Hands-on Network Machine Learning with Scikit-Learn and Graspologic

Joshua Vogelstein, Alex Loftus, and Eric Bridgeford

Dec 03, 2021

CONTENTS

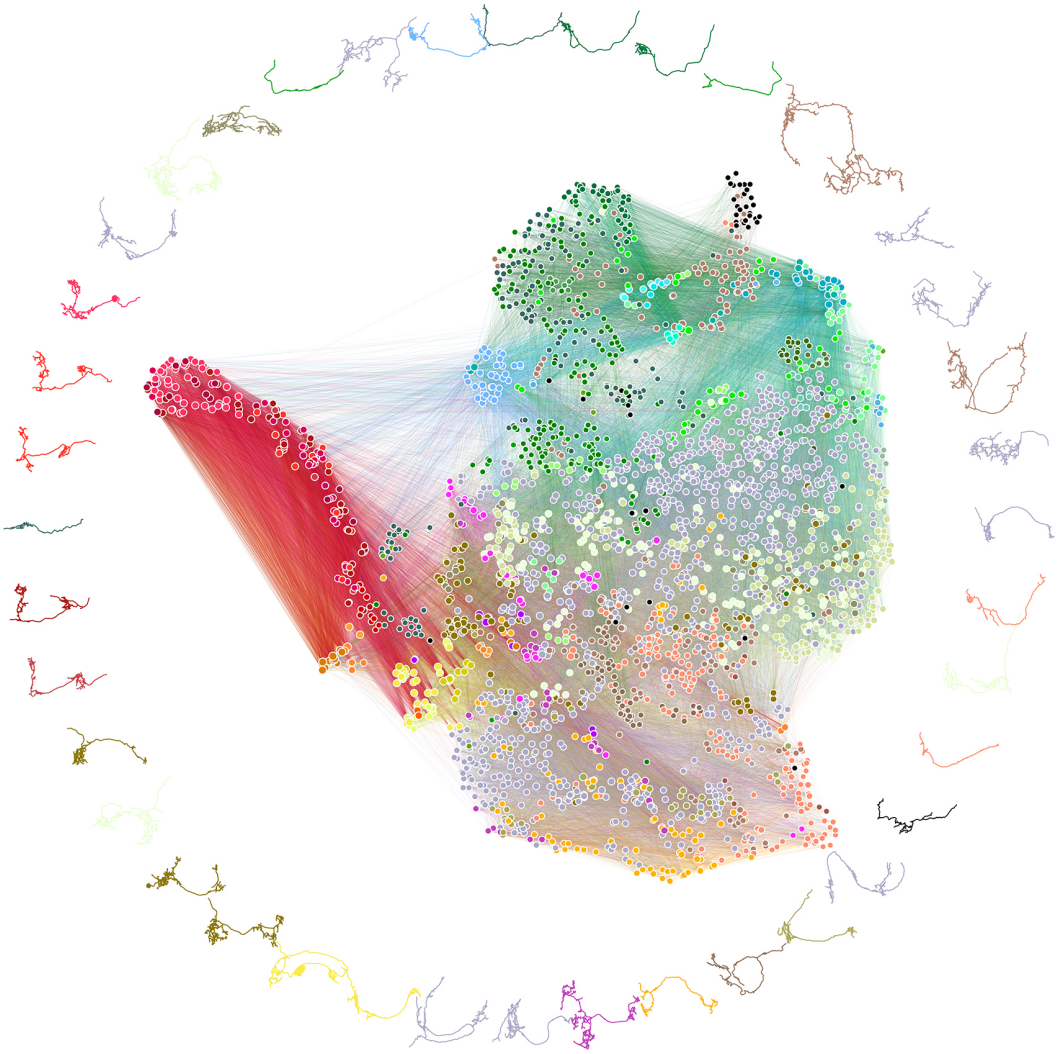


Fig. 1: 2D representation of a *Drosophila* larva brain connectome network. Credit to Ben Pedigo, PhD student at Johns Hopkins University.

Part I

Introduction

CHAPTER

ONE

PREFACE

1.1 Network Machine Learning and You

This book is about networks, and how you can use tools from machine learning to understand and explain them more deeply. Why is this an interesting thing to learn about, and why should you care?

Well, at some level, every aspect of reality seems to be made of interconnected parts. Atoms and molecules are connected to each other with chemical bonds. Your neurons connect to each other through synapses, and the different parts of your brain connect to each other through groups of neurons interacting with each other. At a larger level, you are interconnected with other humans through social networks, and our economy is a global, interconnected trade network. The Earth's food chain is an ecological network, and larger still, every object with mass in the universe is connected to every other object through a gravitational network.

So if you can understand networks, you can understand a little something about everything!

1.2 Network Machine Learning in Your Projects

So, naturally you are excited about network machine learning and you would love to join the party!

Perhaps you're a researcher and you want to expose shadowy financial networks and corporate fraud? Or create a network framework for measuring teamwork in healthcare? Maybe you're interested in evolutionary releationships between different animals, or maybe you want to model communities of neurons in the brain?

Or maybe you're a data scientist and your company has tons of data (user logs, financial data, production data, machine sensor data, hotline stats, HR reports, etc.), and more than likely you could view the data as a network and unearth some hidden gