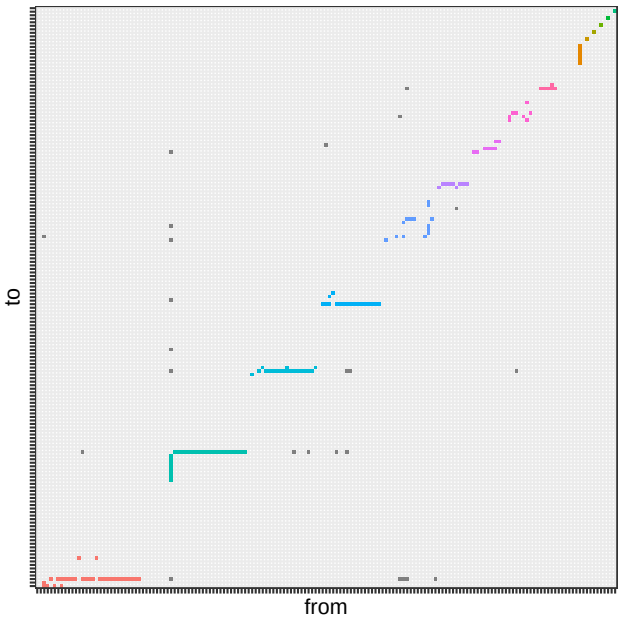
in two rows, according to their types. The positions within the rows are then determined by an algorithm designed to minimise edge crossings.



10.1.2.2 Adjacency Matrix

One popular alternative to a network diagram is an *adjacency matrix*. In this case, the x and y axes contain each name in the network. A filled square is drawn for each edge, where they intersect.

This method can be particularly useful for small, dense networks.



10.2 Network Visualisations with R and ggraph.

Visualisations like the examples above can be created with another R package, called `ggraph`. This uses the same basic syntax as the plotting library `ggplot2`, we used in an earlier lesson, but adds some special functions to visualise networks.

We'll spend today and the next class on learning how to visualise graphs with R and `ggraph`. First, we'll learn how to make a basic network diagram of nodes and edges, how to change the layout algorithm, and how to set the node and edge sizes. In the next class, we'll customise it with colour, arrows, text labels, and so forth.

Using this ggraph workflow, a network diagram consists of a number of elements:

- The network itself, in the form of a network object, either a `tbl_graph` or an `igraph` network.
- The `ggraph` function, which tells R to start making a network plot.
- The relevant `geom_` layers, which tell ggraph to map the network to nodes and edges
- Aesthetics, which tell ggraph to map particular visual attributes (size, colour and so forth) to specified values.

To begin with, create a `tbl_graph` network object from an edge list, using the same data and code as last week:

```
library(tidygraph)
library(tidyverse)
library(ggraph)
letters = read_csv("letter_data.csv", col_types = cols(.default = "c"))

edge_list = letters %>%
  group_by(from_id, to_id) %>%
  tally(name = 'weight')

sample_tbl_graph = edge_list %>%
  as_tbl_graph()
```

10.2.1 The ggraph function

To begin making the network diagram, we start with the `ggraph()` function. This tells R to begin drawing a graph. It has optional arguments: for example, you can set the layout to something other than the default using the argument `layout =`. There are a large number of layouts available to use, some of which will be shown below.

```
sample_tbl_graph %>%
  ggraph()
```

```
## Using `stress` as default layout
```

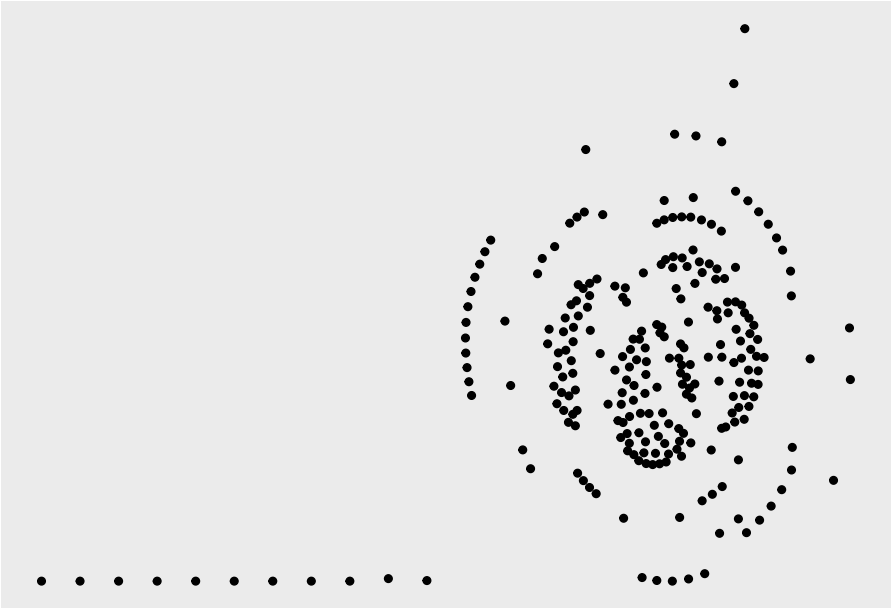
For now, you'll notice that it doesn't draw anything other than a blank grey background.

To draw the nodes and edges of the graph, we'll use what are known as 'geoms'. These are a family of functions which map the data to relevant elements. They all begin with `geom_`. These are added as layers to the ggraph object using `+` followed by the relevant geom.

- First, adding `+ geom_node_point()` as a layer will draw the nodes of your network as points. The nodes are positioned according to the chosen (or default) layout algorithm:

```
sample_tbl_graph %>%
  ggraph() +
  geom_node_point()
```

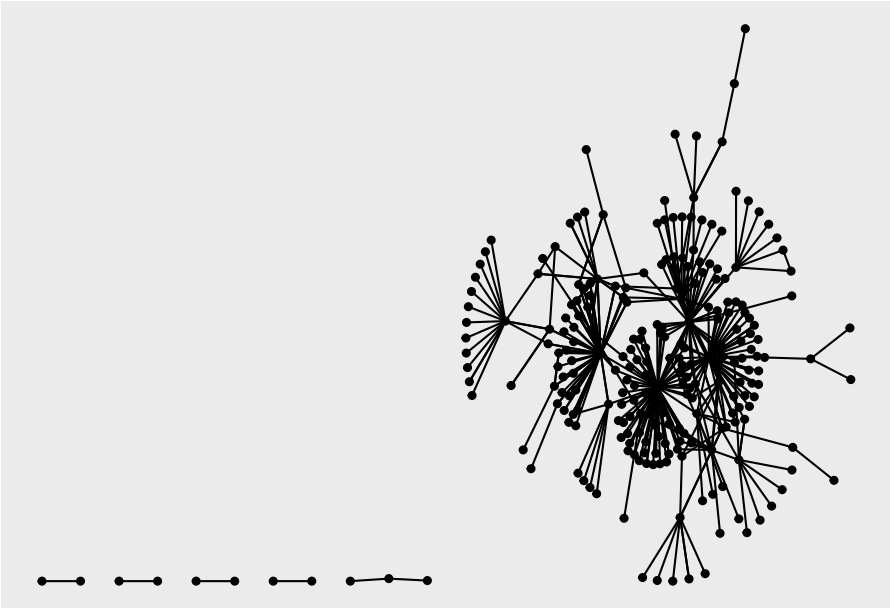
```
## Using `stress` as default layout
```



- Next, `geom_edge_link()` will drawn the edges as connecting lines:

```
sample_tbl_graph %>%
  ggraph() +
  geom_node_point() +
  geom_edge_link()
```

```
## Using `stress` as default layout
```



Adjusting visual attributes.

There are many additional ways you can manipulate the visual appearance of your nodes and edges, by adding color, size, shape, arrows, and so forth.

In ggraph (and ggplot), visual elements can be manipulated in two ways:

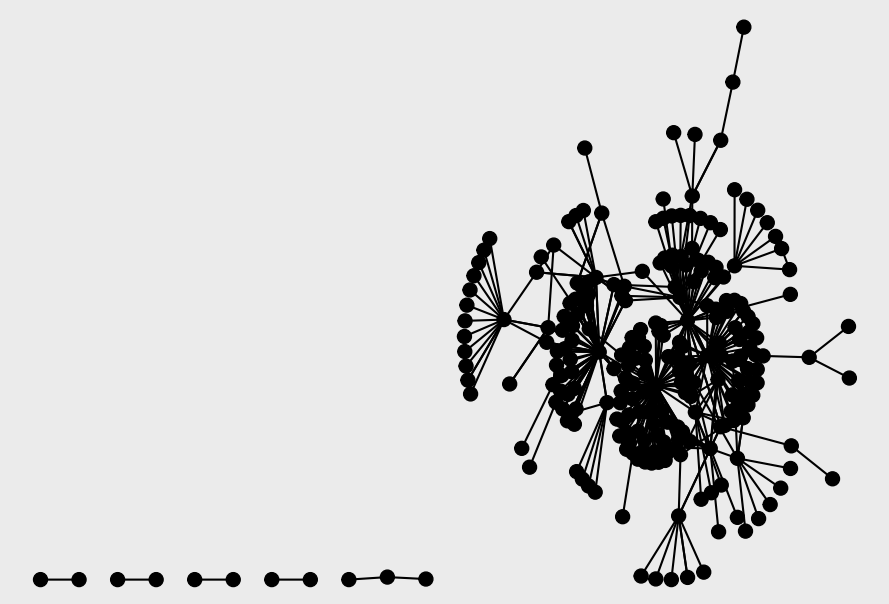
- You can specify an absolute value for an aesthetic, for example setting the size of all nodes to value 3
- You can specify that a value is mapped to a variable in your data. This could be some metric of your network calculated in a previous step, for example setting the size to be proportional to the degree score of a node, or it could be some external data, wuch as setting the colour to indicate the node's occupation.

This is all done within the relevant geom. We can check to see which aesthetics are recognised by a particular geom by typing `?` followed by the function in the console and pressing return, i.e `?geom_node_point()`. This tells us that it recognises `alpha, colour, fill, shape, size, stroke, and filter`.

To set a one of these visual attributes (or aesthetic) to an absolute value, simply set it within the parentheses of `geom_node_point()`:

```
sample_tbl_graph %>%
  ggraph() +
  geom_node_point(size = 3) + # we specified that size should be set at the value 3
  geom_edge_link()
```

```
## Using `stress` as default layout
```



We can also map aesthetics to a particular *variable* in the data. To do this, the value should exist as a column in the nodes table of the `tbl_graph` object. For example, we can calculate the degree score for each node:

```
sample_tbl_graph %>%
  mutate(degree = centrality_degree())
```

```
## # A tbl_graph: 248 nodes and 364 edges
## #
## # A directed simple graph with 6 components
## #
## # Node Data: 248 x 2 (active)
##   name                    degree
##   <chr>                    <dbl>
## 1 E000145-S012650-T000000      1
## 2 E000189-S014115-T000000      3
## 3 E000312-S012741-T000000      1
## 4 E000393-S036175-T000000      1
## 5 E000520-S001769-T000000      2
## 6 E000799-S042396-T000000      1
## # ... with 242 more rows
## #
## # Edge Data: 364 x 3
```

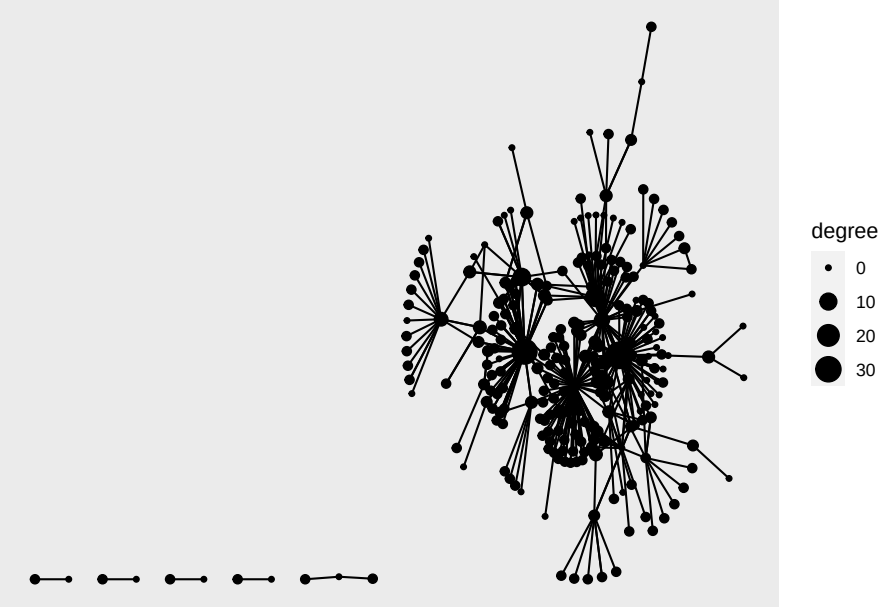
```
##    from    to weight
##   <int> <int>  <int>
## 1     1    89      1
## 2     2    22      4
## 3     2   199      1
## # ... with 361 more rows
```

And it is added as an additional column in the first table.

To map this column to be proportional to the node size, we do so within the function `aes()`, within the relevant geom:

```
sample_tbl_graph %>%
  mutate(degree = centrality_degree()) %>% # calculate a value for degree
  ggraph() +
  geom_edge_link() + # switch around the order of the nodes and edges as it's easier t
  geom_node_point(aes(size = degree)) # map size to degree score
```

```
## Using `stress` as default layout
```



You'll notice that ggraph has added a **legend**: a guide to help us read exactly how the various sizes of the nodes map to the degree score.

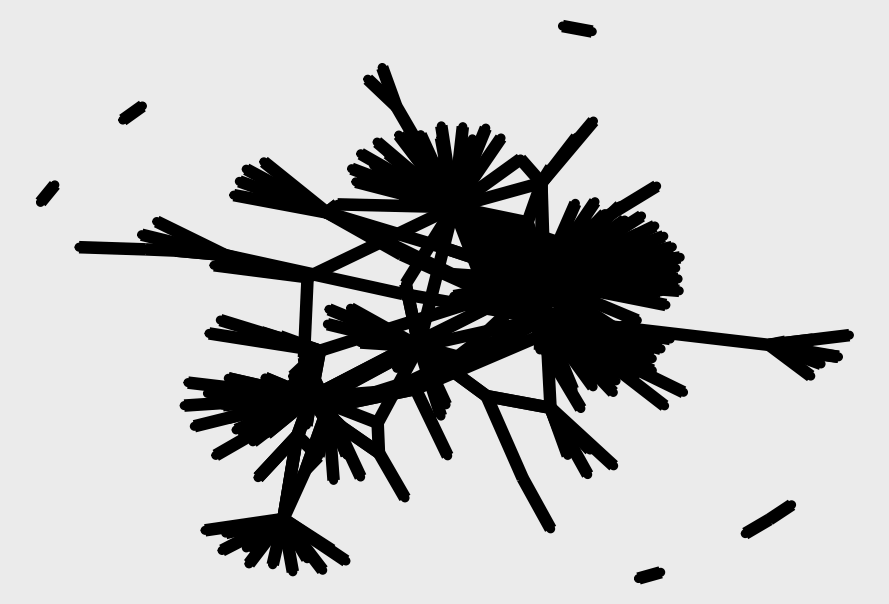
You can experiment in the notebook with the other aesthetics, including `alpha` and `shape`. You'll notice that some of them cannot be mapped to a numerical (or continuous variable). We'll look into this in the next class.

10.3 Adjusting edge aesthetics

The visual appearance of the edges can also be adjusted, using the same syntax.

We can set the thickness of the edges using the `width` aesthetic:

```
sample_tbl_graph %>%
  mutate(degree = centrality_degree()) %>%
  ggraph('fr') +
  geom_edge_link(width = 3) +
  geom_node_point()
```



As with nodes, these aesthetics can be mapped to particular values. In this case, the variable to be mapped should exist as a column in the edges table in the `tbl_graph` object. If we look at the object, we can see that there is already a 'weight' column which we created when making the graph:

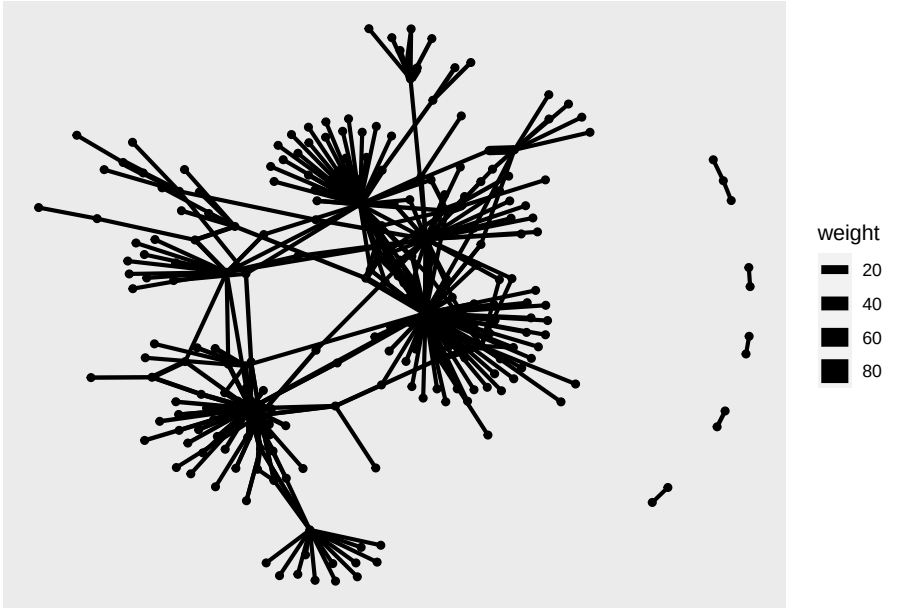
```
sample_tbl_graph
```

```
## # A tbl_graph: 248 nodes and 364 edges
## #
## # A directed simple graph with 6 components
## #
## # Node Data: 248 x 1 (active)
```

```
##   name
##   <chr>
## 1 E000145-S012650-T000000
## 2 E000189-S014115-T000000
## 3 E000312-S012741-T000000
## 4 E000393-S036175-T000000
## 5 E000520-S001769-T000000
## 6 E000799-S042396-T000000
## # ... with 242 more rows
## #
## # Edge Data: 364 x 3
##    from    to weight
##   <int> <int>  <int>
## 1     1    89      1
## 2     2    22      4
## 3     2   199      1
## # ... with 361 more rows
```

Again, this can be mapped using the same syntax as with nodes. Set the `width` aesthetic to the `weight` variable within the `aes()` command:

```
sample_tbl_graph %>%
  mutate(degree = centrality_degree()) %>%
  ggraph('fr') +
  geom_edge_link(aes(width = weight)) +
  geom_node_point()
```



10.4 Layout algorithms

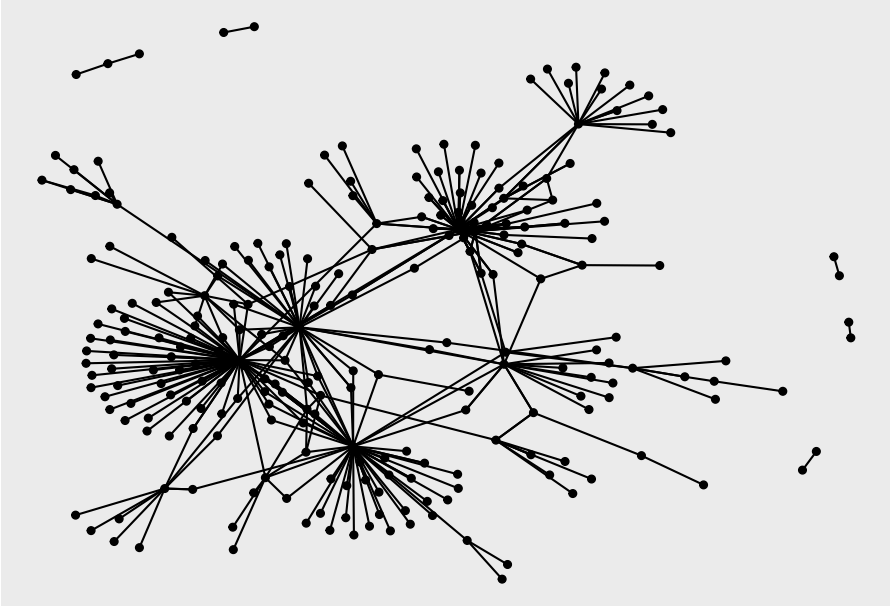
Igraph has a large number of graph layouts. Depending on your network size and structure, different ones may be appropriate. Where possible, it is a good idea to add some other information, for example group membership or simply node names to your visualisation, at least temporarily. That way you can check to see if the visual structures, such as clusters, centre-periphery, and orientation, make any sense. Here are the most common graph layouts:

10.4.0.1 Fruchterman-Reingold

A popular 'force-directed' layout, which treats the connections between nodes as springs.

Add to ggraph using `layout = 'fr'`.

```
sample_tbl_graph %>%
  ggraph(layout = 'fr') +
  geom_edge_link() +
  geom_node_point()
```

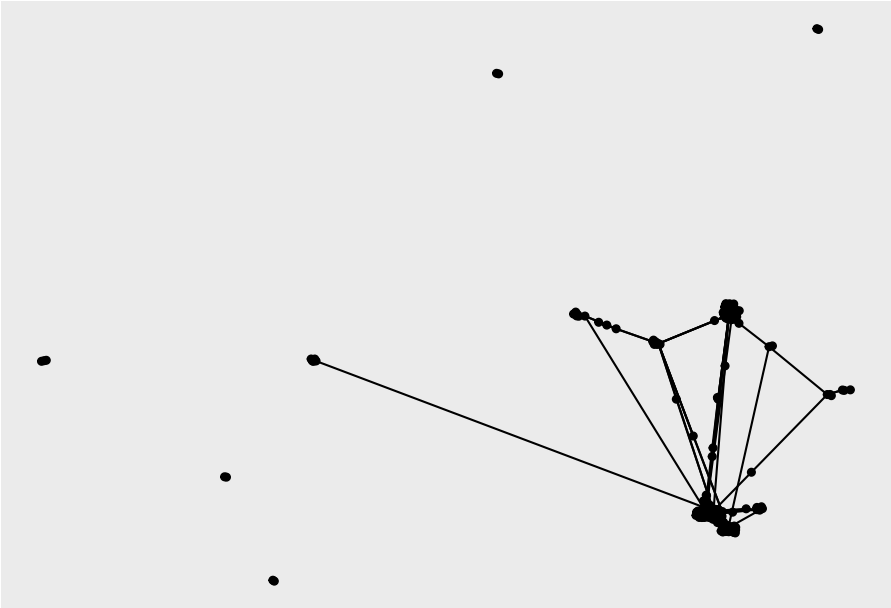


10.4.0.2 DRL

Another force-directed layout, often used with larger graphs.

Add to ggraph using `layout = 'drl'`.

```
sample_tbl_graph %>%
  ggraph(layout = 'drl') +
  geom_edge_link() +
  geom_node_point()
```

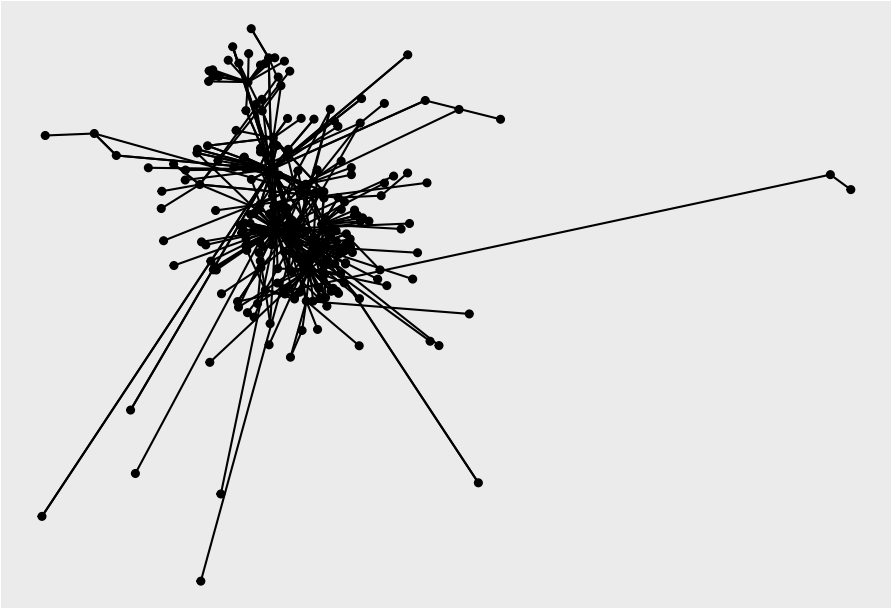


10.4.0.3 Kamada-Kawai

Another force-directed layout. This doesn't work well with 'disconnected graphs: below I have added a function to only include the main connected component of the network.

Add using `layout = 'kk'`

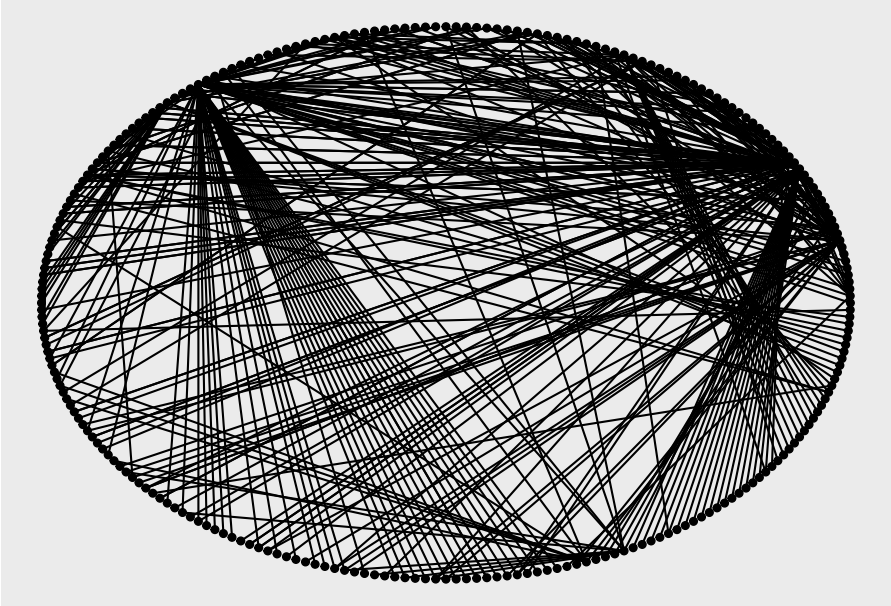
```
sample_tbl_graph %>%
  mutate(component = group_components()) %>%
  filter(component == 1) %>%
  ggraph('kk') +
  geom_edge_link() +
  geom_node_point()
```



10.4.0.4 Circle

Places nodes in a circle, in order:

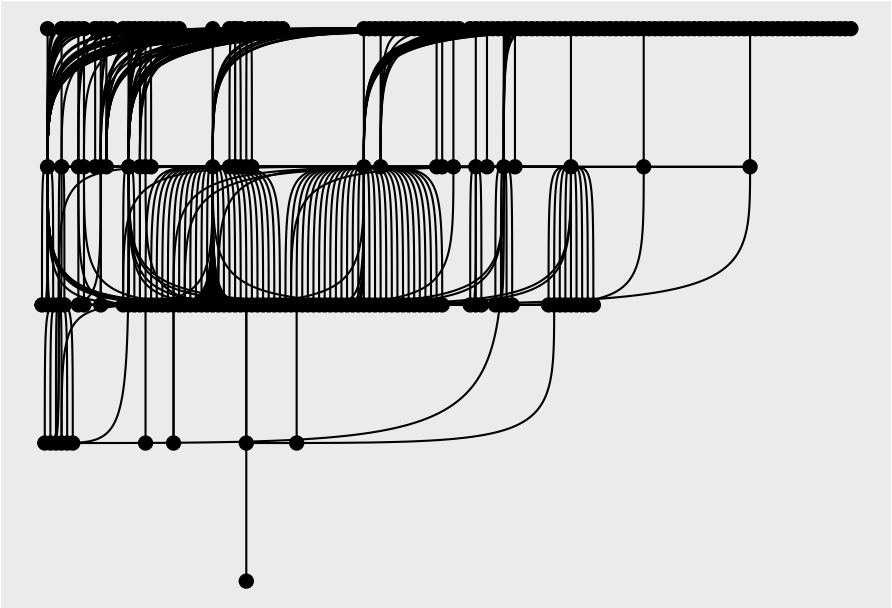
```
sample_tbl_graph %>%
  ggraph('circle') +
  geom_edge_link() +
  geom_node_point()
```



10.4.0.5 Tree

For hierarchical graphs. Places nodes in a tree diagram, according to their hierarchy.

```
sample_tbl_graph %>%
  ggraph('tree') +
  geom_edge_bend() + # here we use a different geom_edge_ more appropriate to tree graphs
  geom_node_point(size = 3)
```



It is worth checking some of these out, particularly if you have a specific network structure, for example a hierarchy, or if you have a bipartite network.

Also note there are lots of other ways to visualise a network, for example by making a heatmap from an adjacency matrix. It's worth experimenting a bit with these (and explaining choices in your final project).

Chapter 11

Week 5, Class 2: Visualising Networks with ggraph, II

11.1 Discreet versus continuous variables

A very important concept to bear in mind when making visualisations is the differences in the way **discrete** and **continuous** value are treated. For the purposes of visualisation we can think of the differences like this:

- A **continuous** value is something you can measure, for example height. There are infinite values that a height can be.
- A **discrete** value is one where the values are one of a limited set of set values. For example, occupation, or colour. Another name for this is **categorical** value.

It is easier to understand with an example of how this affects how visual elements (aesthetics) are mapped to values. First, we'll create an example dataset of fruit with three variables: total, weight, and delivery (day):

```
df = tibble(fruit = c("banana", "apple", "orange"),
            total = c(100, 200, 300) ,
       weight = c(50.2, 75.11, 20.547),
       delivery = c("Monday", "Wednesday", "Saturday"))

df
```

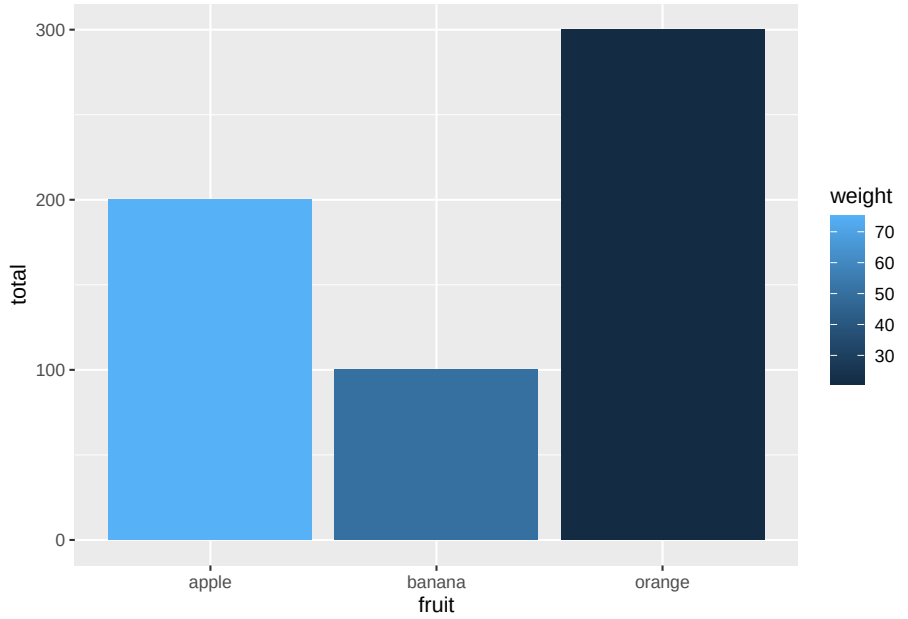
```
## # A tibble: 3 x 4
##   fruit  total weight delivery
```

```
##   <chr>  <dbl>  <dbl> <chr>
## 1 banana   100   50.2 Monday
## 2 apple    200   75.1 Wednesday
## 3 orange   300   20.5 Saturday
```

- In this data, R treats *fruit* as a **discrete** value, *total* and *weight* as **continuous** values, and *delivery* as a **discrete** value.

The type of data has an important role in how the data will be visualised and is worth understanding. To demonstrate, we'll assign the **fill** aesthetic first of all to a continuous value (weight) and then to a discrete value (delivery):

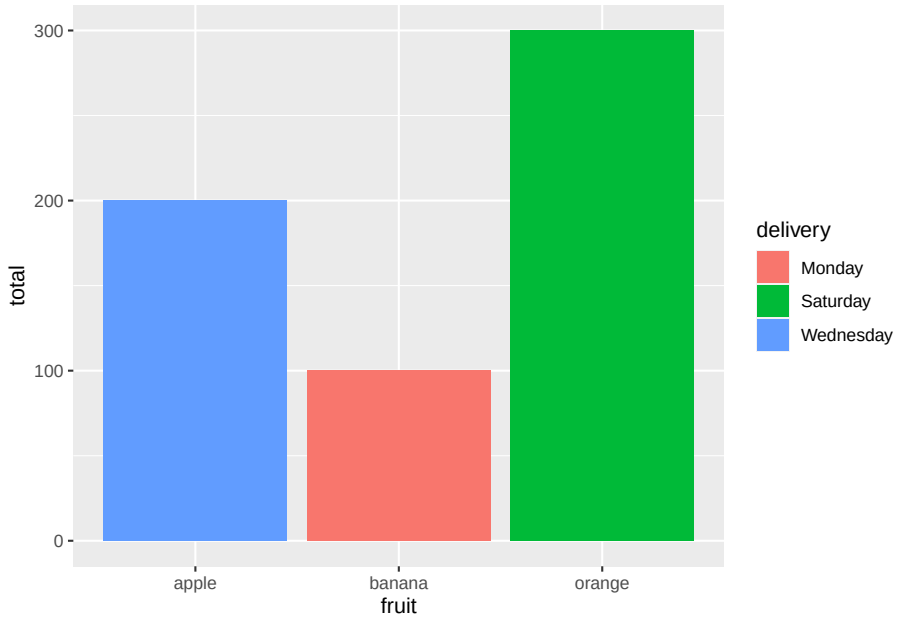
```
df %>% ggplot() + geom_col(aes(x = fruit, y = total, fill = weight))
```



Ggplot (and ggraph will work in the same way) assigns each bar a colour from a spectrum running from dark to light blue, according to its value. In this case, **the larger the weight value, the lighter the colour will be.** This works well for continuous values because it allows our brain to quickly understand which values are heavier and which are lighter - something which would be particularly useful if the data visualisation was much larger and had many more bars (in this case, we would also probably like to reverse the scale so that lighter weights were a lighter colour, but that is an easy change to make).

Now if we plot with a **discrete** value, **delivery:**

```
df %>% ggplot() + geom_col(aes(x = fruit, y = total, fill = delivery))
```



In this case, ggplot recognises that the values are discrete, and changes the colour scheme accordingly. Rather than try and map the values across a colour scale, it picks three **constrasting** colours to highlight that each value is a separate category, and they are not from across a continuous range, as with weight.

You will notice similar changes when you map additional aesthetics to your network using ggraph. Depending on whether the value you try to map is continuous or discrete, you will see different results.

Some aesthetics can only map to one of the two values. We saw this in the last class when we tried to map a continuous variable (degree) to the **shape** aesthetic. Unlike with colour, it's not possible to have a continuous spectrum of shapes - something can only be one shape or another.

An opposite example is the aesthetic **alpha,** which means **transparency.** Transparency cannot be categorical (at least in practical terms), so mapping alpha to a **discrete** variable doesn't make any sense.

11.2 Other aesthetics: colour, alpha

Using the same data as yesterday, we'll demonstrate some of the other aesthetics which can be mapped to values.

We'll also use `left_join` to add the table of node attributes used in week 4, which we can then visualise in the network. Using `left_join` means they are joined to the nodes table. Note that some of them are discrete and some are continuous variables.

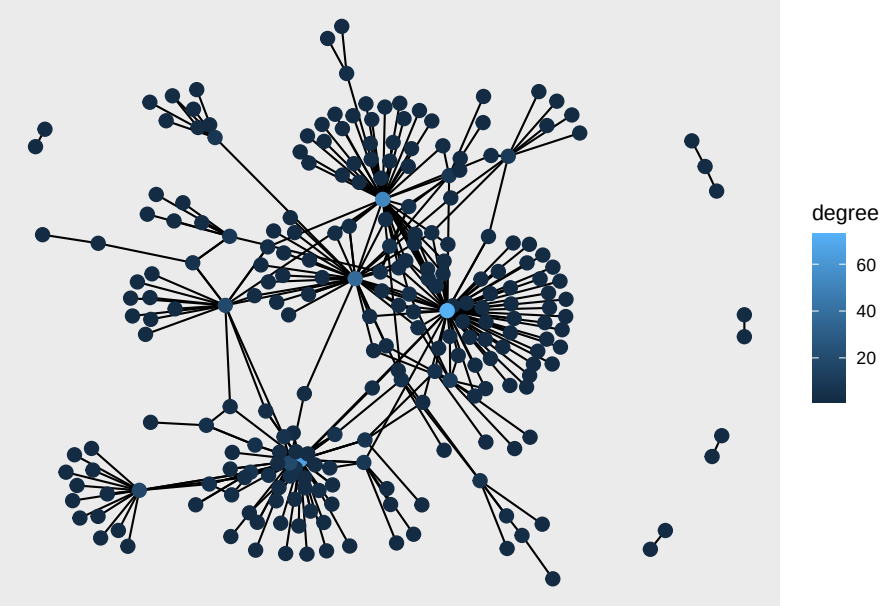
```
library(tidygraph)
library(tidyverse)
library(ggraph)
letters = read_csv("../applying-network-analysis-to-humanities/data/letter_data1670.csv

node_attributes = read_csv('../applying-network-analysis-to-humanities/data/node_attrib


edge_list = letters %>%
  group_by(from_id, to_id) %>%
  tally(name = 'weight')

sample_tbl_graph = edge_list %>%
  as_tbl_graph()

sample_tbl_graph = sample_tbl_graph %>% left_join(node_attributes, by = 'name')

sample_tbl_graph
```

```
## # A tbl_graph: 248 nodes and 364 edges
## #
## # A directed simple graph with 6 components
## #
## # Node Data: 248 x 13 (active)
##   name  main_n~ all_na~ links birth_~ death_~ gender roles_~ wikida~ occupa~
##   <chr> <chr>   <chr>   <chr>   <dbl>   <dbl> <chr>  <chr>   <chr>   <chr>
## 1 E000~ Dr. La~ Dr. La~ http~    1632    1703 male   chapla~ http:/~ chapla~
## 2 E000~ Edward~ Edward~ http~    1638    1697 male   astron~ http:/~ astron~
## 3 E000~ Dr. Ri~ Dr. Ri~ http~    1619    1681 male   priest  http:/~ priest
## 4 E000~ Solms-~ Prince~ http~    1602    1675 female art co~ http:/~ art co~
## 5 E000~ Arthur~ Arthur~ http~    1614    1686 male   politi~ http:/~ politi~
## 6 E000~ Sir Jo~ Sir Jo~ http~    1603    1671 male   politi~ http:/~ politi~
## # ... with 242 more rows, and 3 more variables: place_of_birth <chr>,
## #   place_of_death <chr>, politician <chr>
## #
## # Edge Data: 364 x 3
##    from    to weight
##   <int> <int>  <int>
## 1     1    89      1
## 2     2    22      4
```

```
## 3     2   199      1
## # ... with 361 more rows
```

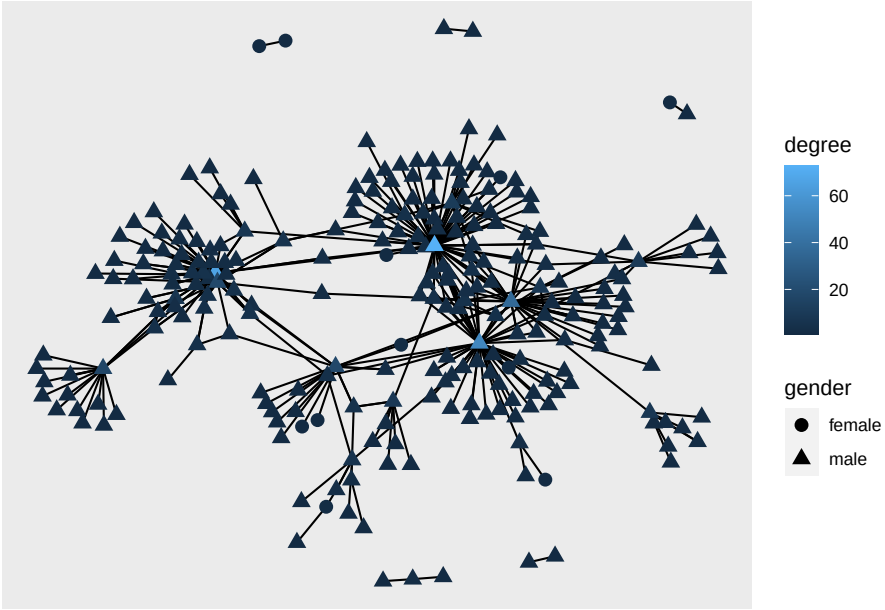
First, we'll calculate a degree score, and map this to colour, which is a continuous variable. We'll also increase the size of all nodes (notice it is outside the `aes()` function) to make it easier to see the changes.

```
sample_tbl_graph %>%
  mutate(degree = centrality_degree(mode = 'all')) %>%
  ggraph(layout = 'fr') +
  geom_edge_link() +
  geom_node_point(size = 3, aes(color = degree))
```



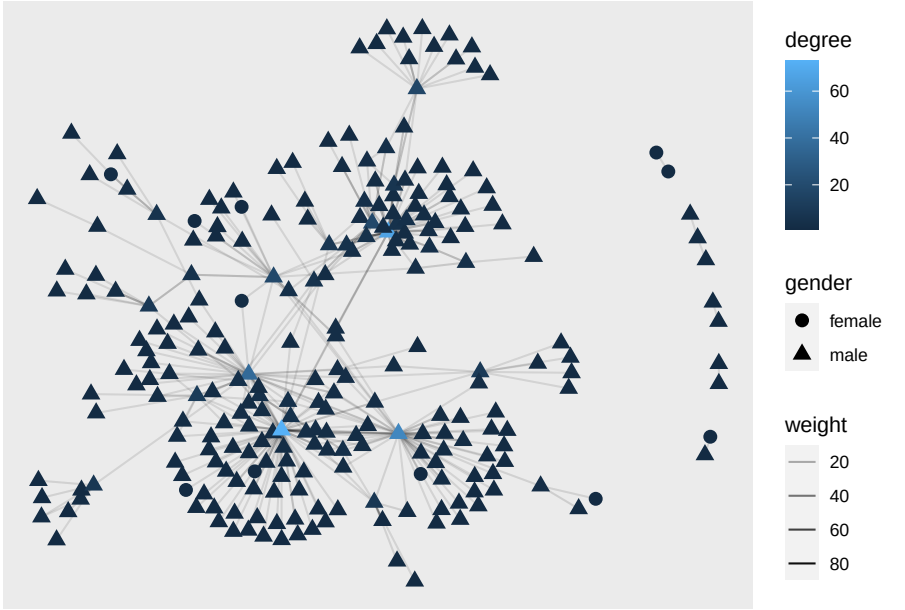
Next, we'll map gender to the shape aesthetic, which (clearly this is a traditional, binary system) is coded as a discrete value:

```
sample_tbl_graph %>%
  mutate(degree = centrality_degree(mode = 'all')) %>%
  ggraph(layout = 'fr') +
  geom_edge_link() +
  geom_node_point(size = 3,
                  aes(color = degree, shape = gender))
```



Last, we'll map the weight value in the edges table to the alpha aesthetic, which controls transparency. Note that the code for this is included within `geom_edge_link()` this time.

```
sample_tbl_graph %>%
  mutate(degree = centrality_degree(mode = 'all')) %>%
  ggraph(layout = 'fr') +
  geom_edge_link(aes(alpha = weight)) +
  geom_node_point(size = 3, aes(color = degree, shape = gender))
```



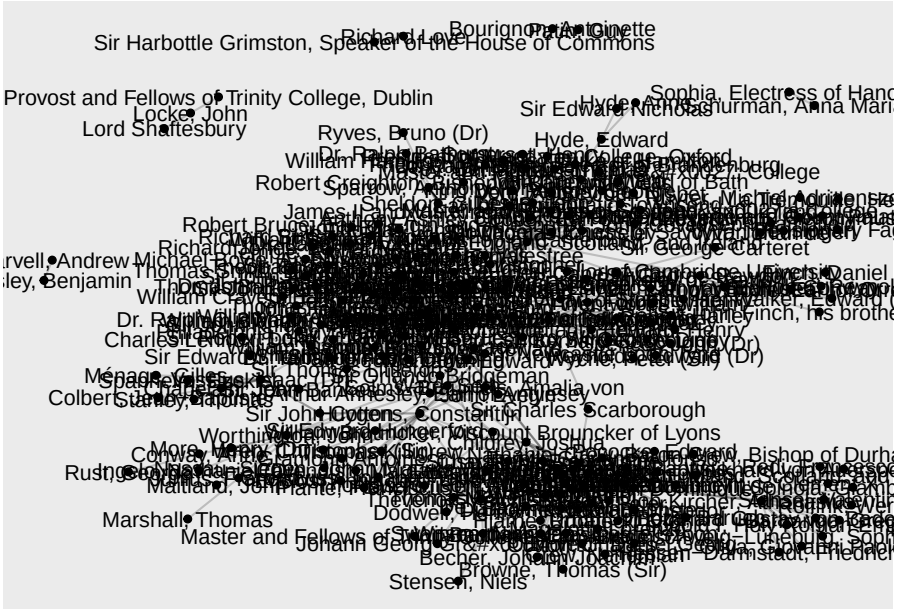
Try changing the mappings in the code above to see what works best, and to notice the difference between continuous and discreet variables: for example, map gender to colour, and degree to alpha.

11.3 Adding node labels

To add text labels to your network, you need to add a separate geom layer to your network. The correct one to use is `geom_node_text()`.

To specify which data ggraph should use to draw the labels, map the label aesthetic to the required column. In this case, we will use `main_name`.

```
sample_tbl_graph %>%
  ggraph('fr') +
  geom_node_point() +
  geom_node_text(aes(label = main_name)) +
  geom_edge_link(alpha = .2)
```



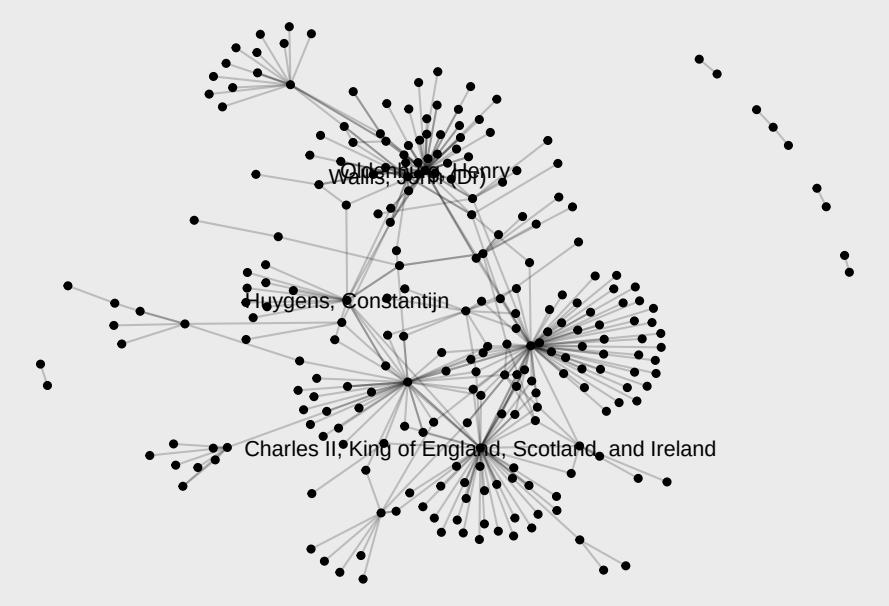
You'll notice that the text is unreadable because there are too many overlapping labels.

In a larger network, it can be helpful to only show labels belonging to the most-connected nodes.

It's a little complicated to explain the syntax, but you can use copy and paste this example to your own networ: just change the `main_name` to the correct column you wish to get the labels from in your own network.

```
sample_tbl_graph %>%
  mutate(degree = centrality_degree()) %>%
  ggraph('fr') +
  geom_node_point() +
  geom_node_text(aes(label = if_else(degree >10, main_name, NULL))) +
  geom_edge_link(alpha = .2)
```

```
## Warning: Removed 244 rows containing missing values (geom_text).
```



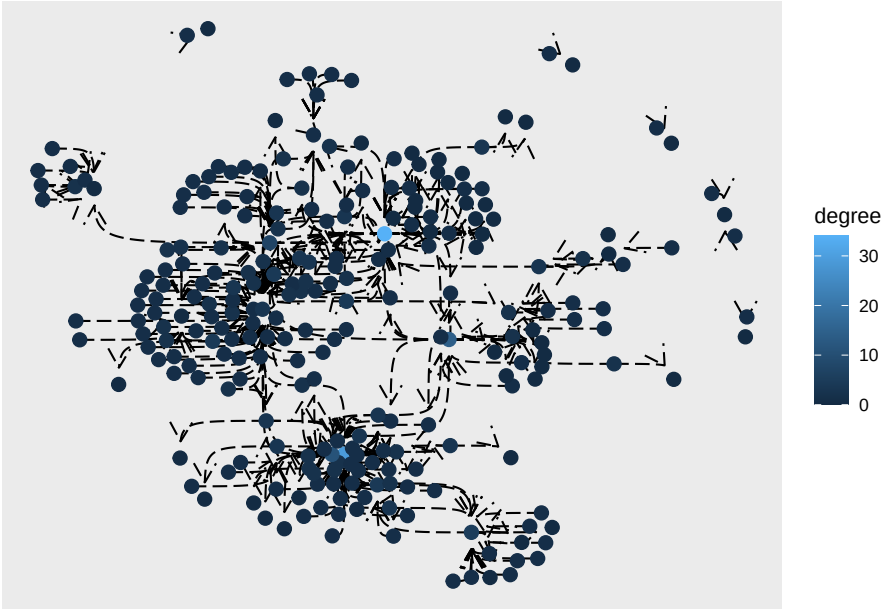
11.4 Some other things you can change

We won't go into detail with these, but it might be useful to have as a reference when you are building your own graph.

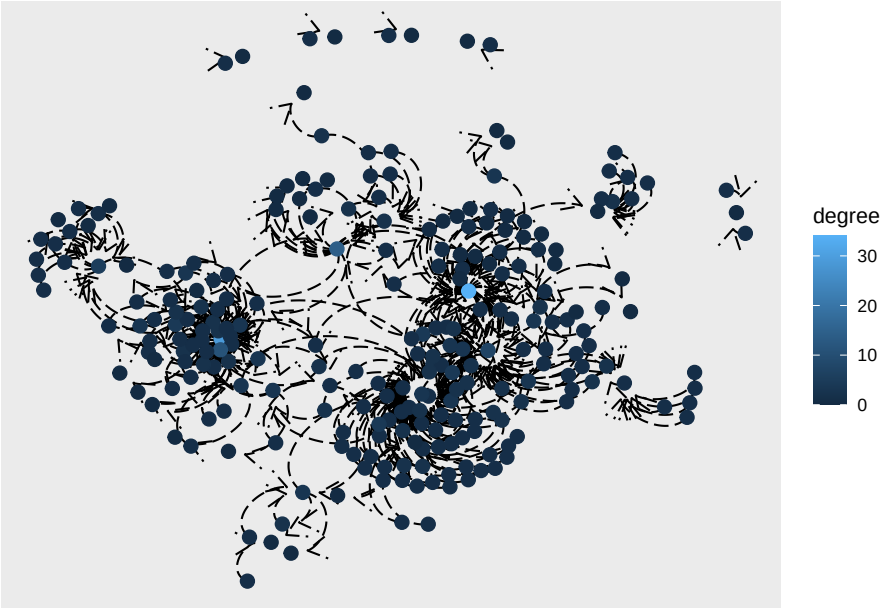
11.4.0.1 Edge types

You can make extensive changes to the type of edge. In many cases, these may make the diagram more readable. Some have particular uses, for example `geom_edge_fan()` is used for a multigraph (where you have multiple different types of edges), and `geom_edge_arc` can be useful for when you want to visualise the difference between incoming and outgoing edges:

```
sample_tbl_graph %>%
  mutate(degree = centrality_degree()) %>%
  ggraph('fr') +
  geom_edge_bend(linetype = 5,
                 arrow = arrow(length = unit(4, 'mm')),
                   end_cap = circle(3, 'mm')) +
  geom_node_point(size = 3, aes(color = degree))
```



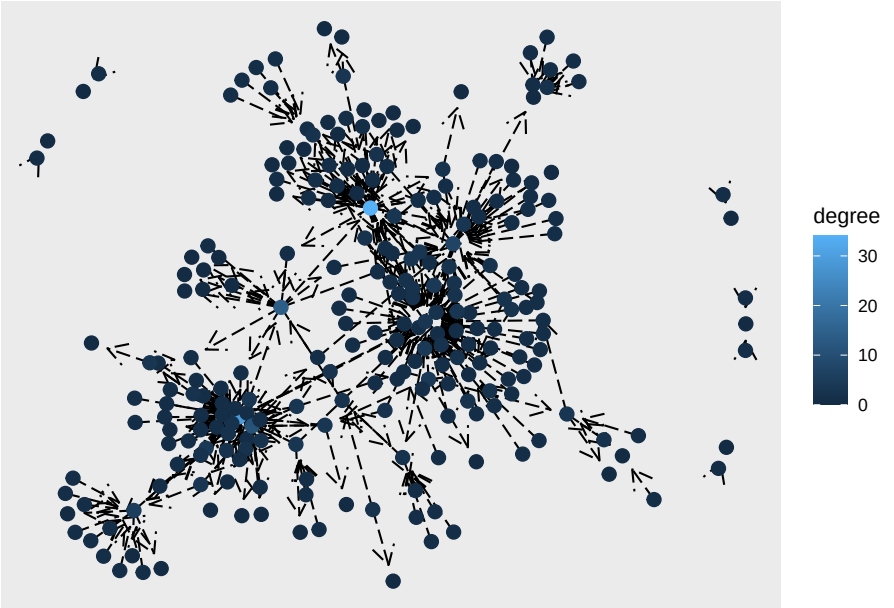

```
sample_tbl_graph %>%
  mutate(degree = centrality_degree()) %>%
  ggraph('fr') +
  geom_edge_arc(linetype = 5,
                arrow = arrow(length = unit(4, 'mm')),
                  end_cap = circle(3, 'mm')) +
  geom_node_point(size = 3, aes(color = degree))
```



11.4.0.2 Arrows

Add arrows with the following syntax. The `length` and `endcap` arguments control the appearance of the arrow, so you can experiment with changing those.

```
sample_tbl_graph %>%
  mutate(degree = centrality_degree()) %>%
  ggraph('fr') +
  geom_edge_link(linetype = 5,
                 arrow = arrow(length = unit(4, 'mm')),
                   end_cap = circle(3, 'mm')) +
  geom_node_point(size = 3, aes(color = degree))
```



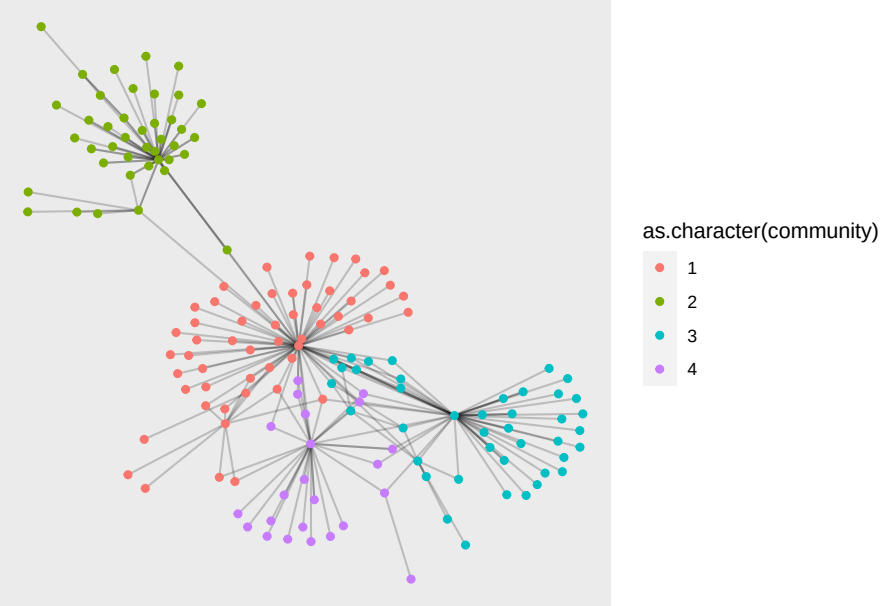
11.4.1 Calculating and coloring by community detection

One very common visualisation is colouring the various groups in the network, which might give us a clue as to its structure.

This is an interesting case of where we need to be mindful of the differences between continuous and discrete values. The output for the community detection is a number: node 1 will be placed in group 1, node 2 in group 2, node 3 in group 1, and so forth. By default, ggraph treats these numbers as continuous variables, and will map them to a colour scale as in the first example above. However, group membership should be considered a discrete value.

In order to tell ggraph to treat it as a discete value, we wrap the value in the `as.character()` function. This tells R to treat the value as a character string instead of a number, and ggraph treats character strings as discrete variables.

```
sample_tbl_graph %>%
 to_undirected() %>% # change the graph to undirected, in order to make the community
  mutate(community = group_louvain(weights =NULL)) %>% # calculate the group membershi
  filter(community %in% 1:4) %>% # filter to only include the first four groups
  ggraph('fr')  +
  geom_edge_link(alpha = .2) +  # reduce the transparency of the edges to make it easi
  geom_node_point(aes(color = as.character(community)))  # map the colour aesthetic to
```



11.5 What makes a good network visualisation?

You will rarely be able to make a good network visualisation simply by pushing a button and using the default settings. You should think about what your network is trying to communicate and to what audience, and spend some time

- Treat the visualisation as carefully as you would an essay or other piece of text. Is it easy to understand?
- Make sure you really need a network graph, and that another method of communicating the information wouldn't be better. For example, if your diagram is simply showing the most influential nodes, would a bar chart (or even a table) with their degree work just as well?
- Use colour/shape/size if you need to, but make sure you explain what each element is doing.
- Spend some time showing your readers how to 'read' the visualisation. What is interesting about the left/right/top/bottom of the diagram? What is the significance of nodes grouped together, and what about nodes in the centre versus nodes towards the edge? Are there obvious clusters and gaps, and do these show up in the visualisation?

11.6 Case study: Scientists and Politicians

In this final section, I'll demonstrate how these methods might be used to carry out exploratory data analysis on the sample letter dataset.

The node attributes can used either to filter the network, or added as extra visual elements. Doing so helps us to understand a bit more about why the network might look the way it does.

First, load the same node attributes table as before:

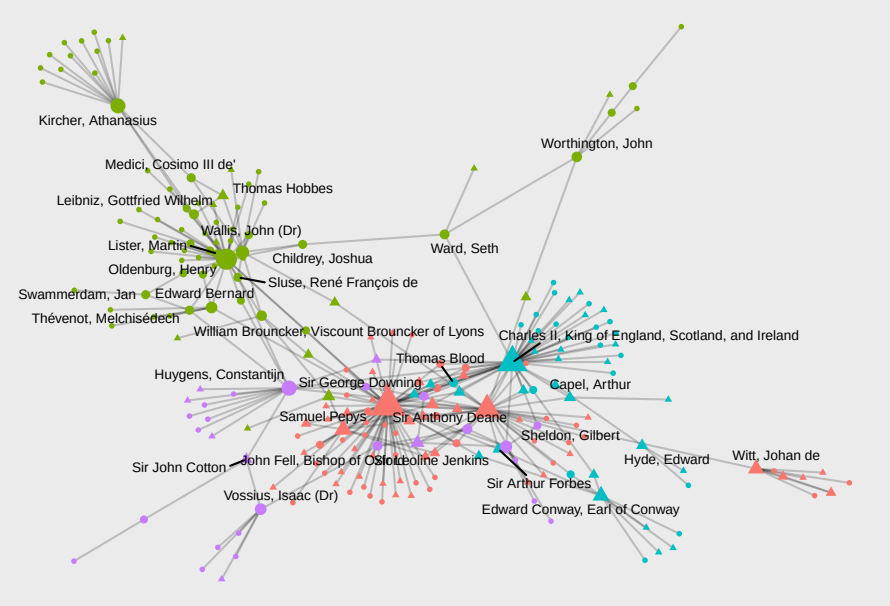
```
node_attributes = read_csv('node_attributes.csv')
```

```
## Rows: 248 Columns: 13
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (11): name, main_name, all_names, links, gender, roles_titles, wikidata_...
## dbl  (2): birth_year, death_year
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

This external information might help us to make sense of the clusters found by the community detection algorithm used above. One of the fields in the data is whether that person is listed as a politician on Wikidata: the field is a simple flag of either `yes` or `no`.

To check whether this might be a clue towards the structure of the network, we will set the *color* to the community detection results, and the *shape* to the politician flag, with the following code:

```
sample_tbl_graph %>%
  igraph::as.undirected() %>%
  as_tbl_graph()%>%
  activate(nodes) %>%
  mutate(degree = centrality_degree(mode = 'total')) %>%
  left_join(node_attributes) %>%
  mutate(community = group_edge_betweenness(weights =NULL)) %>%
  filter(community %in% 1:4) %>%
  ggraph('fr') +
  geom_edge_link(alpha = .2)+
  geom_node_point(aes(size = degree, color = as.character(community), shape = politici
  geom_node_text(aes(label = ifelse(degree >2, main_name, NA)), size = 2.5, repel = T)
  theme(legend.position = 'none')
```



It does look like one of the four clusters (cluster 2) has far fewer politicians. This cluster seems to be a group of what might be described as natural philosophers.

Other things to look out for in this visualisation:

- What is the importance of the nodes sitting in between the scientists and the politicians? What metrics might they score highly on, and what role might they occupy in this system? What kinds of information may they be able to pass on?
- What is the position of Athanasius Kircher? What does it mean to be on the periphery of this network?
- What effect might additional data have on the metrics in this network? Should this make us cautious about any inference from this network?

11.7 Conclusions

Hopefully, from reading this chapter, you'll be convinced that network visualisations are useful, but that they should also be approached with caution. The bottom line is, visualisations of themselves are not a result: at the very least, they need extensive commentary in order to explain them, and in some cases, they may simply be useful ways of describing the network.

At the same time, in combination with the additional attributes from our data model, thoughtful visualisations can be incredibly useful for exploring a network

dataset, revealing patterns that are otherwise hidden. I encourage you to use visualisations in your final project if they make sense, are not just 'hairballs', and if they can be justified with relevant commentary.

Chapter 12

Week 6, Class 1: Asking and Answering Questions with Networks

12.1 Introduction

In this chapter, we'll explore how you might go about using your new-found network analysis skills to actually tackle a humanities question. Producing a network diagram is not enough: you need to use these techniques as a tool. To come up with an interesting question, think about the reading we've been doing over the past weeks, and think about the kinds of questions networks are particularly adept at answering: questions to do with local vs. global, flows, structures, and groups.

The purpose of this chapter is to walk through two sample analyses, one with each of the course recommended datasets you worked with in weeks 2 and three. The aim is to demonstrate an example workflow which takes you from raw data to a particular analysis. Take note of the methods and the coding steps, which may be helpful when you come to do you final project. The first example creates a **bipartite network** from the ESTC data, and the second uses **twitter data** to look at the networks surrounding the campaign to repeal the ban on abortion in Ireland.

12.1.1 How to go about tackling humanities questions of networks?

Of course, how you do this depends very much on your question and dataset. However, the set of steps below are a good starting-point for developing a work-

flow which will allow you to understand and analyse humanities networks.

- A good place to start is with an **exploratory analysis**. Calculate **global network metrics**, get the **size**, **density**, **clustering coefficients**. Try to interpret these, with your own data in mind. What does it mean for this particular network to be dense or sparse? What does the clustering mean?
- With this information, decide whether its worth making an exploratory **network visualisation**. Spend some time picking out the most **informative visualisation algorithm**, and **filter** the data if needed. **Interpret your visualisation**, with a critical eye. As the reading in week 5, think about what the orientation, the clustering, and the centre and peripheries says about this network.
- Armed with the information from your exploratory analysis, **formulate a question or hypothesis** for your network. Use **existing work** for inspiration if needed!
- Run community detection algorithms. If you have additional node attributes from your data model, use these to summarise each group. Look for patterns amongst the nodes in each group.
- 'Zoom in', and look closely at the groups, nodes, and their data from a humanities perspective. Are there any unusual or stand-out nodes?
- Use your additional **node attributes** and **network measurements** to **test your hypothesis**. Use **statistics** if you're comfortable with it, but don't be afraid to use more humanities-type methods such as **close reading**.
- **Write up your findings**. Justify the choices you've made with your data and methods. Think about how you can communicate your relevant network metrics and a network diagram, if needed. Work on making your plots and diagrams **readable**.

12.1.2 Practical Example: ESTC

In this example, we'll produce a large network from the English Short Title Catalogue and use it to tackle questions about eighteenth century publishing. The questions we are interested in:

- What is the structure of the network of publishers in 18th century Britain?
- What nodes form clusters, and what are these clusters made of?
- Who are the most important 'bridge' publishers connected different groups?

For many of these questions, we would ideally know detailed information about the nodes, for example their places of birth, the types of books they published, and so forth. Because we don't have much information on most of these publishers, we will infer it, for example by counting the books attached to their records.

12.1.2.1 Making the dataset

First, download and make a network of the ESTC. We have restricted it to 1700:1750, and have already made a dataset containing the relevant books.

The first step is to turn this into a bipartite, projected network, following the steps in the previous lesson. This will result in a network of publishers connected by shared co-occurrences on book title pages.

Make the publisher->book network, using the techniques from previous chapters.

```
library(tidyverse)
library(tidygraph)
library(igraph)
library(ggraph)
estc_actor_links = read_csv('actor_links.csv')
```

```
## Rows: 147140 Columns: 2
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): estc_id, actor_id
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
book_info = read_csv('book_info.csv')
```

```
## Rows: 84808 Columns: 6
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (5): estc_id, short_title, publication_place, author_id, author_name_pri...
## dbl (1): publication_year
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
publisher_graph = estc_actor_links %>% distinct(estc_id, actor_id) %>%
  graph_from_data_frame(directed = FALSE) %>% as_tbl_graph()
```

Add a type attribute to the nodes, based on whether they are in the first or second column, using `bipartite mapping`. *Project* it to publisher->publisher with `bipartite.projection`.

```
V(publisher_graph)$type <- bipartite_mapping(publisher_graph)$type
```

```
g = bipartite.projection(publisher_graph, which = TRUE)
```

12.1.2.2 Get global network metrics

In any analysis, a useful first step is to calculate some basic network metrics, so we know how dense it is, to what extent it is clustered, and so forth. Use the commands from the previous lesson to calculate network size (in nodes and edges), density, transitivity (clustering), and average path length:

```
edge_size = gsize(g)
node_size = gorder(g)
density = igraph::graph.density(g)
global_clustering = igraph::transitivity(g, type = 'global')
average_local_cluster = igraph::transitivity(g, type = 'localaverage')
average_path = igraph::average.path.length(g)
```

To view this, we can turn these numbers into a single dataframe:

```
tibble(edge_size, node_size, density, global_clustering, average_local_cluster, averag
```

```
## # A tibble: 1 x 6
##   edge_size node_size  density global_clustering average_local_cluster average~1
##       <dbl>     <int>    <dbl>             <dbl>                 <dbl>     <dbl>
## 1     50383     10621 0.000893             0.225                 0.731      4.80
## # ... with abbreviated variable name 1: average_path
```

Of course, interpreting the numbers here is not straightforward, because it depends on the particular network and its size. However, we could say that this is actually quite a dense network - there are five times as many edges as there are nodes, and it has a high amount of clustering. So, we might surmise that this network is formed of tightly-connected groups.

12.1.2.3 Calculate node-level statistics

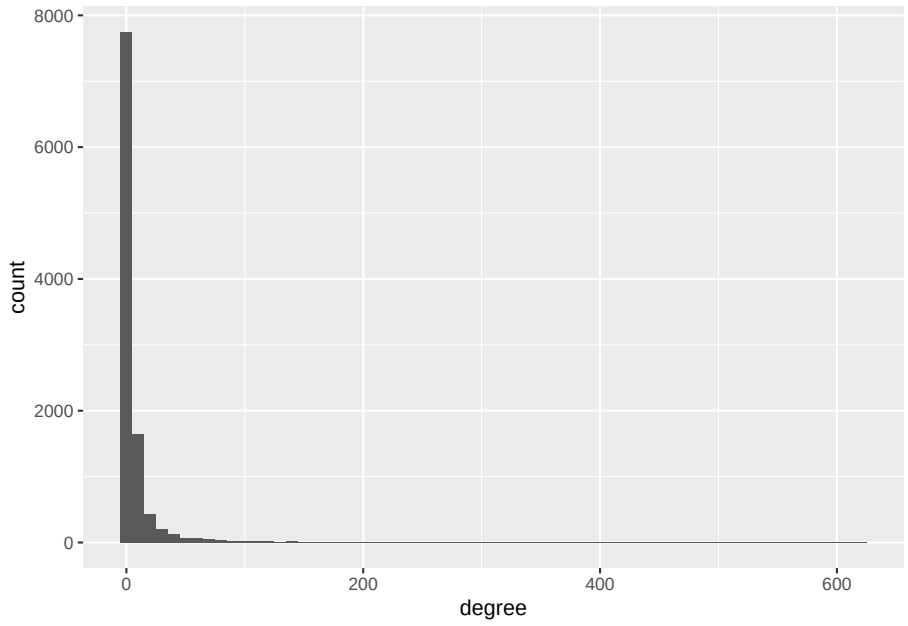
Next, we'll calculate some node-level statistics, including centrality measurements and community membership. Turn this projection into a `tbl_graph`, and then use `mutate()` plus the commands to calculate the various information on the nodes. We also calculate community membership using a community detection algorithm, with `group_louvain()`.

```
tbl_g = g %>% as_tbl_graph() %>%
  mutate(degree = centrality_degree(weights = NULL, mode = 'all'))%>%
  mutate(between = centrality_betweenness(weights = NULL)) %>%
  mutate(group = group_louvain(weights = NULL))
```

```
## Warning in betweenness(graph = graph, v = V(graph), directed = directed, :
## 'nobigint' is deprecated since igraph 1.3 and will be removed in igraph 1.4
```

Again, we can use this information to get some aggregated statistics. One interesting one to look at is degree distribution. We can make a histogram of the degree scores for each node:

```
tbl_g %>% as_tibble() %>%
  ggplot() +
  geom_histogram(aes(degree), binwidth = 10)
```



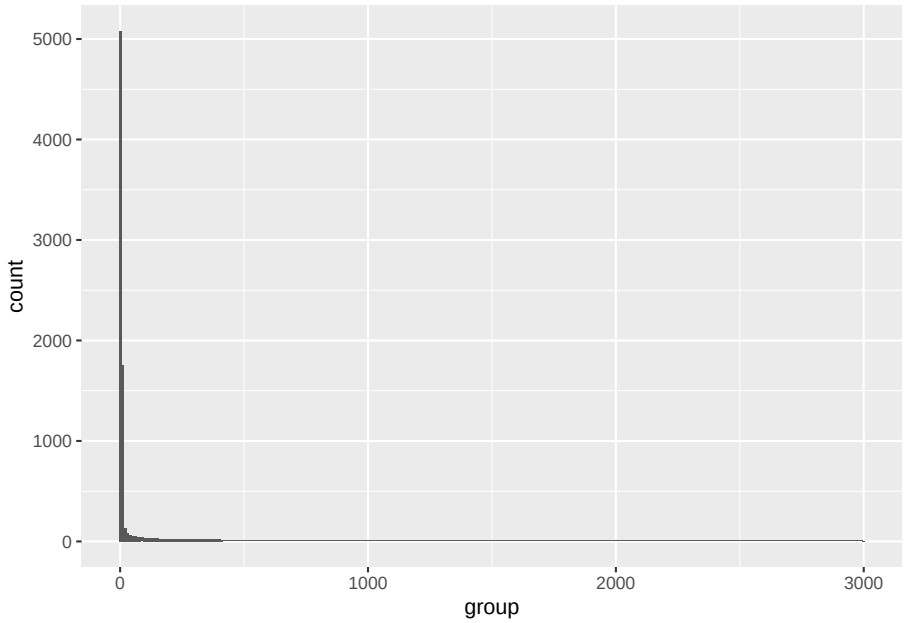
This shows that the vast majority of nodes have a degree between 0 and 10, then far fewer between 10 and 20, and so forth. A very small number of nodes (probably 1) has a degree score of around 600.

12.1.2.4 Look at individual groups

Next we want to understand something about the groups which make up this network. We know they cluster together, but do they cluster together based on location? Or time period? Or something else?

Again, we can look at how many nodes there are in total in the groups:

```
tbl_g %>% as_tibble() %>% ggplot() + geom_histogram(aes(group), binwidth = 10)
```



As with the degree distribution, a very small number of the groups contain nearly all the nodes. The first 10 or so groups are large, then there are a large number of very small groups. Many of these are probably disconnected components, cut off from the wider data. We can concentrate our analysis on the first 10 groups using `filter()`.

We can use the tidyverse to create a summary dataset of the groups, attached to some additional information. First, we need to get all the publication places and years for all actors in the dataset. Then, we'll join this information to the groups we have, and count how many of each occurrences we have.

```
publication_places = estc_actor_links %>%
  select(actor_id, estc_id) %>%
  left_join(book_info %>%
              select(estc_id, publication_place)) %>%
  count(actor_id, publication_place)
```

```
## Joining, by = "estc_id"
```

```
publication_years = estc_actor_links %>%
  select(actor_id, estc_id) %>%
  left_join(book_info %>%
              select(estc_id, publication_year)) %>%
  count(actor_id, estc_id, publication_year)
```

```
## Joining, by = "estc_id"
```

Next, use `group_by()` and `summarise()` to make a dataset counting the places of publication for each group. As we can see, there are two large London groups, a Scottish group, an Irish group, an American group, and a number of smaller mixed ones. So we can say quite comprehensively that the network clusters together based on geographic proximity.

```
tbl_g %>% filter(group %in% 1:10) %>%
  as_tibble() %>%
  left_join(publication_places, by = c('name' = 'actor_id')) %>%
  count(group, publication_place, wt = n) %>%
  arrange(desc(n)) %>%
  mutate(tot = paste0(publication_place, " (", n, ")")) %>%
  group_by(group) %>% summarise(tots = paste0(tot, collapse = '; ')) %>%
  DT::datatable()%>%
  DT::formatStyle(columns = 1:3, fontSize = '70%')
```

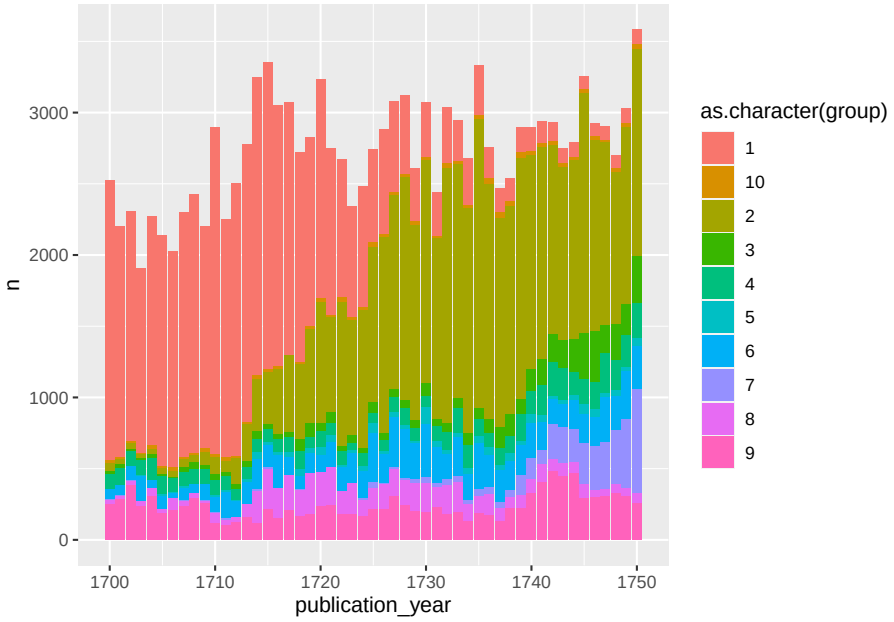
Show 10 entries Search:

	group	tots
1	1	London (46578); Oxford (262); Edinburgh (96); Cambridge (52); NA (38); York (27); Nottingham (16); Exeter (14); Norwich (12); Dublin (11); Durham (6); Liverpool (6); Bristol (5); Amsterdam (4); Ipswich (2); Salisbury (2); Aldershot (1); Boston Ma (1); Colchester (1); Derby (1); Manchester (1); Portsmouth (1); Reading (1); Shrewsbury (1); Stamford (1)
2	2	London (45638); NA (128); Oxford (124); York (62); Cambridge (57); Dublin (28); Edinburgh (23); Bristol (18); Birmingham (16); Bath (10); Eton (10); Coventry (9); Reading (9); Salisbury (9); Sheffield (9); Exeter (7); Ipswich (7); Charleston SC (6); Chester (6); Leeds (6); Newcastle (6); Nottingham (6); Stamford (5); Boston Ma (4); Glasgow (4); Lincoln (4); Manchester (4); Northampton (4); Paris (4); Gloucester (3); Trevoux (3); Worcester (3); Aberdeen (2); Carlisle (2); Hannover (2); Hull (2); Maidstone (2); Amsterdam (1); Canterbury (1); Derby (1); Sherborne (1); Shrewsbury (1)
3	3	London (1885); Cambridge (902); Bristol (566); Exeter (395); Oxford (179); York (167); Norwich (116); Reading (101); Salisbury (84); Ipswich (75); Manchester (62); Bath (61); Gloucester (48); NA (38); Lincoln (37); Colchester (36); Stamford (27); Worcester (26); Newcastle (21); Taunton (15); Liverpool (9); Malton (9); Sherborne (9); Shrewsbury (8); Bury St Edmunds (7); Chichester (4); Canterbury (1); Dublin (1); Edinburgh (1)
4	4	Edinburgh (5159); Glasgow (853); London (244); Newcastle (144); NA (83); Aberdeen (80); York (4); Bristol (3); Amsterdam (2); Dublin (2); Dumfries (2); Leiden (2); Northampton (2); Lincoln (1); Manchester (1)
5	5	London (447); Northampton (302); York (297); Nottingham (262); Leeds (80); Birmingham (62); Derby (59); Worcester (43); NA (31); Cambridge (12); Oxford (11); Hull (8); Coventry (4); Bristol (3); Colchester (2); Daventry (1); Gloucester (1); Ipswich (1); Leicester (1)
6	6	Dublin (9538); London (97); Cork (42); NA (37); Edinburgh (5); Limerick (5); New York NY (2)
7	7	London (4168); NA (10); Newcastle (9); Preston (9); Edinburgh (8); Bristol (5); Exeter (4); Cirencester (3); Oxford (3); Reading (3); Eton (2); Sherborne (2); Birmingham (1); Dublin (1); Gloucester (1); Ipswich (1); Paris (1); Yeovil (1)
8	8	London (4404); Oxford (1697); Canterbury (59); Cambridge (30); Reading (11); NA (10); Exeter (3); Dublin (2); Gloucester (2); Salisbury (2); Birmingham (1); Bristol (1); Nottingham (1); Shrewsbury (1)
9	9	Boston Ma (7477); London (1566); Philadelphia Pa (1069); New York Ny (818); New London Ct (598); NA (364); Annapolis Md (62); Germantown Pa (57); Newport Ri (44); Cork (34); New York NY (6); Burlington Nj (2); Charleston Sc (2); Edinburgh (1); Perth Nj (1)
10	10	London (894); Norwich (130); Canterbury (12); Wolverhampton (5); NA (4); Stamford (3); Chichester (2); Fersfield (2); Rotterdam (2)

Showing 1 to 10 of 10 entries Previous 1 Next

In a further step, we can add the year of publication for all works by all nodes, to see if there are patterns there too:

```
tbl_g %>%
 as_tibble() %>%
 left_join(publication_years, by = c('name' = 'actor_id')) %>%
 count(group, publication_year, wt = n) %>%
  filter(group %in% 1:10) %>%
 ggplot() + geom_col(aes(x = publication_year, y = n, fill = as.character(group)))
```



We see that the separate London groups do seem to be clustered along temporal lines.

To visualise this and quickly communicate it, we could add an additional table of data to the nodes, indicating the percentage of their output which was published in Edinburgh. If we then draw a network diagram, we can colour the nodes based on this metric. If high-percentage Edinburgh nodes are clustered together, this would be another indication of the degree to which this group of publishers was separate.

First, make this new dataset:

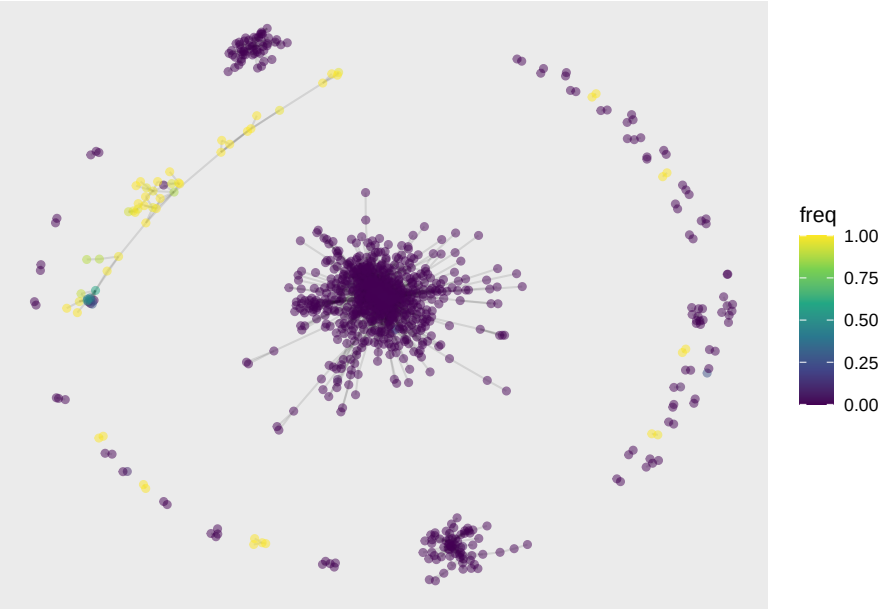
```
percent_edinburgh = book_info %>% left_join(estc_actor_links, by = 'estc_id')%>%
  group_by(actor_id, publication_place) %>%
  summarise(n = n()) %>%
  mutate(freq = n / sum(n)) %>%
  filter(publication_place %in%  'Edinburgh')
```

```
## `summarise()` has grouped output by 'actor_id'. You can override using the
## `.groups` argument.
```

Next, draw the network,

```
tbl_g %>%
  left_join(percent_edinburgh, by = c('name' = 'actor_id')) %>%
```

```
  mutate(comp  = group_components()) %>%
  mutate(freq = replace_na(freq, 0)) %>%
  filter(comp %in% 1:5) %>%
  filter(group %in% 1:10) %>% activate(edges) %>%
  filter(weight>5) %>%
  activate(nodes) %>%
  mutate(degree = centrality_degree(weights = weight)) %>%
  filter(degree>0) %>%
  ggraph('fr') +
  geom_edge_link(alpha = .1) +
  geom_node_point(aes(color = freq), alpha = .5) +
  scale_color_viridis_c()
```



12.1.3 Conclusions

- The network is highly clustered, along geographic and temporal lines.

12.2 Practical example: Twitter (and dealing with big data)

The next section is an example of the kinds of questions we might ask using the Twitter data. We'll take a look at the networks of tweets

and retweets surrounding the conversation about the [campaign to remove Ireland's ban on abortion(https://en.wikipedia.org/wiki/Thirty-sixth_Amendment_of_the_Constitution_of_Ireland). The campaign consisted of the 'yes' side, who campaigned for people to vote 'yes' to repealing the existing article of the consitution, which essentially made abortion illegal, and the 'no' side, who wished to maintain the status quo. The same steps could be followed with any subset of data downloaded and hydrated using the Tweetsets, as we explored in weeks 2 and 3.

12.2.1 Download data

The first thing I did was to download the relevant data, following the steps in our earlier lesson. I downloaded the full dataset of 2,276,808 tweets, then used the tool to 'hydrate' them, which took approximately a day (worth bearing in mind if you're using Twitter data for your final project). These hydrated tweets were saved in a .csv file called `repeal_tweets.csv`.

```
repeal_tweets = data.table::fread('../../Downloads/repeal_tweets.csv')

str(repeal_tweets)
```

```
## Classes 'data.table' and 'data.frame':   1367666 obs. of  35 variables:
##  $ coordinates               : chr  "" "" "" "" ...
##  $ created_at                : chr  "Sat May 19 22:32:42 +0000 2018" "Sat May 19 22:33:28 +000
##  $ hashtags                  : chr  "LoveBothVoteNo 8thref" "" "repealthe8th" "together4Yes Re
##  $ media                     : chr  "https://twitter.com/loveboth8/status/997876898578550790/p
##  $ urls                      : chr  "" "" "https://twitter.com/oliviao_neill/status/9994160278
##  $ favorite_count            : int  0 0 34 0 0 0 0 0 0 0 ...
##  $ id                        :integer64 997968315049005056 997968507257151489 999946594870005
##  $ in_reply_to_screen_name   : chr  "" "" "" "" ...
##  $ in_reply_to_status_id     :integer64 NA NA NA NA NA NA NA NA ...
##  $ in_reply_to_user_id       :integer64 NA NA NA NA NA NA NA NA ...
##  $ lang                      : chr  "qme" "en" "en" "en" ...
##  $ place                     : chr  "" "" "" "" ...
##  $ possibly_sensitive        : logi  FALSE NA FALSE FALSE FALSE NA ...
##  $ quote_id                  :integer64 NA NA NA NA NA NA NA NA ...
##  $ retweet_count             : int  4 6 9 681 5 102 271 63 0 5 ...
##  $ retweet_id                :integer64 997876898578550790 997865462531698688 NA 997917600343
##  $ retweet_screen_name       : chr  "lovebothireland" "lovebothireland" "" "phlaimeaux" ...
##  $ source                    : chr  "<a href=\"\"http://twitter.com\"\" rel=\"\"nofollow\"\">T
##  $ text                      : chr  "#LoveBothVoteNo #8thref https://t.co/xc4jslD0nm" "Have yo
##  $ tweet_url                 : chr  "https://twitter.com/corkmankeane/status/9979683150490050
##  $ user_created_at           : chr  "Mon Apr 30 07:24:29 +0000 2012" "Mon Apr 30 07:24:29 +000
##  $ user_id                   :integer64 567006745 567006745 235172191 4149894277 567006745 451
```

```
##  $ user_default_profile_image: logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ user_description          : chr  "Prolife\nHusband and Dad" "Prolife\nHusband and
##  $ user_favourites_count     : int  44701 44701 177490 6814 44701 5518 6605 49400 3
##  $ user_followers_count      : int  680 680 27325 292 680 329 184 1546 528 832 ...
##  $ user_friends_count        : int  1287 1287 14899 1222 1287 1393 558 1741 983 969
##  $ user_listed_count         : int  20 20 143 3 20 2 1 28 16 42 ...
##  $ user_location             : chr  "Ireland" "Ireland" "" "Edinburgh, Scotland" ..
##  $ user_name                 : chr  "Joseph Keane" "Joseph Keane" "barney farmer" "
##  $ user_screen_name          : chr  "corkmankeane" "corkmankeane" "barneyfarmer" "ca
##  $ user_statuses_count       : int  48207 48207 133078 1089 48207 5544 3298 9249 173
##  $ user_time_zone            : logi  NA NA NA NA NA NA ...
##  $ user_urls                 : chr  "" "" "http://barneyfarmer.com" "" ...
##  $ user_verified             : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

This data doesn't include detailed follower information, but it does include replies and retweets. Either of these could be represented as a network. I'm going to use the retweets.

The aim is to get an *edge list* containing the user ID of the retweeter, and the user ID of the author of the original tweet. This can be used to create a directed network. The relevant columns are `user_screen_name` and `retweet_screen_name`. Use dplyr functions to count these, then turn it into a network object with `as_tbl_graph()`:

```
retweet_graph = repeal_tweets %>%
  filter(! retweet_screen_name=='') %>%
  count(user_screen_name, retweet_screen_name, name = 'weight') %>%
  as_tbl_graph()

retweet_graph
```

```
## # A tbl_graph: 252296 nodes and 743767 edges
## #
## # A directed multigraph with 3055 components
## #
## # Node Data: 252,296 x 1 (active)
##   name
##   <chr>
## 1 _________rb_
## 2 ________simon
## 3 _______2_s__
## 4 _____botong
## 5 _____Brad_____
## 6 _____Gem
```

```
## # ... with 252,290 more rows
## #
## # Edge Data: 743,767 x 3
##    from     to weight
##   <int>  <int>  <int>
## 1     1 109062      1
## 2     2  82664      1
## 3     2  82669      1
## # ... with 743,764 more rows
```

12.2.2 Extend the data model by creating a

We will also make a basic table of node information, containing the user friends and follower numbers for each account. This plus the network will consist of our data model.

For each tweet in the dataset, there is a count of that person's friends (those they are following) and followers. We will take the maximum of these two values, plus the date the user was created, and make it into a new dataset.

```
user_info = repeal_tweets %>%
  group_by(user_screen_name) %>%
  summarise(max_friends = max(user_friends_count),
            max_followers = max(user_followers_count),
            date_created = min(user_created_at))
```

At a later stage, this can be joined to the network using the `user_screen_name` key.

12.3 Basic network statistics

The result is a very large graph (250,000 nodes, 750,000 edges), which means it's not very feasible to visualise it, for example. To get an idea of the 'shape' of this network, it's better to use basic global network metrics.

To do this, we use the functions described in chapter 4, class 2:

12.3.0.1 Network size

```
retweet_graph %>% gsize()
```

```
## [1] 743767
```

```
retweet_graph %>% gorder()
```

```
## [1] 252296
```

12.3.0.2 Density

(the number of links present out of all possible links):

```
retweet_graph %>% igraph::graph.density()
```

```
## [1] 1.168471e-05
```

12.3.0.3 Average path length

(the average number of hops between every pair of nodes in the network). This can take a long time with a large network, because it has to calculate every possible path between every pair of nodes.

```
#retweet_graph %>% igraph::average.path.length()
```

12.3.0.4 Clustering coefficient:

Because there are a number of ways to calculate clustering in a network, a method needs to be specified. The clustering coefficient is also known as *transitivity*, and it is defined as the ratio of completed triangles and connected triples in the graph. This measurement can be *global* (which counts the overall ratio) or *local* (which counts the individual ratio for each node). Because we want the global measurement, specific this with the `type =` argument.

```
retweet_graph %>% igraph::transitivity(type = 'global')
```

```
## [1] 0.01054959
```

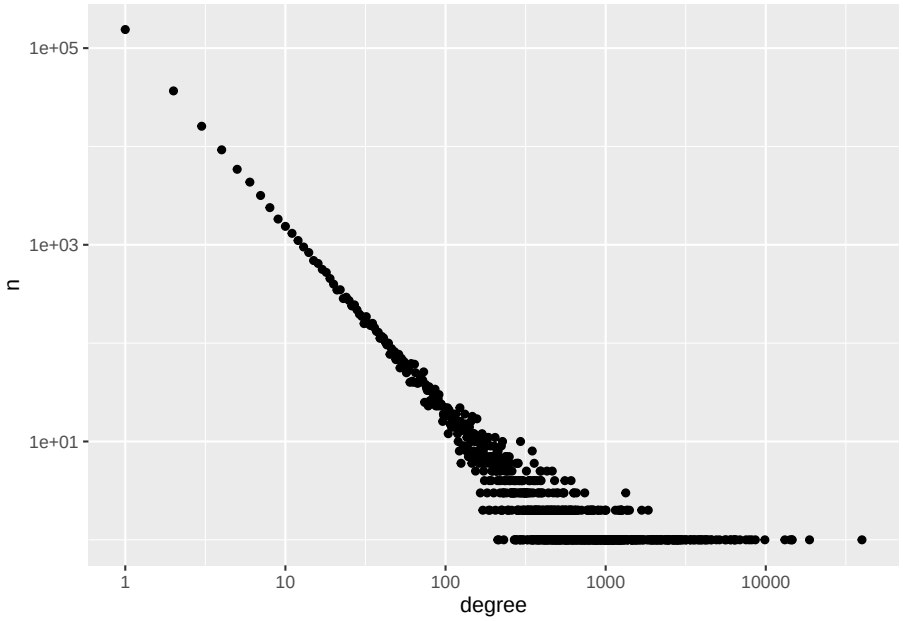
How many of the edges are reciprocated.

```
retweet_graph %>% igraph::reciprocity()
```

```
## [1] 0.02561819
```

Another measurement we can calculate is the degree distribution. A diagonal descending line means that the degree distribution follows a power law, as described by Barabasi:

```
retweet_graph %>% mutate(degree = centrality_degree(weights  = weight, mode = 'all')) %>%
  as_tibble() %>% count(degree) %>% ggplot() + geom_point(aes(degree, n)) + scale_x_log10() + sca
```



From these basic measurements we can say that the graph is large (3/4 of a million edges), it is not dense, and only a small proportion of its links are reciprocated. We also know that a very small number of users have the vast majority of links. This is a clue that the influence in this network is, like many networks, concentrated in a small number of individual twitter users.

12.4 Node-level metrics

The next step is to calculate some 'node-level' metrics, again, using commands we learned in week 4. These node-level metrics will allow us to look at the individual twitter users in this network and understand their importance or 'centrality'.

First, we calculate in-degree and out-degree separately. In this particular network, a high in-degree means that a person was retweeted a lot, and out-degree means that a person was a prolific retweeter. We'll create a new version of the

graph, called `df`, which contains these scores. We'll add the information on friends and followers too.

```
df = retweet_graph%>%
  mutate(indegree = centrality_degree(mode = 'in', weights = weight)) %>%
  mutate(outdegree = centrality_degree(mode = 'out', weights = weight)) %>%
  left_join(user_info, by = c('name' = 'user_screen_name'))
```

First, we take a look at the highest in-degree users (those with the highest number of retweets).

These are a mix of official campaign accounts, for both yes and no sides (Together4yes, SaveLivesAlways, lovebothireland), public figures (the then Taoiseach, or Prime Minister of Ireland, Leo Varadkar, the director of Amnesty International and human rights campaigner Colm O Gorman), a number of authors. There are also a few accounts with low numbers of followers. We can check to see if they went 'viral' with a single tweet, or whether they had consistent retweeting.

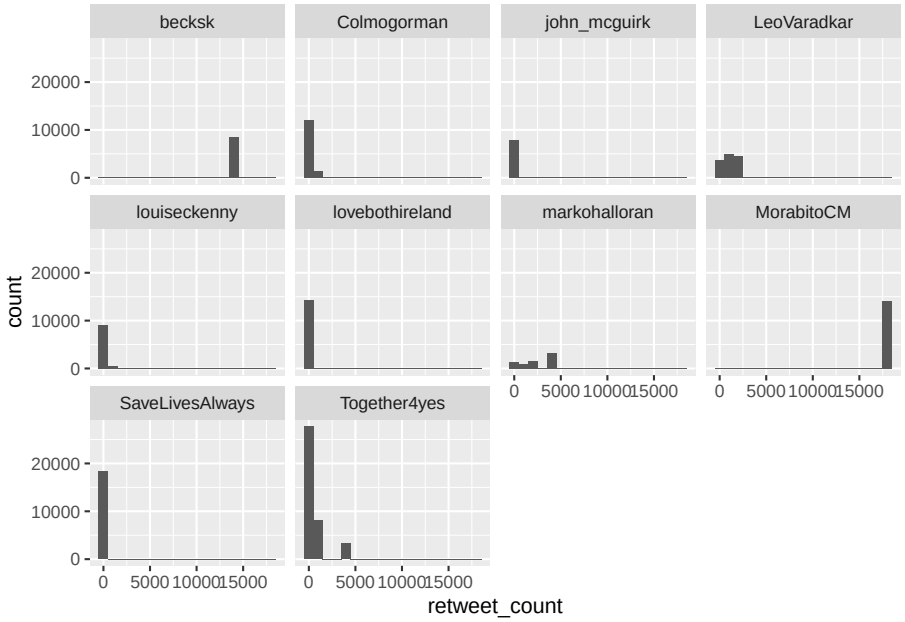
```
df %>% arrange(desc(indegree)) %>% as_tibble() %>% head(10)
```

```
## # A tibble: 10 x 6
##    name            indegree outdegree max_friends max_followers date_created
##    <chr>              <dbl>     <dbl>       <int>         <int> <chr>
##  1 Together4yes       39087       782         823         24458 Fri Feb 02 10:4~
##  2 SaveLivesAlways    18353       401         546          4202 Wed Jun 07 22:1~
##  3 lovebothireland    14366       241         732          4634 Thu Oct 06 13:5~
##  4 MorabitoCM         14184         0        1227          1026 Thu Jul 16 06:4~
##  5 Colmogorman        13454      1104         912         59411 Sat Apr 18 12:0~
##  6 LeoVaradkar        13138        16        1785        451116 Wed Dec 22 13:3~
##  7 louiseckenny        9556       314        1941         13975 Sat Aug 01 21:3~
##  8 becksk              8563        41        1774          1078 Mon May 18 23:5~
##  9 john_mcguirk        7759       106        2470         24676 Thu Mar 12 13:5~
## 10 markohalloran       7046      1113        3535         18512 Tue Apr 21 23:2~
```

First use `pull()` to get a vector of the top ten accounts. Next, only include retweets by them. Then use `ggplot` to look at the distribution of those tweets - whether they were concentrated or dispersed.

```
top_in_degree = df %>%
  arrange(desc(indegree)) %>%
  as_tibble() %>%
  head(10) %>%
  pull(name)
```

```
repeal_tweets %>%
  filter(retweet_screen_name %in% top_in_degree) %>%
  ggplot() +
  geom_histogram(aes(retweet_count), binwidth = 1000) +
  facet_wrap(~retweet_screen_name)
```



This shows that many of these accounts were retweeted consistently, but that some (becksk and MorabitoCM) had a single viral tweet, accounting for their high in-degree.

```
df %>% arrange(desc(outdegree)) %>% as_tibble() %>%
  left_join(user_info, by = c('name' = 'user_screen_name'))
```

```
## # A tibble: 252,296 x 9
##    name          indeg~1 outde~2 max_f~3 max_f~4 date_~5 max_f~6 max_f~7 date_~8
##    <chr>           <dbl>   <dbl>   <int>   <int> <chr>     <int>   <int> <chr>
##  1 kaydnan             0    7524     173     136 Wed Ma~     173     136 Wed Ma~
##  2 Irishprolifer    1631    4707    7930    7087 Wed No~    7930    7087 Wed No~
##  3 mobyrne100       1100    3809    2498    1050 Sat Ju~    2498    1050 Sat Ju~
##  4 MaryOGrady8         6    3785     183     863 Wed Ja~     183     863 Wed Ja~
##  5 christi85573~      54    3125    5003    2432 Tue Ju~    5003    2432 Tue Ju~
##  6 Declan1497          3    3042    2390    1110 Sun Ma~    2390    1110 Sun Ma~
##  7 DunL4Choice       943    2921     911    1434 Sat Se~     911    1434 Sat Se~
##  8 BernadetteCo~       4    2641     231     189 Sun Fe~     231     189 Sun Fe~
```

```
##  9 EmmaMurphy12~    3607    2439    3932    3149 Fri Au~    3932    3149 Fri Au~
## 10 Paul71           1843    2066    3894    1566 Thu Ma~    3894    1566 Thu Ma~
## # ... with 252,286 more rows, and abbreviated variable names 1: indegree,
## #   2: outdegree, 3: max_friends.x, 4: max_followers.x, 5: date_created.x,
## #   6: max_friends.y, 7: max_followers.y, 8: date_created.y
```

12.4.1 PageRank

However, these numbers simply tell us how many times a given node retweeted, and nothing about their influence in the network. For example, the top user by outdegree is not retweeted by others at all, which means they were very active, but their reach was limited: they were not having their own tweets broadcast by others.

Another way to calculate the influence of a user in this network is to look at their PageRank score. This is a recursive metric, which scores a node highly if they are retweeted by other importance nodes. It is particularly appropriate here, because it means that the 'value' of a node's tweets is weighted by how often its retweeters are themselves retweeted.

```
pr_scores = retweet_graph %>%
  mutate(page = centrality_pagerank(weights = weight)) %>%
  mutate(degree = centrality_degree(mode = 'in', weights = weight)) %>%
  as_tibble()

pr_scores %>% arrange(desc(page)) %>% head(10)
```

```
## # A tibble: 10 x 3
##    name               page degree
##    <chr>             <dbl>  <dbl>
##  1 Together4yes    0.0300   39087
##  2 LeoVaradkar     0.00971  13138
##  3 newschambers    0.00886   6868
##  4 Colmogorman     0.00854  13454
##  5 MorabitoCM      0.00844  14184
##  6 gavreilly       0.00770   5551
##  7 SaveLivesAlways 0.00706  18353
##  8 shannonmaile_   0.00672   1531
##  9 becksk          0.00638   8563
## 10 louiseckenny    0.00492   9556
```

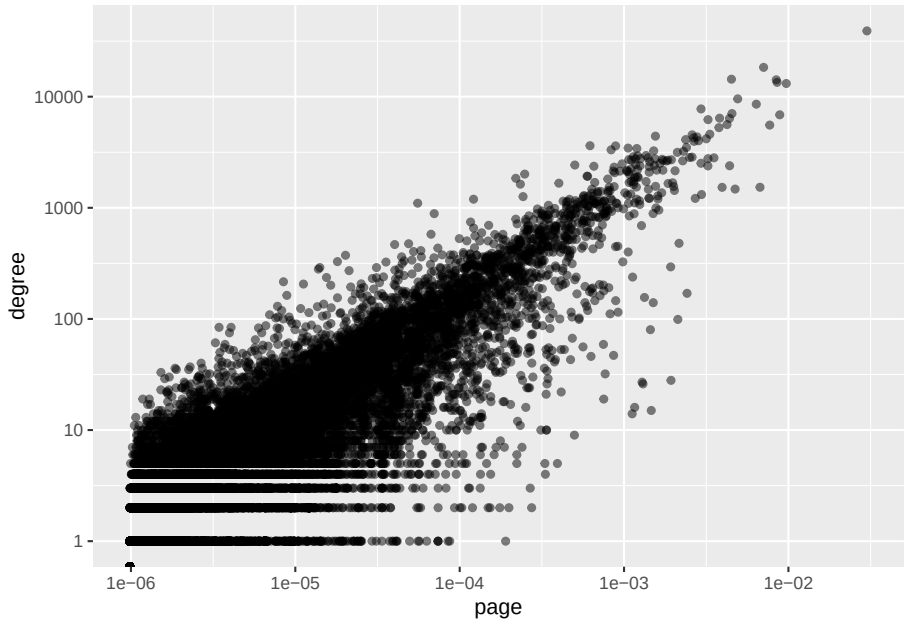
This list looks very similar to the list of those with the highest in-degree, but there are some differences if we compare the two. The accounts newschambers, gavoreilly, and shannonmaile_ are new to the top ten.

We can also plot the relationship between PageRank and in-degree, to get an idea of just how closely related these two measurements are.

To make it easier to read, we use a log scale with `scale_x_log10()` and `scale_y_log10()`. This is useful when your data has a skewed distribution like this.

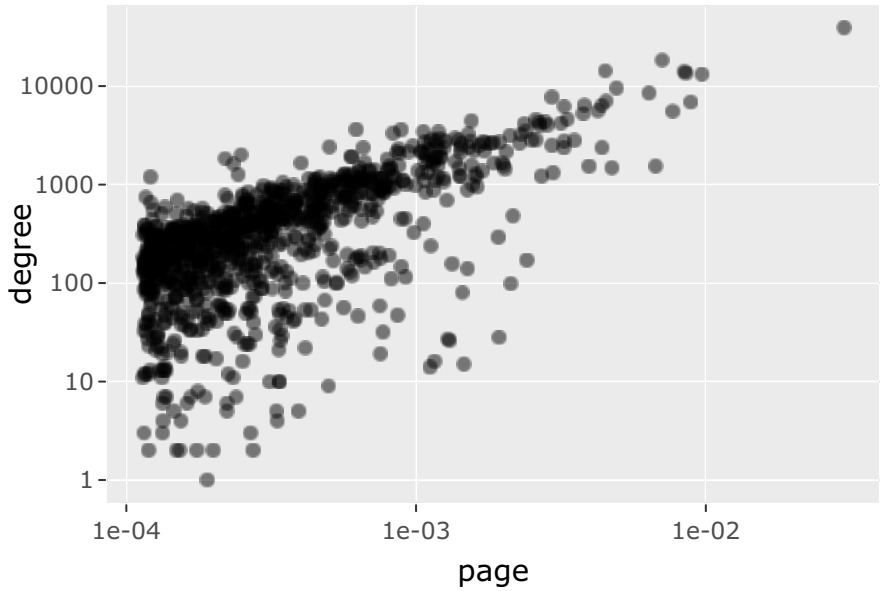
```
pr_scores %>%
  arrange(desc(page))  %>%
  ggplot(aes(text =name)) +
  geom_point(aes(page, degree),alpha = .5) + scale_x_log10() + scale_y_log10()
```

```
## Warning: Transformation introduced infinite values in continuous y-axis
```



We can easily turn this into an interactive graph using another R library, Plotly. This can be very useful for this kind of exploratory data analysis, as it allows you to quickly zoom in on individual points. For this, we make new version of the data, only looking at the top 1000-ranked for PageRank.

```
p = pr_scores %>% slice_max(page, n = 1000) %>%
  arrange(desc(page))  %>%
  ggplot(aes(text =name)) +
  geom_point(aes(page, degree),alpha = .5) + scale_x_log10() + scale_y_log10()

plotly::ggplotly(p)
```



Of particular interest are points (nodes) in the bottom-left of the graph, which have high PageRank despite not having the highest in-degree overall. Several of these are journalists or news sites, for example the RTE news anchor Dyayne Connor, and the 'morningireland' account.

12.5 Some other questions you might ask of this data?

Which are being consistently retweeted, and which just have one 'big' tweet which goes viral?

Is there a difference between the 'yes' and 'no' camps in terms of the way they are retweeted etc?

Are there any genuinely 'in the middle'?

12.5.1 Community Detection

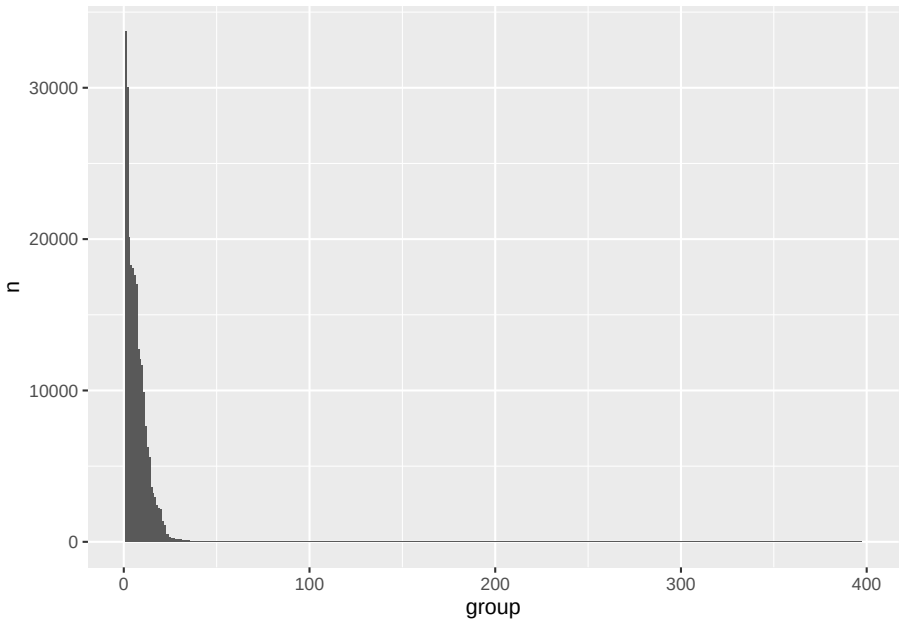
The power of networks, as we've said, is that we can consider groups as well as individuals and individual counts of tweets or importance. We can do this with community detection, to look at whether there are distinct groups within this network. We'll make a new version of the network, this time undirected, because the most widely-used community detection algorithm only works

with undirected networks. Then we'll calculate the group membership using `group_louvain()`.

```
retweet_graph_u = repeal_tweets %>%
  filter(! retweet_screen_name=='') %>%
  count(user_screen_name, retweet_screen_name, name = 'weight') %>%
  as_tbl_graph(directed = FALSE) %>% mutate(comp = group_components()) %>% filter(comp ==1) %>%
  mutate(group = group_louvain())
```

If we do this, we see that there are a large number of groups, over 3000, but that many of them are just a few nodes. We can remove these 'disconnected components' first, to make things easier to work with. Now there are about 400 groups, the largest being 30,000 nodes.

```
retweet_graph_u %>% as_tibble() %>% count(group) %>% ggplot() + geom_col(aes(x = group, y = n))
```



12.6 Adding other analysis: text mining

In order to understand this data, we can use text mining techniques on the tweets or hashtags, which should give a clue as to the makeup of each of the groups. We'll concentrate on the ten largest groups.

```
df_t = retweet_graph_u %>% as_tibble() %>%  left_join(repeal_tweets, by = c('name' = 'u
```

```
hashtags = df_t %>% select(group, hashtags) %>% tidytext::unnest_tokens(output = token
```

```
hashtags_tf = hashtags %>% count(group, token) %>% tidytext::bind_tf_idf(token, group,
```

```
hashtags_tf_top = hashtags_tf %>% group_by(group) %>% slice_max(tf, n = 50) %>% filter
```

```
hashtags_tf_top %>%
  mutate(h = paste0(token, " (", n, ")")) %>%
  group_by(group) %>% summarise(hashtags = paste0(h, collapse = '; ')) %>%
  DT::datatable() %>%
  DT::formatStyle(columns = 1:3, fontSize = '70%')
```

Show 10 entries Search:

	group		hashtags
1	1	repealthe8th (38784); hometovote (35746); together4yes (10179); 8thref (4792); togetherforyes (3761); voteyes (3682); repeal (2981); ireland (1971); repealedthe8th (1669); grandfathers4yes (1485); repealtheeighth (1388); yes (1144); savethe8th (980); trustwomen (974); referendum2018 (754); hometovoteyes (706); itstime (685); london (630); repeal8th (568); repealth8th (528); repealthe8h (363); together2vote (324); nowforni (314); 8thamendment (275); prochoice (259); hometoveote (248); abroadforyes (247); bemyyes (245); votermotor (229); motorvoter (225); abortionreferendum (210); repealthe8 (209); repealtheeigth (203); womensrights (197); repealed (193); men4yes (187); mybodymychoice (187); abortion (185); loveboth (184); ivotedyes (180); northernireland (163); solidarity (159); savita (154); women (152); toronto (142); táformná (139); irish (134); progressivehealthcare (127); rtept (127); farmers4yes (126)
2	2	repealthe8th (124407); together4yes (98012); 8thref (43954); togetherforyes (37441); hometovote (30236); voteyes (21333); repeal (9958); repealedthe8th (7423); cblive (5057); men4yes (5023); trustwomen (4563); savethe8th (4000); loveboth (3962); itstime (3934); yes (3865); rtept (3748); whoneedsyouryes (3381); repealtheeighth (3345); 8thamendment (2890); repealth8th (2870); repeal8th (2805); votermotor (2596); latelateshow (2545); referendum2018 (2360); together2vote (2256); motorvoter (2217); pkshow (2096); ireland (2011); latelate (1781); ivotedyes (1723); stoppunishingtragedy (1723); repealfacts (1667); youryesmatters (1480); tonightv3 (1472); roscommon4yes (1389); doctors4yes (1163); corksaysyes (1161); savita (1155); waterford (1153); repealed (1092); farmers4yes (1083); prochoice (994); hometovoteyes (968); menforyes (962); register4yes (960); loveboats (900); repealthe8h (840); donegalsaysaye (818); time4choice (800); bemyyes (738)
3	3	repealthe8th (20599); hometovote (5943); savethe8th (3168); together4yes (3066); 8thref (3028); voteyes (2988); referendum2018 (2283); scotref (1553); togetherforyes (1334); indyref2 (1286); ireland (1085); repealedthe8th (830); youyesyet (678); mayhem (667); istandwithnicola (630); repeal (573); yes (543); repealtheeighth (532); itstime (511); endtorymisrule (442); freescotland (408); communistscumbagfeministgoblins (399); idonotconsent (345); abortionreferendum (302); trustwomen (299); hometovoteyes (246); brexitshambles (211); repealth8th (196); dup (191); abortion (161); grandfathers4yes (158); snp (158); indyref (154); 8thamendment (152); nowforni (152); hometoveote (148); repeal8th (145); repealthe8h (143); northernireland (141); brexit (139); endthefarce (127); yesireland (125); scotland (109); equalmarriage (98); prochoice (95); together2vote (93); repealthe8 (85); frackoff (80); pisaf (79); men4yes (76)
4	4	savethe8th (47920); 8thref (32442); lovebothvoteno (26817); voteno (11827); votenotoabortion (10289); repealthe8th (10286); prolife (5384); lifecanvass (4917); loveboth (4446); ireland (3649); hometovote (3459); together4yes (3450); abortion (3317); cblive (3131); rtept (2706); jointherebellion (2402); 8thamendment (2249); toofarforme (2059); latelate (2049); togetherforyes (1847); ivotedno (1793); tonightv3 (1109); hometovoteno (886); pkshow (872); no (854); latelateshow (785); irlanda (780); redarmy (779); repeal (774); savelives (763); votenoroadshow (761); trustwomen (749); standupforlife (737); abortionreferendum (671); liveline (658); dublin (622); todaysor (614); chooselife (589); lovethemboth (510); repealedthe8th (469); cork (463); voteyes (459); referendum2018 (444); men4yes (433); yes (387); dublincity (383); aborto (365); savethe8thvoteno (348); prochoice (337); irish (329)
5	5	repealthe8th (8985); hometovote (7049); together4yes (1488); ireland (1284); itstime (1051); 8thref (813); voteyes (757); togetherforyes (504); mybodymychoice (503); plannedparenthood (495); smartdissent (490); trustwomen (410); repealtheeighth (385); prochoice (322); yes (303); repealedthe8th (258); suebellcobbforgov (249); santafehighschool (215); repeal (213); abortion (207); hometovoteyes (207); vi (185); irelandreferendum (173); savitahalappanavar (163); 8thamendment (151); evangelical (151); altrighteous (149); humanity (149); suebellcobb2018 (142); gunreformnow (137); savethe8th (133); womenempowerment (120); enoughisenough (118); santafe (116); santafemassacre (116); texasgovernor (115); reprorights (108); homeforyes (104); referendum2018 (102); repealthe8h (92); theresistance (88); repealth8th (82); womensrightsarehumanrights (79); wearorange (78); womensrights (75); irish (69); savita (69); grandfathers4yes (65); righttochoose (59); abortionreferendum (58)
6	6	repealthe8th (14386); together4yes (8975); 8thref (6296); hometovote (5590); togetherforyes (4172); voteyes (2980); repeal (1498); yes (783); repealedthe8th (669); repealtheeighth (639); savethe8th (617); trustwomen (614); ireland (445); together2vote (401); referendum2018 (380); repealth8th (376); men4yes (373); rtept (332); repeal8th (310); prochoice (297); progressivehealthcare (288); 8thamendment (287); pkshow (242); itstime (221); ivotedyes (215); loveboth (213); cblive (203); repealthe8h (181); reapealthe8th (169); theweightofftheeighth (141); repealed (133); latelateshow (130); hometovoteyes (114); menforyes (113); referendum (111); grandfathers4yes (104); stoppunishingtragedy (104); repealthe8 (98); abortion (89); womensrights (88); bemyyes (85); latelate (85); youryesmatters (83); whoneedsyouryes (81); mybodymychoice (79); no (76); tá (76); abortionreferendum (72); savita (71); repealtheeigth (69)
7	7	repealthe8th (14028); hometovote (10245); together4yes (1207); 8thref (767); ireland (672); voteyes (637); togetherforyes (550); itstime (548); savethe8th (517); repeal (354); pope (303); popefrancis (303); archbishopofadelaide (299); catholicclergy (299); childsexualabuse (299); convicted (299); coveredup (299); yes (285); abortion (280); repealedthe8th (270); repealtheeighth (213); 100yearsofevil (195); papalsecrecy (195); philipwilson (195); irish (132); savita (126); repealth8th (110); harveyweinstein (108); criminalclergy (107); sack (107); vatican (107); vaticano (107); archbishopphilipwilson (106); now (106); referendum2018 (106); repeal8th (93); repealthe8h (93); dublin (79); abortionreferendum (76); grandfathers4yes (75); humanrights (74); trustwomen (72); mybodymychoice (67); bernardshaw (66); womensrights (63); hometovoteyes (56); metoo (55); pastlimenocrime (55); 8thamendment (51); marriageequality (51)
8	8	repealthe8th (1432); hometovote (525); together4yes (211); voteyes (175); 8thref (119); togetherforyes (79); repealtheeighth (66); repeal (45); savethe8th (34); repealedthe8th (31); itstime (19); ireland (18); pulangmengundi (18); referendum2018 (18); repealth8th (18); abortionreferendum (15); yes (13); repealthe8h (10); repeal8th (9); trustwomen (9); grandfathers4yes (8); hometovoteyes (8); ivotedyes (8); abortionrights (7); mybodymychoice (7); cblive (6); loveboth (6); prochoice (6); aborto (5); northernireland (5); repealthe8 (5); tá (5); together2vote (5); votáilitá (5); animalsthemovie (4); notinmyname (4); savita (4); womensrights (4); bemyyes (3); eurovision (3); hearmeout (3); hometoveote (3); lasttotravel (3); latelateshow (3); men4yes (3); pkshow (3); yesforchoice (3); NA (3); clairebyrnelive (2); communistscumbagfeministgoblins (2); dogsatpollingstations (2); healthcarenotairfare (2); irl (2); irlanda (2); itsover (2); latelate (2); neveragain (2); no (2); notallheroeswearcapes (2); repealtheeight (2); repealtheeigth (2); rt (2); rtept (2); save8th (2); tonightv3 (2); vote2018 (2); voteno (2); werepealedthe8th (2); womensrightsarehumanrights (2)
9	9	8thref (12670); repealthe8th (5184); hometovote (2393); together4yes (1546); togetherforyes (919); voteyes (908); repeal (394); ireland (392); 8thamendment (370); repealedthe8th (370); referendum2018 (288); yes (243); savethe8th (234); abortionreferendum (188); repealtheeighth (188); mybodymychoice (165); donegal (138); ntfm (137); justiceformouna (132); reif8u (132); savenoura (132); timesup (130); metoo (120); trustwomen (118); repealth8th (108); itstime (101); rteexitpoll (99); savita (97); repeal8th (95); savitahalappanavar (93); voterfraud (81); rtept (79); referendum (73); repealthe8h (72); ivotedyes (68); repealed (64); carlowkilkenny (62); eurovision (60); kerry (60); men4yes (60); latelate (59); dublin (58); together2vote (58); iestaff (57); loveboth (55); abortion (54); cblive (53); rosgal (53); pkshow (44); exitpoll (43)
10	10	repealthe8th (7737); voteyes (1720); hometovote (1551); together4yes (1146); 8thref (653); togetherforyes (537); repeal (363); repealtheeighth (317); repealedthe8th (289); yes (237); ireland (187); savethe8th (173); trustourwomen (132); referendum2018 (117); repeal8th (90); trustwomen (88); repealth8th (74); repealthe8h (74); repealthe8 (71); itstime (60); hometovoteyes (54); ivotedyes (52); men4yes (52); together2vote (51); 8thamendment (46); abortionreferendum (45); prochoice (43); grandfathers4yes (38); mybodymychoice (36); repealtheeigth (29); pkshow (27); cblive (21); loveboth (19); abortion (18); repealedtheeighth (18); savita (17); womensrights (17); irelandreferendum (14); repealed (14); voteno (13); yesforchoice (13); youryesmatters (13); choice (12); latelate (12); london (12); menforyes (12); humanrights (11); nowforni (11); tonightv3 (11); women (11); yesvote (11)

Showing 1 to 10 of 30 entries Previous 1 2 3 Next

Listing the top hashtags for each group shows that most groups seem to include mostly pro 'yes' campaign hashtags, except one, group 4, which is mostly 'no' campaign. Unsurprisingly for such a divisive issue, the groups are divided along ideological lines.

12.7 Exporting the edge list for Gephi.

The network is too large to visualise in R. We can create an edge list and use it in another software, Gephi, which is better suited for visualising very large networks:

```
repeal_tweets %>%
  filter(! retweet_screen_name=='') %>%
  count(user_screen_name, retweet_screen_name, name = 'Weight') %>%
  select(Source = user_screen_name, Target = retweet_screen_name) %>% write_csv('repeal_for_gephi
```

12.7.1 Conclusions

- Helps when working with big data
- Keep in mind what you are analysing - not a full social network but a specific network of retweets around a particular subject. So conclusions should be grounded in that.

12.7.2 Networks and regression

12.7.3 Statistical approaches to networks - ERGM and SOAM (RSiena)

Chapter 13

Week 6, Class 2: Reflections and Pitfalls of Networks

13.1 What are implications of networks?

Hierarchies

'Rich get richer' effect

13.2 Network representations

How do we communicate that this is one representation/model of data

13.3 Problems with network representation

Pitfalls etc.

13.3.1 Missing data

13.3.2 Specific problems with networks and cultural/humanities data

13.4 Visualisations

> "There is a tendency when using graphs to become smitten with one's own data. Even though a graph of a few hundred nodes quickly becomes unreadable, it is often satisfying for the creator because the resulting figure is elegant and complex and may be subjectively beautiful, and the notion that the creator's data is"complex" fits just fine with the creator's own interpretation of it. Graphs have a tendency of making a data set look sophisticated and important, without having solved the problem of enlightening the viewer." — Ben Fry, *Visualizing Data: Exploring and Explaining Data with the Processing Environment* (Sebastopol, CA: O'Reilly Media, 2007)

Do you agree with this statement? Another viewpoint is here: https://gephi.wordpress.com/2011/10/12/everything-looks-like-a-graph-but-almost-nothing-should-ever-be-drawn-as-one/

Chapter 14

Week 7: Final Project

There are two stages to the final project:

- In the final week of the course, either December 14 or 16, you'll be assigned a 10-minute presentation slot, where you should outline your progress so far, including your research question, datasets used, any problems you encountered (or think you might). This is a chance to get feedback from your peers and course leaders.
- The final project itself is due on **Tuesday, January 17.**

14.1 Presentation:

A good presentation should try and cover the following:

- Outline your research question
- What are the steps you need to carry out?
- How will you get your data? What size is the dataset, is it manageable with your resources, how long will it take to put together (for example if you are hydrating tweets, how long will this take?)
- What expertise do you need, what do you have and where will you find help if needed?
- Do you have any existing examples you'd like to emulate?
- What problems might arise and how are you going to deal with them?

14.2 What should the final project look like?

The final project tasks you with using your network data and related data model to tackle an interesting research questions. The project should take the form of an R Markdown notebook, as you have learned to create in the first few weeks of the course. An R Markdown document is a format which allows you to combine text, chunks of code, and the output of those chunks. You'll write up this document and then turn it into a HTML page - a process known as 'knitting'.

14.2.1 Practical steps

Gather your datasets and make sure they are available. If you use a 'non standard' dataset, export a copy of the final dataset used and include it along with the knitted HTML file and the original R Markdown.

14.2.2 Project Aim

The aim is to communicate to a reader the value of your project and its implications. To do this, the project should take the form of 'literate programming', meaning that it should contain code, visualisations, derived data, and text - it should read not just as a piece of code and its output, but as a report and discussion on your findings. Interpretation and discussion is essential.

You can also include supplementary elements, for example data.

Include the original markdown, all data used, and the knitted html version.

Some examples of literate programming final projects

14.3 R Markdown

The final project takes the form of a **notebook** - think of this as a modern-day 'lab notebook', where you share the steps you carried out, but also what you were thinking when you did them, the rationale behind them, and the conclusions you came to. This allows you to communicate your findings, and others to reproduce your steps, but also understand your thought processes and why you took the decisions you did.

We have already gone through R Markdown in detail, but it has a number of other features/packages you can use if you want to extend your final project. The packages VisNetwork and Plotly allow you to include interactive visualisations within the report. The DT package allows you to do the same for tables. These elements are not essential, but can help to communicate your results to the reader in some cases.

14.3.1 Reproducible research

The point of this is that it allows anyone to reproduce your results. This is done by including your code and data within the report. This is done automatically if you use R Markdown, because the code is included by default in your outut. You can use external sources and code/applications, but you should explain what you have used in detail, so that others can follow the steps you've carried out and (hopefully) reach the same conclusions.

14.3.2 Questions

Your final project should be structured to answer the following questions:

- **Introduction** - what is your question/problem you're trying to solve?
- **Datasets** - what datasets are you using? Where did you get them? Is it openly-available? Do they have missing data? Did you need to do pre-processing such as data harmonisation?
- What is your **data model**? How does your data relate to itself?
- **Method** - How did you create the network? What subset of your data did you use and why did you choose it? What additional information about your network (attributes) did you use?
- What are the basic **network statistics** of this network? With your 'finished' network, how many nodes and edges are there? What is the density? Is it clustered or very dense?
- Outline the particular **network methods** you used - community detection, centrality, global network metrics, others?
- **Findings**. What are your conclusions? What has the data and network told you?
- **Discussion**. What are the implications for this research/method? What does it tell us? What didn't work? What are the pitfalls of looking at your data like this? What else did you try that didn't work? Were you able to use visualisations successfully?

14.4 Final Project Datasets

As well as the ESTC and Twitter datasets we have provided for the final project, you can use any network dataset you like. In every case, you should reflect critically on the dataset used, provide it in full for reproducibility, and make a note on what bias it might have and how this might affect your findings.

Almost any humanities data can be represented as a network - particularly bipartite. You should outline the dataset you intend on using during the final project presentation. Many of these datasets will require the extraction of structured data or cleaning before they are suitable for network analysis, so bear in mind the time and expertise that will take.

14.4.1 Available datasets

Below are some other datasets or collections of datasets which might be useful. Important to note: don't just pick a dataset because you think it might make an interesting network graph. **Think of the research question you would like to tackle first.**

Dataset collections:

Stanford Large Network Dataset Collection

UCIrvine Network Data Repository

Correspondence data:

Willem I van Oranje-Nassau: letters and network in 1572

Correspondence of Daniel van der Meulen (1554 - 1600)

There is also another dataset of early modern correspondence available, from the State Papers - ask Yann for more details.

Citation/Co-occurrence networks

Citation network on Kaggle

Marvel Universe Social Network (a good example of a bimodal network, that should generally be projected to a unimodal one for analysis)

Game of Thrones Books Character Network (and series)

Other cultural data:

Project Gutenberg (for full texts)

RECIRC Project(database of texts by female authors - individual searches can be exported as a .csv file for use in network analysis)

Spotify API (as we used in a previous class)

Chapter 15

Appendix and extra materials

Useful as examples for the final project - things we didn't have time to cover in class.

15.1 Ego Networks

Very often when applying network analysis to humanities datasets, you'll be working with *ego networks*. An ego network is a network seen from the perspective of a focal node, known as the ego, and all the nodes to which it is connected, known as *alters*. In a true ego network, the network will include all the links *between* the alters.

These types of networks are very common in real-world networks because data is often collected from the perspective of an ego node or a number of ego nodes, for example, in a survey, or perhaps in a collection of correspondence. Because of this it is important to understand what they are, and think about whether the data is representative.

At the same time, ego network data can be very useful. Studies have shown that many metrics in an ego network correlate strongly with metrics from a full network (meaning that if an alter is central to the ego network, it is likely also central if we had the full network data too.

Network analysis from the point of view of egos can tell us about the structure of a particular node and its relationship to its neighbours. Comparing many ego networks can be used to good effect. Lerman et al (2022) created a list of features (attributes) for the citation ego networks of authors elected to the

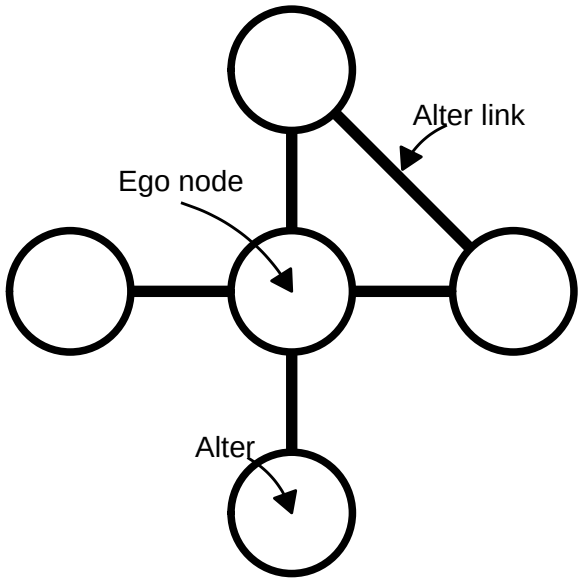


Figure 15.1: Example ego network

National Academy of Sciences, successfully using those features to predict the gender of the author:

> Women's ego networks have higher average degree, edge density, and clustering coefficient. Together, these features suggest that women are more tightly embedded within their research communities. This is consistent with previous findings that women tend to gravitate to certain communities (11). Women have fewer peers than men, but these peers are more productive (publish more papers) and receive more citations. Finally, women NAS members have more women among their peers.

Many of the datasets you might be considering using for your final project may in fact be ego networks. It is worth thinking about the implications of this, and considering which kinds of metrics, or comparisons would work best. Is each ego network collected using the same method? If not, does this skew the results?

15.2 Case Study: A Spotify Ego Network

This section walks through the code for creating and analysing an ego network drawn from a very different type of humanities dataset: Spotify's 'related' artists information. This is available through Spotify's API. One function of the API

lets us download the twenty 'most-related' artists to a given 'seed' artist. The algorithm for how the most-related are calculated is not public, but it's likely by comparing overlapping listeners (if lots of people listen to both artist X and artist Y, those artists are marked as related).

15.2.0.1 Requirements

If you want to complete the (optional) exercise in the corresponding file on CSC notebooks, you'll need to sign up for a Spotify developer account. Once you've done this, create a new application using the dashboard. This will give you the necessary client ID and password. You'll then need to install the `spotifyr` package using `install.packages('spotifyr)`.

An exercise relating to the bipartite network below is another option - if for whatever reason you don't want to sign up for an account, or have difficulty accessing the API. The data for this is already available.

15.2.0.2 Get related artists from the Spotify API.

The `spotifyr` package makes it easy to access the Spotify API functions using R.

Once you have installed the spotifyr package, you need to tell it where to find your user ID and password. Copy these from your dashboard on the Spotify API page. Then, use the following code to temporarily store them in R's memory:

```
#Sys.setenv(SPOTIFY_CLIENT_ID = '') # uncomment these lines and add your client ID and password
#Sys.setenv(SPOTIFY_CLIENT_SECRET = '')
```

Next, you'll use the command `get_spotify_access_token()` to use this ID and password to generate an access token.

Following this, we need to find the spotify ID for the 'ego' artist. The easiest way to do this is with the `get_artist_audio_features()` function. This can take a character string and will return the closest match, if an ID is not specified. To get the ID for the band Radiohead, we do the following:

```
# install.packages('spotifyr') # install the package if necessary

library(spotifyr)


access_token <- get_spotify_access_token()
```

```
radiohead <- get_artist_audio_features('radiohead')

glimpse(radiohead)
```

```
## Rows: 193
## Columns: 39
## $ artist_name                  <chr> "Radiohead", "Radiohead", "Radiohead", "R~
## $ artist_id                    <chr> "4Z8W4fKeB5YxbusRsdQVPb", "4Z8W4fKeB5Yxbu~
## $ album_id                     <chr> "6ofEQubaL265rIW6WnCU8y", "6ofEQubaL265rI~
## $ album_type                   <chr> "album", "album", "album", "album", "albu~
## $ album_images                 <list> [<data.frame[3 x 3]>], [<data.frame[3 x ~
## $ album_release_date           <chr> "2021-11-05", "2021-11-05", "2021-11-05",~
## $ album_release_year           <dbl> 2021, 2021, 2021, 2021, 2021, 2021, 2021,~
## $ album_release_date_precision <chr> "day", "day", "day", "day", "day", "day",~
## $ danceability                 <dbl> 0.296, 0.630, 0.488, 0.167, 0.165, 0.402,~
## $ energy                       <dbl> 0.463, 0.428, 0.754, 0.302, 0.146, 0.757,~
## $ key                          <int> 5, 5, 2, 6, 6, 7, 0, 3, 2, 7, 7, 0, 11, 1~
## $ loudness                     <dbl> -11.412, -15.520, -8.552, -11.644, -21.35~
## $ mode                         <int> 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0,~
## $ speechiness                  <dbl> 0.0449, 0.0358, 0.0378, 0.0345, 0.0362, 0~
## $ acousticness                 <dbl> 7.05e-01, 2.62e-01, 2.37e-03, 3.16e-01, 8~
## $ instrumentalness             <dbl> 4.82e-02, 8.52e-01, 8.51e-01, 7.97e-01, 8~
## $ liveness                     <dbl> 0.0954, 0.2780, 0.2240, 0.1100, 0.1090, 0~
## $ valence                      <dbl> 0.0629, 0.1590, 0.3880, 0.1900, 0.0577, 0~
## $ tempo                        <dbl> 123.943, 112.923, 91.517, 102.026, 134.50~
## $ track_id                     <chr> "62dUmjtkYOUwYOALt6pefh", "797AyTQcqoQgqW~
## $ analysis_url                 <chr> "https://api.spotify.com/v1/audio-analysi~
## $ time_signature               <int> 5, 4, 4, 4, 3, 4, 4, 3, 5, 4, 3, 4, 4, 4,~
## $ artists                      <list> [<data.frame[1 x 6]>], [<data.frame[1 x ~
## $ available_markets            <list> <"AD", "AE", "AG", "AL", "AM", "AO", "AR~
## $ disc_number                  <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2,~
## $ duration_ms                  <int> 251426, 284506, 351693, 356333, 222600, 3~
## $ explicit                     <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,~
## $ track_href                   <chr> "https://api.spotify.com/v1/tracks/62dUmj~
## $ is_local                     <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,~
## $ track_name                   <chr> "Everything In Its Right Place", "Kid A",~
## $ track_preview_url            <chr> "https://p.scdn.co/mp3-preview/74e765ad68~
## $ track_number                 <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 1, 2, ~
## $ type                         <chr> "track", "track", "track", "track", "trac~
## $ track_uri                    <chr> "spotify:track:62dUmjtkYOUwYOALt6pefh", "~
## $ external_urls.spotify        <chr> "https://open.spotify.com/track/62dUmjtkY~
## $ album_name                   <chr> "KID A MNESIA", "KID A MNESIA", "KID A MN~
## $ key_name                     <chr> "F", "F", "D", "F#", "F#", "G", "C", "D#"~
## $ mode_name                    <chr> "minor", "major", "major", "minor", "majo~
## $ key_mode                     <chr> "F minor", "F major", "D major", "F# mino~
```

This returns all of Radiohead's tracks along with some more information. The second column in each row is the `artist_id`, in this case `4Z8W4fKeB5YxbusRsdQVPb`, which we will use as the input for further functions.

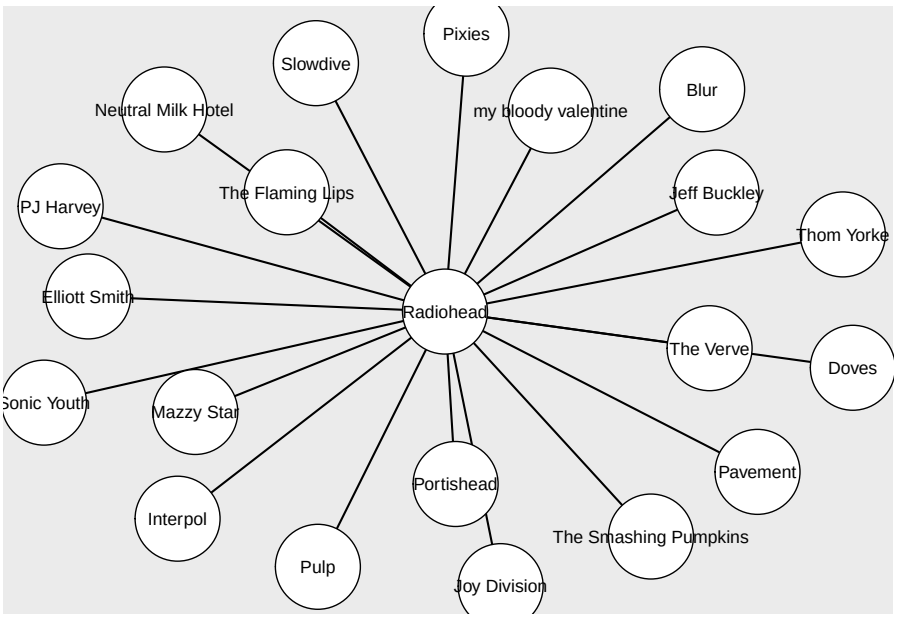
The function to get the related artists to one artist is `get_related_artists()`. The below syntax will get Radiohead's related artists list:

```
radiohead_related = spotifyr::get_related_artists('4Z8W4fKeB5YxbusRsdQVPb')

glimpse(radiohead_related)
```

```
## Rows: 20
## Columns: 11
## $ genres                <list> <"electronica", "glitch pop", "indie rock", "ox~
## $ href                  <chr> "https://api.spotify.com/v1/artists/4CvTDPKA6W06~
## $ id                    <chr> "4CvTDPKA6W06DRfBnZKrau", "7MhMgCo0Bl0Kukl93PZbY~
## $ images                <list> [<data.frame[3 x 3]>], [<data.frame[3 x 3]>], [~
## $ name                  <chr> "Thom Yorke", "Blur", "Pixies", "Jeff Buckley", ~
## $ popularity            <int> 57, 68, 70, 61, 60, 59, 57, 58, 58, 54, 63, 63, ~
## $ type                  <chr> "artist", "artist", "artist", "artist", "artist"~
## $ uri                   <chr> "spotify:artist:4CvTDPKA6W06DRfBnZKrau", "spotif~
## $ external_urls.spotify <chr> "https://open.spotify.com/artist/4CvTDPKA6W06DRf~
## $ followers.href        <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
## $ followers.total       <int> 875467, 2672511, 2353614, 921399, 1665014, 78011~
```

The result is a dataframe with a list of the 20 closest-related artists to Radiohead, using Spotify's algorithm, with some more information including the artists' popularity and genres - and their Spotify ID. This can be visualised using the method from the previous lesson:

```
library(ggraph)
library(tidygraph)
library(igraph)
radiohead_related %>%
  mutate(ego_name = 'Radiohead') %>%
  distinct( ego_name, name) %>%
  as_tbl_graph() %>%
  ggraph('stress') + geom_edge_link() +
  geom_node_point(size = 20, fill = 'white', color = 'black', pch = 21)+
  geom_node_text(size = 3, aes(label =name))
```



15.2.0.3 'Crawling' through the related artists information

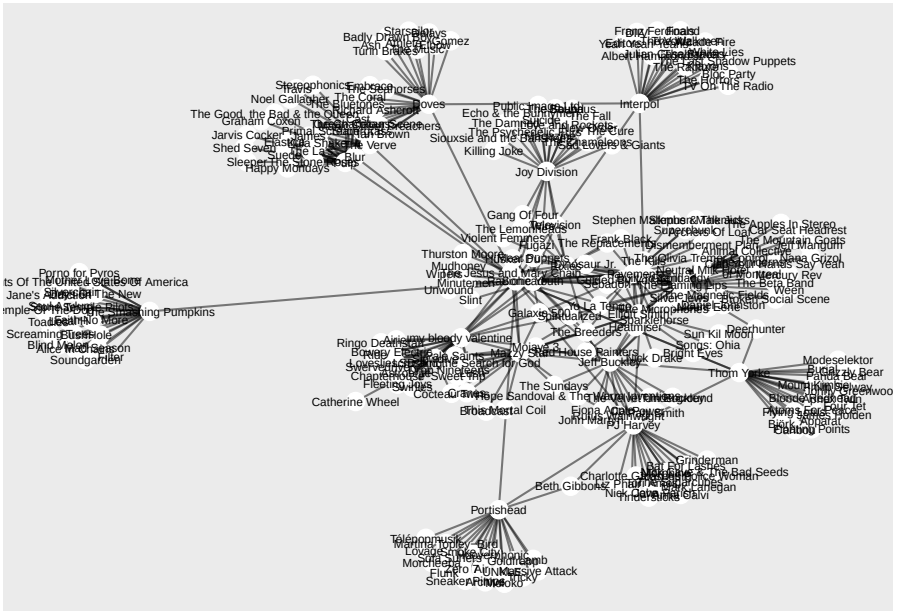
As you might be able to tell, there is very little useful information in this type of ego network: which only includes links to and from the ego node. Each 'alter' has a degree of one, and will have the same scores for all network metrics.

However, we can use the new artists' IDs to 'crawl' through the API and get the 20 most related artists for each of them, taking a snowball sample approach to creating a network. The easiest way to do this is to use a R function `lapply`, which applies a function to every element in a list, and returns a new list.

```
radiohead_related_related = lapply(radiohead_related$id, get_related_artists)

radiohead_related_related = data.table::rbindlist(radiohead_related_related)

radiohead_related_related = radiohead_related_related %>%
  mutate(seed_id = rep(radiohead_related$id, each= 20))%>%
  mutate(ego_name = rep(radiohead_related$name, each= 20))
```

This results in dataset of the twenty related nodes plus all their own connections, which can be drawn as a network:

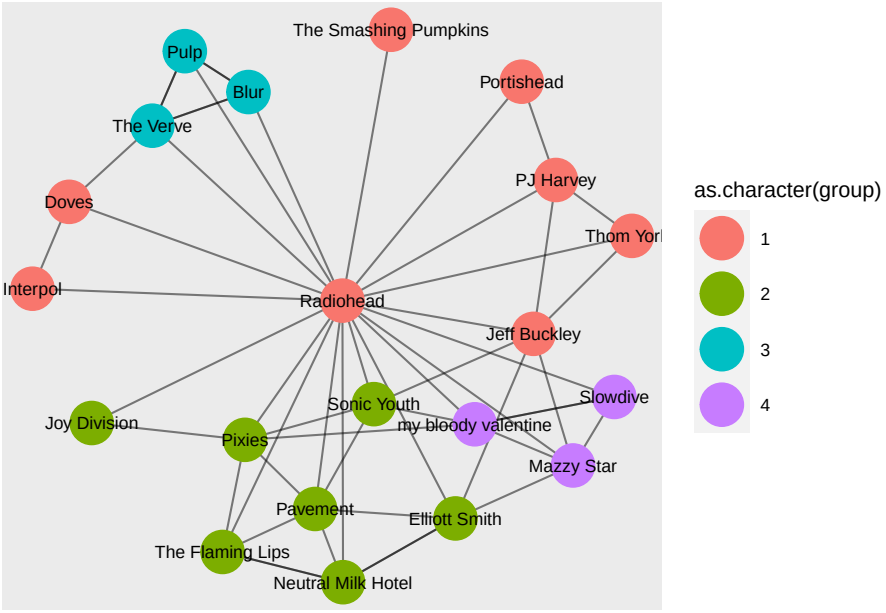
```
g = radiohead_related_related   %>%
  rbind(radiohead_related %>%
  mutate(seed_id = '4Z8W4fKeB5YxbusRsdQVPb')%>%
  mutate(ego_name = 'Radiohead') ) %>%
  distinct(ego_name, name) %>%
  as_tbl_graph(directed = F) %>%
  ggraph('fr')+ geom_edge_link(alpha = .5) +
  geom_node_point(size = 4, color = 'white') +
  geom_node_text(size = 2, aes(label =name))
g
```



This network links all the alters to all 20 of their closest-related artists. You can see that many of the artists are not actually connected to: it is all the alters plus all their additional related artists. To simplify things, we can start with a 'true' ego network, which just includes the ego (Radiohead), the alters (the 20 related artists) and the alter links (any pairs of alters who are also related to each other).

```
radiohead_related_related   %>%
  rbind(radiohead_related %>%
  mutate(seed_id = '4Z8W4fKeB5YxbusRsdQVPb') %>%
  mutate(ego_name = 'Radiohead') ) %>%
  filter(id %in% radiohead_related$id) %>%
  distinct(ego_name, name) %>%
```

```
as_tbl_graph(directed = F) %>%
mutate(group = group_louvain()) %>%
ggraph('fr')+ geom_edge_link(alpha = .5) +
geom_node_point(size = 10, aes(color = as.character(group))) +
geom_node_text(size = 3, aes(label =name))
```



We can run community detection on the network to look for distinct clusters. Adding these alters we can see some clusters already emerging: 90s britpop (The Verve, Blur, Pulp), triphop (Portishead), Shoegaze (Slowdive, Mazzy Star), College rock/indie folk (NMH, Elliot Smith):

This ego network can tell us a number of things:

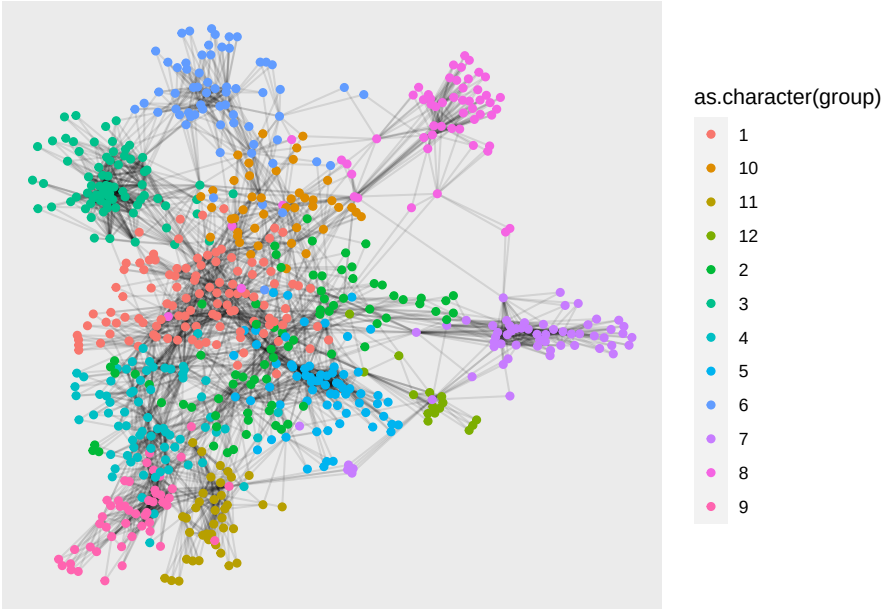
- There are a large number of *triads*, meaning that artists are clustered together (if the Verve and Pulp are connected to Blur, it's also likely that they are connected to each other). What does a clustered network mean in this circumstance?
- There is almost a 'clique' (a group of fully-connected nodes), including all those in green. What does a clique tell us?
- There is one node (Smashing Pumpkins) with no other connections.

As a further step, we can crawl another level deep into the related artists' network. This will give us all the connections at a *third* level removed from Radiohead.

```
level_3_related = lapply(radiohead_related_related$id, get_related_artists)

level_3_related = level_3_related %>% data.table::rbindlist() %>%
  mutate(seed_id = rep(radiohead_related_related$id, each= 20))%>%
  mutate(ego_name = rep(radiohead_related_related$name, each= 20))

glimpse(level_3_related)
```

```
## Rows: 8,000
## Columns: 13
## $ genres                <list> <"electronica", "glitch pop", "indie rock", "ox~
## $ href                  <chr> "https://api.spotify.com/v1/artists/4CvTDPKA6W06~
## $ id                    <chr> "4CvTDPKA6W06DRfBnZKrau", "2f88S1uYsEwP0n4x36wvG~
## $ images                <list> [<data.frame[3 x 3]>], [<data.frame[3 x 3]>], [~
## $ name                  <chr> "Thom Yorke", "Ultraísta", "Modeselektor", "Jonn~
## $ popularity            <int> 57, 31, 46, 45, 38, 38, 55, 51, 36, 44, 49, 48, ~
## $ type                  <chr> "artist", "artist", "artist", "artist", "artist"~
## $ uri                   <chr> "spotify:artist:4CvTDPKA6W06DRfBnZKrau", "spotif~
## $ external_urls.spotify <chr> "https://open.spotify.com/artist/4CvTDPKA6W06DRf~
## $ followers.href        <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
## $ followers.total       <int> 875467, 23427, 219513, 141129, 23544, 112534, 83~
## $ seed_id               <chr> "7tA9Eeeb68kkiG9Nrvuzmi", "7tA9Eeeb68kkiG9Nrvuzm~
## $ ego_name              <chr> "Atoms For Peace", "Atoms For Peace", "Atoms For~
```

```
df = level_3_related %>%
  #filter(id %in% radiohead_related_related$id)%>%
  distinct(id, seed_id) %>%
  as_tbl_graph(directed = F) %>%
  mutate(total_degree = centrality_degree(mode = 'all')) %>%
  filter(total_degree>1) %>%
  mutate(group = group_louvain())


df %>%
  ggraph('fr') +
  geom_edge_link(alpha = .1) +
  geom_node_point(aes(color =as.character(group)))
```



In this extended network, clusters are clearly visible again. There are too many nodes now to label them or easily divide them into genres. Instead, we'll use summarise functions to make a table of information on each of the groups.

```
genres = level_3_related %>% distinct(id, genres, name)



  genres_total  = df %>% as_tibble()%>%
  inner_join(genres, by = c('name' = 'id')) %>%
  unnest(genres) %>%
  count(group,genres) %>%
    arrange(desc(n)) %>%
    mutate(genres = paste0(genres, " (", n, ")")) %>%
  group_by(group) %>%
    slice_max(order_by = n, n = 20) %>%
    summarise(genres_total = paste0(genres, collapse = ' ;'))

artist_total = df %>% as_tibble() %>%
  inner_join(genres, by = c('name' = 'id')) %>%
  mutate(name.y = paste0(name.y, " (", total_degree, ")")) %>%
  arrange(desc(total_degree)) %>%
  group_by(group) %>% slice_max(order_by = total_degree, n = 20) %>%
  summarise(artists = paste0(name.y, collapse = ' ;'))
```

```
genres_total %>% left_join(artist_total) %>% DT::datatable()%>%
  DT::formatStyle(columns = 1:3, fontSize = '60%')
```

```
## Joining, by = "group"
```

Show 10 entries Search:

	group	genres_total	artists
1	1	indie rock (58) ;alternative rock (55) ;lo-fi (47) ;anti-folk (30) ;gbvfi (25) ;dream pop (24) ;post-hardcore (21) ;alternative pop (19) ;chamber pop (18) ;melancholia (18) ;slowcore (17) ;singer-songwriter (15) ;elephant 6 (14) ;art pop (12) ;art rock (12) ;post-rock (12) ;alternative country (11) ;noise pop (10) ;stomp and holler (10) ;emo (9) ;indie pop (9) ;math rock (9)	Sebadoh (49) ;Guided By Voices (48) ;The Olivia Tremor Control (42) ;Sparklehorse (42) ;Grandaddy (42) ;Pavement (40) ;The Microphones (39) ;Built To Spill (39) ;Yo La Tengo (38) ;Silver Jews (37) ;Stephen Malkmus (36) ;Stephen Malkmus & The Jicks (36) ;Superchunk (34) ;Archers Of Loaf (34) ;Dismemberment Plan (34) ;Heatmiser (33) ;Red House Painters (33) ;The Unicorns (32) ;Neutral Milk Hotel (31) ;Mercury Rev (30) ;of Montreal (30) ;Mount Eerie (30)
2	2	art pop (23) ;singer-songwriter (23) ;alternative rock (16) ;lilith (14) ;chamber psych (13) ;indie rock (11) ;indie pop (10) ;melancholia (10) ;permanent wave (10) ;electronica (9) ;alternative dance (8) ;alternative pop (8) ;chamber pop (8) ;dream pop (8) ;electropop (8) ;new rave (8) ;pop rock (8) ;alternative country (7) ;anti-folk (7) ;grunge (7) ;neo-synthpop (7)	PJ Harvey (38) ;Tindersticks (35) ;Mark Lanegan (29) ;Grinderman (28) ;Liz Phair (28) ;Joan As Police Woman (27) ;Anna Calvi (26) ;Bat For Lashes (26) ;Charlotte Gainsbourg (26) ;Nick Cave (25) ;Nick Cave & The Bad Seeds (24) ;Tori Amos (24) ;Morphine (24) ;Cat Power (23) ;Fiona Apple (22) ;John Parish (21) ;Hole (21) ;Björk (21) ;Warpaint (9) ;The Birthday Party (8)
3	3	britpop (66) ;pop rock (24) ;rock (24) ;modern rock (11) ;solo wave (11) ;madchester (10) ;alternative rock (9) ;new wave (5) ;permanent wave (5) ;welsh rock (5) ;chamber pop (4) ;dance rock (4) ;sheffield indie (4) ;big beat (3) ;candy pop (3) ;melancholia (3) ;new wave pop (3) ;british singer-songwriter (2) ;english indie rock (2) ;indie rock (2) ;piano rock (2) ;scottish indie (2) ;welsh indie (2) ;york indie (2)	Supergrass (53) ;The Seahorses (50) ;Ian Brown (50) ;Ocean Colour Scene (49) ;The Charlatans (49) ;Cast (48) ;Doves (46) ;Kula Shaker (45) ;The Bluetones (44) ;Richard Ashcroft (43) ;Shed Seven (42) ;Embrace (40) ;Manic Street Preachers (38) ;James (38) ;Elastica (37) ;The La's (36) ;The Coral (36) ;Sleeper (35) ;Pulp (33) ;Primal Scream (33)
4	4	alternative rock (51) ;post-punk (48) ;punk (33) ;art rock (22) ;new wave (22) ;experimental (20) ;uk post-punk (17) ;alternative pop (16) ;gbvfi (16) ;post-hardcore (14) ;punk blues (14) ;power pop (13) ;pub rock (11) ;rock (11) ;experimental rock (10) ;art punk (9) ;hardcore punk (9) ;protopunk (9) ;singer-songwriter (9) ;permanent wave (8)	Wire (39) ;Meat Puppets (37) ;The Breeders (36) ;Frank Black (35) ;Dinosaur Jr. (32) ;Magazine (32) ;The Fall (31) ;Minutemen (30) ;Gang Of Four (30) ;Wipers (29) ;Public Image Ltd, (29) ;Hüsker Dü (28) ;Television (28) ;Suicide (28) ;The Lemonheads (26) ;The Replacements (26) ;The Damned (24) ;Sonic Youth (23) ;Patti Smith (23) ;Fugazi (23) ;Violent Femmes (23)
5	5	dream pop (60) ;shoegaze (54) ;nu gaze (36) ;alternative rock (21) ;art pop (16) ;indie shoegaze (12) ;noise pop (12) ;dreamo (10) ;american shoegaze (9) ;art rock (9) ;dreamgaze (8) ;ethereal wave (8) ;alternative pop (7) ;britpop (6) ;melancholia (6) ;new wave (6) ;c86 (5) ;dark wave (5) ;indie rock (5) ;singer-songwriter (5)	Pale Saints (46) ;Drop Nineteens (45) ;Lush (43) ;Chapterhouse (42) ;Galaxie 500 (41) ;Swirlies (39) ;Ariel (38) ;Astrobrite (38) ;Ringo Deathstarr (37) ;Ride (35) ;LSD and the Search for God (33) ;Mojave 3 (33) ;Fleeting Joys (33) ;my bloody valentine (32) ;Slowdive (32) ;Hope Sandoval & The Warm Inventions (32) ;The Jesus and Mary Chain (32) ;Lovesliescrushing (32) ;Cranes (31) ;Cocteau Twins (30) ;Catherine Wheel (30) ;Swervedriver (30)
6	6	modern rock (41) ;indie rock (38) ;rock (35) ;new rave (33) ;alternative dance (27) ;alternative rock (18) ;garage rock (16) ;dance-punk (15) ;britpop (8) ;english indie rock (7) ;garage rock revival (6) ;modern alternative rock (6) ;british indie rock (5) ;electronica (5) ;indietronica (5) ;dream pop (4) ;modern blues rock (4) ;electroclash (3) ;neo-synthpop (3) ;scottish rock (3)	Klaxons (32) ;The Horrors (29) ;White Lies (29) ;Editors (27) ;The Bravery (26) ;Albert Hammond Jr (26) ;Franz Ferdinand (25) ;The Kills (24) ;The Rapture (24) ;The Last Shadow Puppets (24) ;Bloc Party (23) ;Julian Casablancas (23) ;Foals (22) ;The Maccabees (9) ;Interpol (8) ;The Cribs (8) ;Hard-FI (8) ;Dirty Pretty Things (7) ;The Libertines (7) ;The Rakes (7) ;We Are Scientists (7)
7	7	electronica (54) ;trip hop (54) ;nu jazz (30) ;downtempo (21) ;big beat (16) ;ninja (16) ;alternative dance (6) ;jazztronica (5) ;turntablism (5) ;electro jazz (4) ;art pop (3) ;alternative metal (2) ;chamber psych (2) ;dance pop (2) ;electropop (2) ;funk metal (2) ;solo wave (2) ;ambient pop (1) ;avant-garde metal (1) ;bleep techno (1) ;dark pop (1) ;devon indie (1) ;disco house (1) ;electro swing (1) ;etherpop (1) ;funk rock (1) ;hip hop (1) ;hip-hop experimental (1) ;icelandic electronic (1) ;icelandic pop (1) ;intelligent dance music (1) ;japanese old school hip hop (1) ;laboratorio (1) ;livetronica (1) ;neo-synthpop (1) ;new rave (1) ;nu skool breaks (1) ;supergroup (1) ;swedish electropop (1) ;swedish jazz (1) ;swedish singer-songwriter (1) ;techno (1) ;tzadik (1)	Lamb (36) ;Martina Topley-Bird (36) ;Tricky (34) ;Flunk (34) ;Morcheeba (33) ;Smoke City (32) ;Télépopmusik (32) ;Beth Gibbons (31) ;Sneaker Pimps (31) ;Hooverphonic (30) ;Sofa Surfers (30) ;Zero 7 (28) ;Massive Attack (28) ;UNKLE (27) ;Archive (24) ;Air (24) ;Moloko (24) ;Goldfrapp (23) ;Lovage (21) ;Portishead (13)
8	8	electronica (47) ;microhouse (26) ;intelligent dance music (20) ;uk bass (20) ;wonky (16) ;minimal techno (15) ;future garage (9) ;ambient (7) ;chillwave (7) ;float house (7) ;art pop (6) ;drill and bass (6) ;glitch (5) ;uk experimental electronic (5) ;deconstructed club (4) ;escape room (4) ;experimental pop (4) ;indie rock (4) ;ninja (4) ;alternative dance (3) ;chamber psych (3) ;deep house (3) ;downtempo (3) ;electra (3) ;fluxwork (3) ;folktronica (3) ;german techno (3) ;glitch hop (3) ;hamburg electronic (3) ;hauntology (3) ;new rave (3) ;outsider house (3) ;oxford indie (3) ;scottish electronic (3)	Thom Yorke (28) ;Burial (28) ;Four Tet (28) ;Mount Kimbie (26) ;Modeselektor (24) ;Floating Points (24) ;Jonny Greenwood (23) ;Apparat (23) ;Caribou (23) ;Philip Selway (22) ;James Holden (22) ;Atoms For Peace (21) ;Radiohead (21) ;Aphex Twin (21) ;Flying Lotus (21) ;Clark (8) ;Nosaj Thing (6) ;Actress (6) ;Teebs (6) ;Gold Panda (5) ;Daphni (5)
9	9	alternative rock (40) ;rock (32) ;grunge (30) ;alternative metal (29) ;pop rock (27) ;post-grunge (18) ;nu metal (17) ;funk metal (9) ;hard rock (8) ;rap rock (8) ;funk rock (5) ;permanent wave (5) ;blues rock (3) ;neo mellow (3) ;supergroup (3) ;australian alternative rock (1) ;australian psych (1) ;comic (1) ;glam rock (1) ;industrial metal (1) ;industrial rock (1) ;modern rock (1) ;punk (1) ;rap metal (1) ;spacegrunge (1)	Screaming Trees (37) ;Mad Season (37) ;Blind Melon (37) ;Local H (37) ;Jane's Addiction (36) ;Mother Love Bone (35) ;Temple Of The Dog (35) ;Days Of The New (35) ;Pomo for Pyros (33) ;Toadies (32) ;Mudhoney (32) ;Soundgarden (32) ;Stone Temple Pilots (30) ;Bush (30) ;Soul Asylum (29) ;Alice In Chains (28) ;Silverchair (27) ;Filter (26) ;Faith No More (24) ;The Presidents Of The United States Of America (23)
10	10	indie rock (40) ;modern rock (21) ;alternative rock (16) ;art pop (15) ;alternative dance (14) ;chillwave (14) ;stomp and holler (14) ;new rave (12) ;dream pop (11) ;noise pop (11) ;indie pop (8) ;experimental pop (7) ;indietronica (7) ;brooklyn indie (6) ;canadian indie (6) ;chamber pop (6) ;rock (6) ;baroque pop (5) ;neo-psychedelic (5) ;garage psych (4) ;noise rock (4) ;nu gaze (4)	Deerhunter (35) ;Grizzly Bear (30) ;Animal Collective (29) ;Broken Social Scene (29) ;Clap Your Hands Say Yeah (29) ;The Walkmen (26) ;TV On The Radio (26) ;Panda Bear (24) ;DIIV (22) ;Yeah Yeah Yeahs (22) ;Arcade Fire (21) ;Destroyer (11) ;Wolf Parade (10) ;Atlas Sound (7) ;The New Pornographers (7) ;Women (6) ;Menomena (6) ;LCD Soundsystem (5) ;Ariel Pink (5) ;Okkervil River (5)

Showing 1 to 10 of 12 entries Previous 1 2 Next

Most importantly, this shows that it's not always necessary to have access to the 'full' network in order to get results. In many cases, we will work with ego networks or sets of connected ego networks. Even so, it is possible to make interesting insights into the data regardless. The key takeaway is that you should take into account the perspective you're looking from.

15.2.0.4 Network metrics

Using the techniques from the previous lesson, we can make a table of network metrics:

```
node_info = level_3_related %>%
  distinct(id, name, genres) %>% rbind(level_3_related %>% select(id = seed_id,name =

g = level_3_related %>%
  #filter(id %in% radiohead_related_related$id) %>%
  distinct(id, seed_id) %>%
  as_tbl_graph(directed = TRUE) %>%
  left_join(node_info, by = 'name') %>%
  mutate(in_degree = centrality_degree(mode = 'in'))%>%
  mutate(out_degree = centrality_degree(mode = 'out'))%>%
  mutate(total_degree = centrality_degree(mode = 'all')) %>%
  mutate(between = centrality_betweenness()) %>%
  arrange(desc(total_degree))
```

```
## Warning in betweenness(graph = graph, v = V(graph), directed = directed, :
## 'nobigint' is deprecated since igraph 1.3 and will be removed in igraph 1.4
```

```
g %>% as_tibble()
```

```
## # A tibble: 1,285 x 7
##    name                   artist_name     genres in_de~1 out_d~2 total~3 between
##    <chr>                  <chr>           <list>   <dbl>   <dbl>   <dbl>   <dbl>
##  1 0sHeX8oQ6o7xic3wMf4NBU Supergrass      <chr>       20      33      53  21170.
##  2 3mbVe260Kgvs1P8YFcCyY7 The Seahorses   <chr>       20      30      50    604.
##  3 3s398TKZNahAURRacx7oIT Ian Brown       <chr>       20      30      50   1294.
##  4 5vIOGcdmx1eIkq3ZtuS12U Ocean Colour S~ <chr>       20      29      49    591.
##  5 5fScAXreYFnuqwOgBsJgSd The Charlatans  <chr>       20      29      49    690.
##  6 2wrhBKGC3DTNNNDRJPaxW6 Sebadoh         <chr>       20      29      49   8797.
##  7 0vBDEQ1aLZpe4zgn2fPH6Z Cast            <chr>       20      28      48    800.
##  8 4oV5EVJ0XFWsJKoOvdRPvl Guided By Voic~ <chr>       20      28      48   6225.
##  9 0LVrQUinPUBFvVD5pLqmWY Doves           <chr>       20      26      46  22570.
## 10 0WY2ddzQUF9eh16GiqrElA Pale Saints     <chr>       20      26      46   2769.
## # ... with 1,275 more rows, and abbreviated variable names 1: in_degree,
## #   2: out_degree, 3: total_degree
```

A few things to note:

The network is asymmetric: if artist A is in the top twenty of artist B, it doesn't necessarily follow that B is in the top twenty of A. We can test that by calculating the network's *reciprocity*:

```
g %>%
  filter(name %in% radiohead_related_related$id) %>%
  igraph::reciprocity()
```

```
## [1] 0.6211688
```

Just over 60% of links are reciprocated.

It's also transitive:

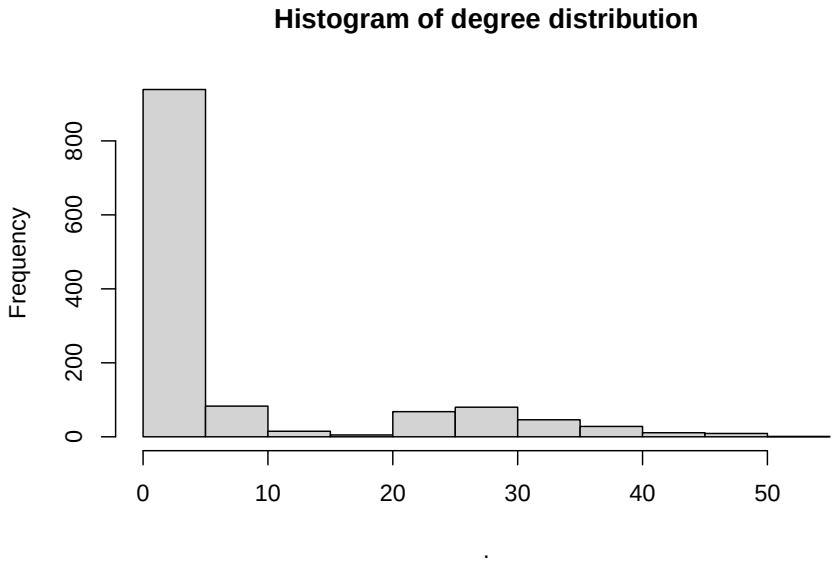
```
g %>%
  filter(name %in% radiohead_related_related$id) %>%
  igraph::transitivity('global')
```

```
## [1] 0.5569406
```

55% of possible triangles are complete.

The degree distribution is very particular to the sampling method. We have the 'full graph' for each of the 400 seed names (Radiohead's top twenty plus all their top twenties), but for the rest of the network (the third level removed from Radiohead), we *only* have their links to the 400 seed names. These nodes all have an incoming degree of 0, because we don't have any of their related artists, only the artists they are related to…

```
g %>% igraph::degree() %>% hist(main = "Histogram of degree distribution")
```



What does this tell us about the way music is organised?

Relatedness is transitive, meaning artists tend to form triangles: if both your friends like a band, it's likely you'll like them too. Spotify can exploit this fact to make better playlist recommendations!

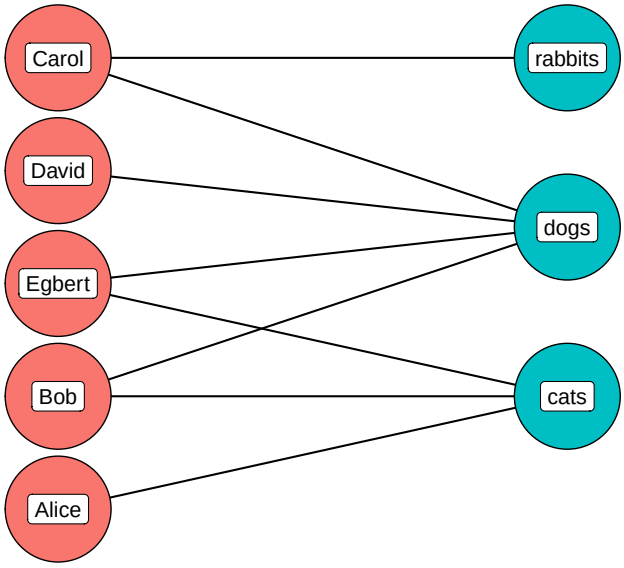
Relatedness is not necessarily reciprocal: there's a difference between A related to B and B related to A. Why might this be?

15.3 Bipartite networks

The work we have done so far has been on networks which are naturally what is known as *unimodel* or *unipartite*: person A sends a letter to person B. In a letter network there is only one type of node (a letter author or recipient), and one type of node (sends/receives a letter).

Many networks are not this straightforward, and have two, or more, types of nodes. For example, a network of twitter users connected to twitter groups, or a network of directors connected to companies. These networks are known as *bimodal* or *bipartite*, if there are two types of nodes, or *tripartite* when there's three, and so forth.

The following is a diagram of a bipartite network of Facebook users and group membership.



In this network, the first type (red, on the left) are people, and the second type (green, on the right) are Facebook groups. A line is drawn from one to the other if they are a member of that group. Carol is a member of the rabbits and

dog group, Carol, David, Bob, and Egbert are members of the dog group, and Egbert and Alice members of the cat group.

In digital humanities research, we often have access to bipartite network data, because almost any two sets of data points can be represented as a bipartite network. In some cases, the dataset can be derived rather than some pre-existing membership or category the data belongs to, as, for instance, in this study of the New Zealand parliament, which constructed a bipartite network of MPs to a set of speech topics, created using LDA topic modelling. In this model MPs are the first type, and topics the second, and these are used to construct a network of MPs based on their similarity across the topics they spoke about in Parliament.

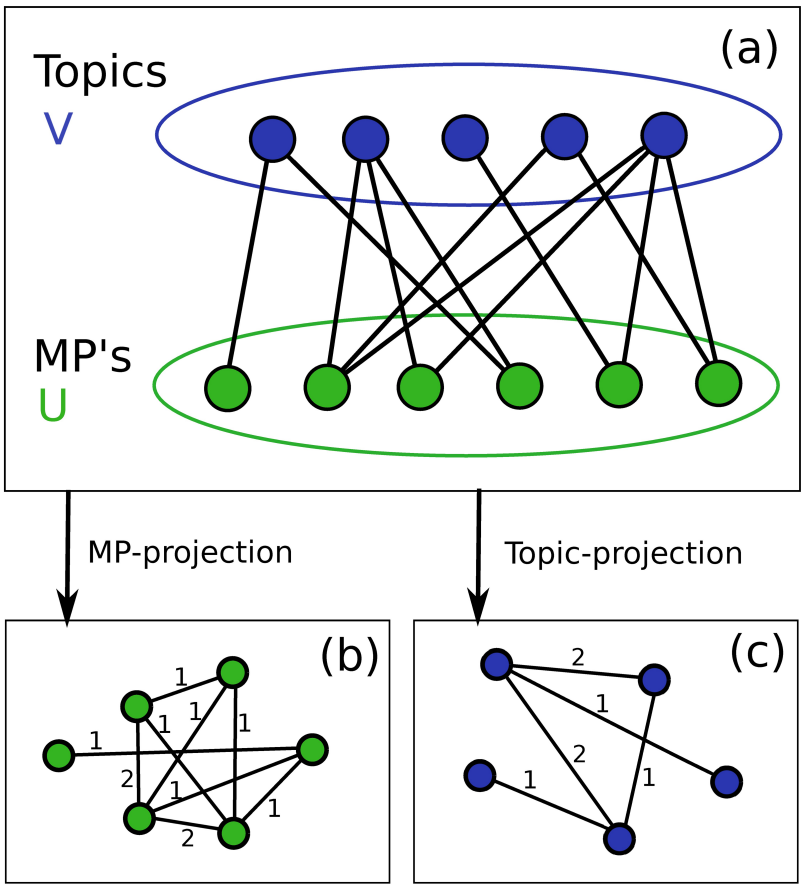
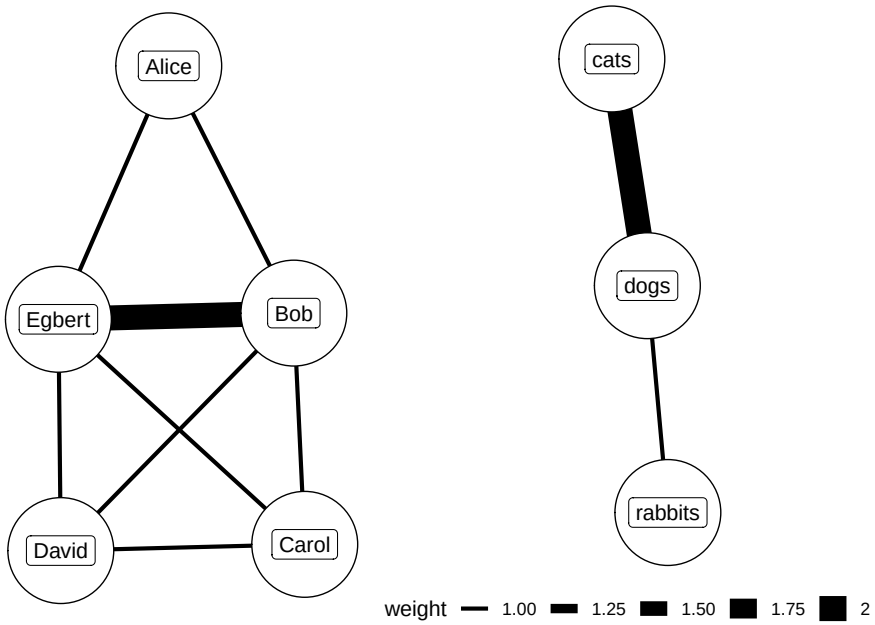


Figure 15.2: From Curran B, Higham K, Ortiz E, Vasques Filho D (2018) Look who's talking: Two-mode networks as representations of a topic model of New Zealand parliamentary speeches. PLoS ONE 13(6): e0199072. https://doi.org/10.1371/journal.pone.0199072

It's important, therefore, to understand the extent to which regular network

methods work or don't work with this structure. Standard network measurements (such as degree) are easy to calculate using these networks, but are not always meaningful. In the above example, the degree count for each node (its connections) is simply a count of its group membership. Unlike in a regular network, the measurement doesn't give any clues as to the most central member of the group. Similar problems exist for other metrics.

In many cases, then, we will need to do something to the network in order to get meaningful analysis from it. The most common thing to do is to *project* the network. This involves collapsing the network, and directly connecting one of the node types, based on their connections to the other. For example, the network above can be collapsed into two separate networks: a network of people connected by shared group membership, and a network of groups connected by shared members:



The network on the left displays a very common aspect of bipartite network projections: *cliques*, a cluster of nodes where each is connected to all the others.

In the network on the left, the edge becomes 'shares a Facebook group with', and on the right, 'has shared members'.

Which of these two networks do you think is more appropriate?

To a certain extent, that depends on the question. The more obvious answer would be to build a network of people, but if we were more interested in the 'ecosystem' of Facebook groups and how they interact, then perhaps the second network type would be of more use.

At this point, regular network metrics can be used. We might use degree, for example, to demonstrate that Alice is peripheral to this network.

However, it's important to be aware of what projecting the network does. Most importantly, there is a potential *loss of information*: in the new network, the edge only records that there is a shared group between two nodes, and the information on which groups specifically were shared is discarded.

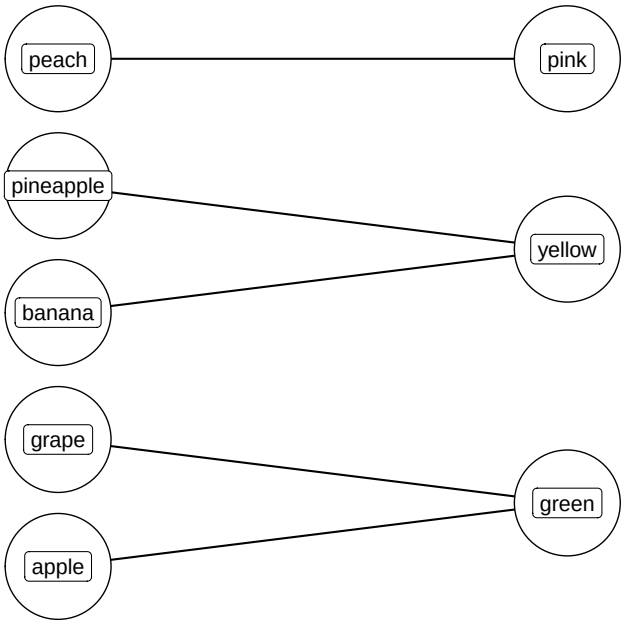
Some of this information can be kept through a weight value attached to each node. In the diagram on the left above, Egbert and Bob have a weight of two, because they share two groups (cats and dogs). This weight information can be incorporated into your network metrics.

Technically almost any data can be modelled as a bipartite network. However, is it always appropriate?

To give a slightly ridiculous example, imagine you had a dataset of fruit, and their corresponding colours.

fruit	color
apple	green
banana	yellow
peach	pink
pineapple	yellow
grape	green

There is nothing stopping you from turning this into a bipartite network of fruit connected to colours, and even projecting this to a network of fruit directly connected by shared colours. It is very easy to technically turn this into a network.



But is it meaningful? Perhaps not, unless there was a very clear reason for doing (biologists may be interested in this very question!).

15.4 Co-occurrence and co-authorship networks.

However, there are many cases where a bipartite network does actually make sense. Two very popular (and related) uses are **co-authorship networks** and **co-citation networks**. In the former, people are connected to the papers they wrote together, and in the latter, they are connected if they were cited in the same paper together.

Unlike fruit and colours, this data has some inherently networked-looking properties. The connections are likely to be clustered into different topics or academic communities, and if A and B both authored separate papers with C, they probably have a higher chance of also authoring a paper together.

Perhaps even more interestingly, this is a way of finding connections where we otherwise may have no data. We probably don't have any information on whether a large group of academic writers were in contact with each other (although maybe some of it can be found through Twitter data). A co-authorship network allows us to *infer* these connections through another dataset.

This is very often the case in humanities datasets, particularly historical, where we only have very limited information on who was in contact with whom, and

then, only if their letters or some other record of their contact survived. However, we may have more information on the companies they worked for, the groups they were part of, or the publications they worked on.

Modelling this data as a network may allow us to understand subject boundaries, highlight influential individuals, and look, for example, at questions of gender or racial bias in patterns of authorship and citation. This recent paper constructed a co-authorship network of digital humanities publications, and found that even through there were less women authors overall, they had important roles as bridges, linking otherwise disconnected areas. This network diagram from that paper shows the centrality of many of the green (female) authors:

15.4.1 A type of co-authorship: publisher networks

We can take a similar approach to the information found in historical books. While co-authorship itself in early modern publishing was rare, most books were *produced* by collaborations between sets of publishers and printers. These relationships are well suited to modelling as a network, and we could imagine they might display some of the network tendencies we've just discussed.

15.4.2 The dataset

The dataset we'll be working with the a dataset of metadata from the English Short Title Catalogue (known as the ESTC). This data lists a unique ID for each publisher, printer, bookseller and author listed on the title pages of books printed between 1500 and 1800. This information comes from the **imprint** of the book: the list of authors, printers, publishers and so forth found on the title page.

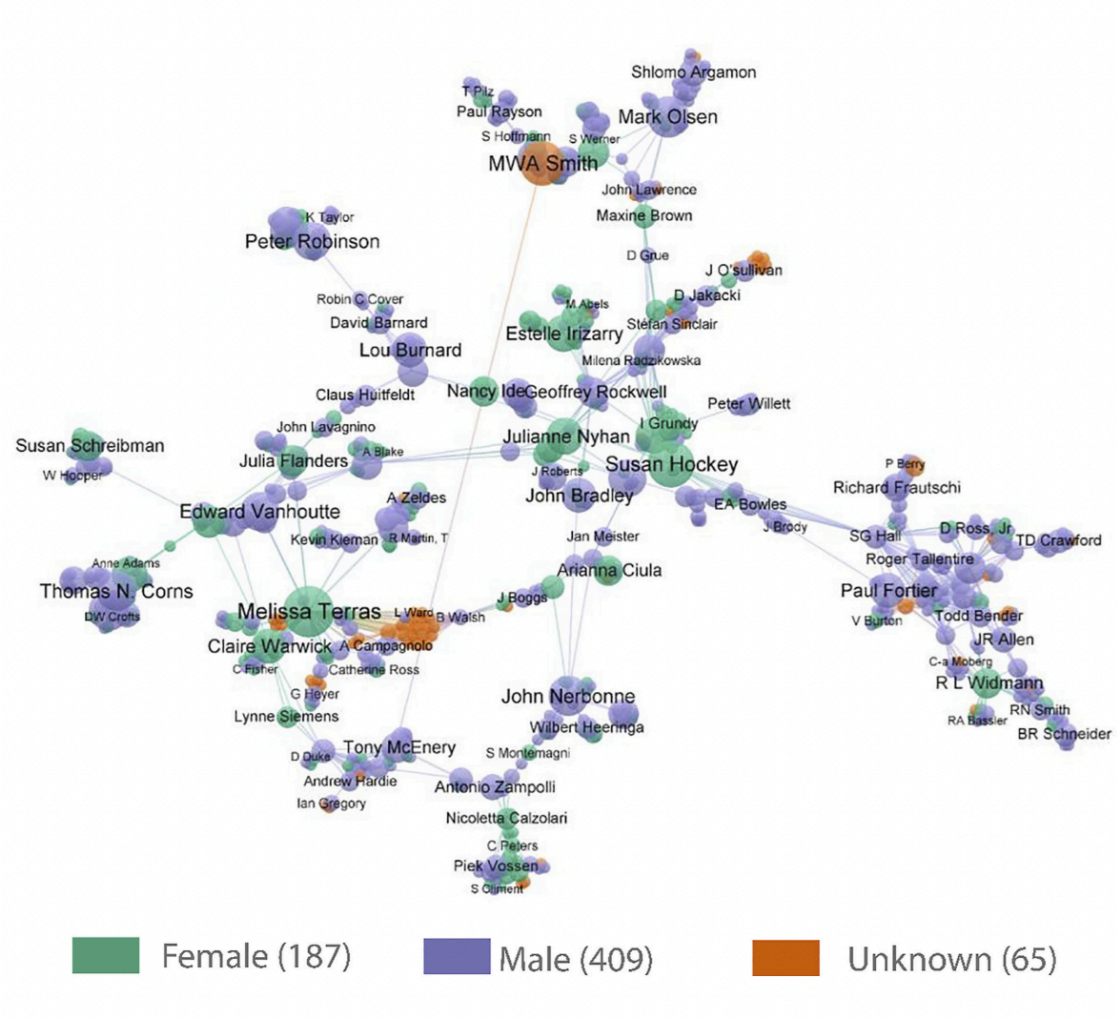


Figure 15.3: From Gao, J., Nyhan, J., Duke-Williams, O. and Mahony, S. (2022), "Gender influences in Digital Humanities co-authorship networks", Journal of Documentation, Vol. 78 No. 7, pp. 327-350. https://doi.org/10.1108/JD-11-2021-0221

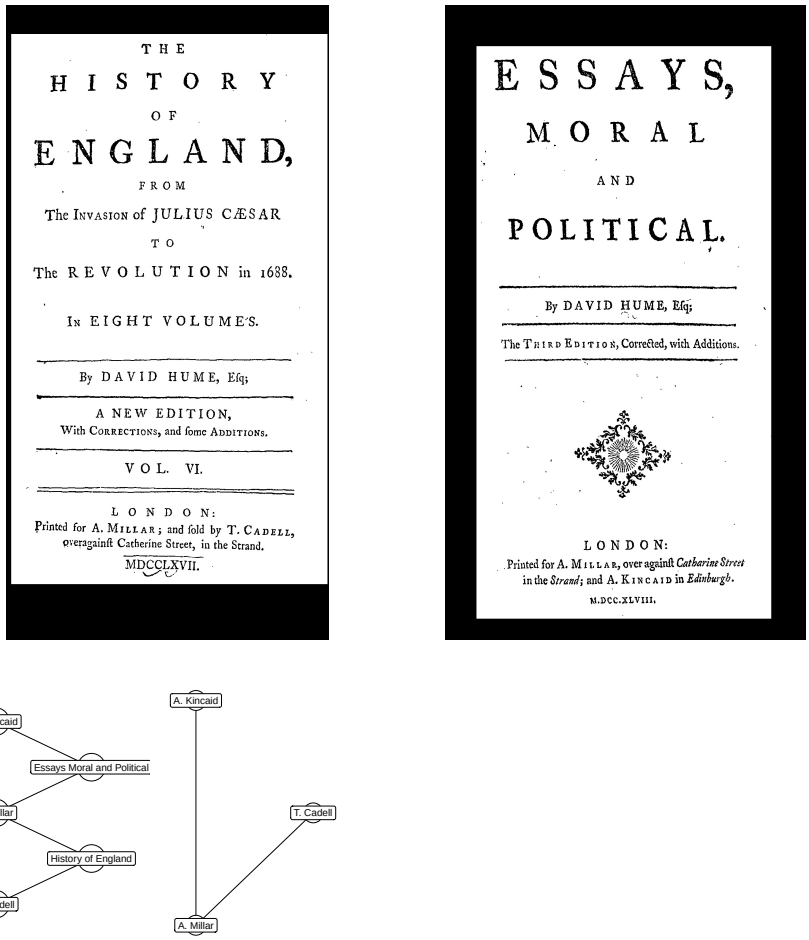


Figure 15.4: Bipartite network from book imprints

As you can see, the title page lists a few pieces of information: the book is 'printed for' A. Millar (who is the publisher), and 'sold by' Thomas Cadell, the bookseller. These pieces of information, showing a connection between Millar and Cadell, can form the basis of a bipartite network.

15.4.3 Method

In this class we'll take this raw data, filter it, and turn it into a bipartite network of publishers and printers connected to books. We'll then *project* the network, and directly connect the publishers and printers, based on their shared co-occurrences on books. This network can then be visualised and analysed.

First, load the data into R:

```
load(file = '../publisher_network/estc_actor_links')
load('../publisher_network/estc_core')
```

The data is organised like this: each row represents a book and a single actor linked to that book (meaning a publisher, printer, bookseller, or author). Each actor and book has a unique code. Further columns give information on the type of actor (some can have multiple, for example be the author and the publisher).

The book IDs and the actor IDs can be used as an *edge list*, and processed exactly the same as the previous lesson on regular one-mode networks. To do this, first filter to the appropriate types of actors, and then use `select()` to choose the `actor_id` and `estc_id` columns:

```
edge_list = estc_actor_links %>% left_join(estc_core %>% select(estc_id, publication_y
  filter(publication_year %in% 1750:1760) %>%
  filter(publication_place == 'Edinburgh') %>%
  filter(actor_role_publisher == TRUE) %>%
  select(estc_id, actor_id)
```

```
## Joining, by = "estc_id"
```

Use the same functions as before to turn this into a network object:

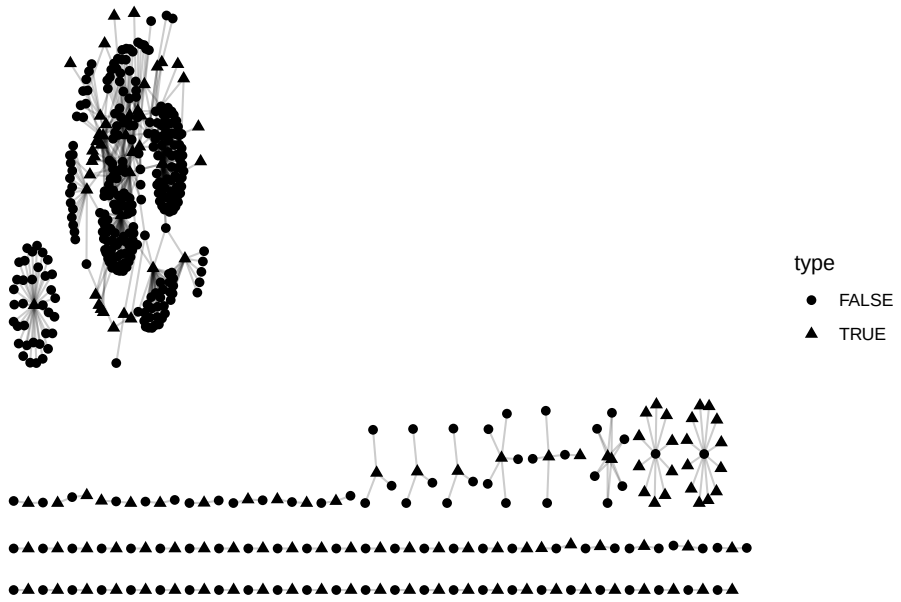
```
publisher_graph  = edge_list%>% as_tbl_graph()
```

The next steps are specific to bipartite networks. In order for R to 'know' that the network is bipartite, each node needs to have an associated type. We use an igraph function for this, called `bipartite_mapping()`. This assigns a TRUE or FALSE value to each node, depending on whether they are found in the first or second column of the data. This is saved as an attribute of the nodes using the following code:

```
V(publisher_graph)$type <- bipartite_mapping(publisher_graph)$type
```

At this point, we can already visualise the network, setting the shape of the node to the type:

```
publisher_graph %>%
  ggraph('stress') +
  geom_edge_link(alpha = .2) +
  geom_node_point(aes(shape = type),size =2) +
  theme_void()
```



It already looks like a network structure, with a number of disconnected components (individuals who never collaborate, or only in a small group), with a central connected 'component', consisting of publishers who often collaborate on books together.

However, we want to know more specifically about the structure of the publisher network. To do this, we project it.

Igraph has another function for this, `bipartite_projection()`. Use this function on the network:

```
proj = bipartite.projection(publisher_graph)
```

The result is a new object, a list, containing to further objects. These are the two network projections (publisher to publisher, and book to book). They can be accessed using `proj[[1]]` and `proj[[2]]`. We'll work with the second item, the publishers, but first, turn it into a tidygraph object so that we can work on it in the same way as the previous lessons.

```
proj[[2]] = proj[[2]] %>% as_tbl_graph()

proj[[2]]
```

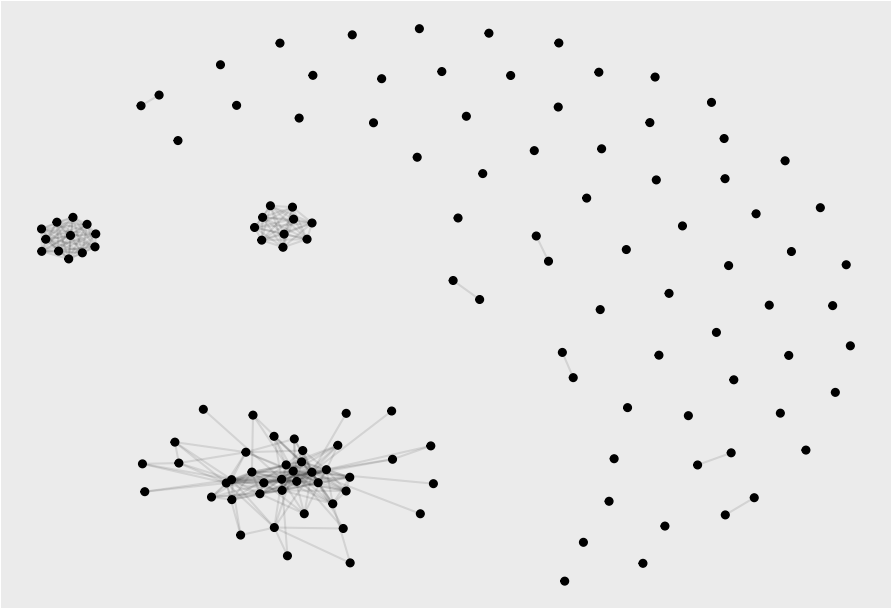
```
## # A tbl_graph: 133 nodes and 308 edges
## #
```

```
## # An undirected simple graph with 66 components
## #
## # Node Data: 133 x 1 (active)
##   name
##   <chr>
## 1 46725009
## 2 robertfleming_0
## 3 andrewstevensonwriter_0
## 4 willgordon_0
## 5 alexanderkincaid_1
## 6 johngray_0
## # ... with 127 more rows
## #
## # Edge Data: 308 x 3
##    from    to weight
##   <int> <int>  <dbl>
## 1     2     4      2
## 2     2     7      1
## 3     2    11      1
## # ... with 305 more rows
```

Now we have a new network, consisting of publisher nodes, and edges. Note, also, that the edges have a weight attached, representing the number of shared books they occur on.

The rest of the steps can be copied directly from the previous lesson on networks. Access the publisher-to-publisher network using `proj[[2]]`, and using the same method for visualising using ggraph, draw a network diagram:

```
proj[[2]] %>%
  mutate(degree = centrality_degree(mode = 'all', weights = weight)) %>%
  ggraph('fr') +
  geom_edge_link(alpha = .1) +
  geom_node_point()
```



Because we used a small sample of books, the network consists of a number of disconnected components. There is one large component, of publishers who work together in groups on several books, 2 small 'cliques' (groups of publishers all mentioned on one book, hence they are all connected together and not to anything else), and a large number of individual nodes with no connections (they were the only person listed on a single book), as well as a few pairs (two individuals listed together on a single book, and not to anyone else).

Next week, we'll build a network like this to tackle interesting humanities questions.

15.5 Conclusions

- A co-authorship network like this can get very dense very quickly, because there are many books, and only a limited number of individuals, so there will be many connections between them. It may be more meaningful to filter the data, for example using the edge weight column, to only consider 'stronger' relationships in the network (nodes which share several books together).

- The network also has a large number of 'isolates': nodes which are disconnected completely from the full network. These could also be removed, for visual clarity at least. To do this, you could filter to remove nodes with a total degree of one.

- This projected network is inherently *undirected*, because the edge 'shares a book title' doesn't have any direction associated with it.

15.6 Exercises:

a) Construct two spotify ego networks. Write up the clusters found in them. Compare their global network metrics. What does this tell you about the difference between them - for instance how wide or narrow their reach is?

Bibliography

Peter Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer Publishing Company, Incorporated, 2012. ISBN 3642311636.

M. De Domenico, A. Lima, P. Mougel, and M. Musolesi. The anatomy of a scientific rumor. *Scientific Reports*, 2980(3):5–23, 2013. doi: https://doi.org/10.1038/srep02980.

Alexandra Hill. *Lost Print in England: Entries in the Stationers' Company Register, 1557–1640*, pages 144–159. Brill, Leiden, The Netherlands, 2016. ISBN 9789004311824.

Mark J Hill, Ville Vaara, Tanja Säily, Leo Lahti, and Mikko Tolonen. Reconstructing intellectual networks: From the estc?s bibliographic metadata to historical material. In *Digital Humanities in the Nordic Countries*, 2019. URL https://kar.kent.ac.uk/90141/.

Leo Lahti, Jani Marjanen, Hege Roivainen, and Mikko Tolonen. Bibliographic data science and the history of the book (c. 1500–1800). *Cataloging & Classification Quarterly*, 57(1):5–23, 2019. doi: 10.1080/01639374.2018.1543747.

Benedek Rozemberczki, Carl Allen, and Rik Sarkar. Multi-Scale Attributed Node Embedding. *Journal of Complex Networks*, 9(2), 2021.

Gautam Kishore Shahi, Anne Dirkson, and Tim A. Majchrzak. An exploratory study of covid-19 misinformation on twitter. *Online Social Networks and Media*, 22:100104, 2021. ISSN 2468-6964. doi: https://doi.org/10.1016/j.osnem.2020.100104. URL https://www.sciencedirect.com/science/article/pii/S2468696420300458.

Richard Sher. *The Enlightenment and the Book : Scottish Authors and Their Publishers in Eighteenth-Century Britain, Ireland, and America*. The University of Chicago Press, 2006.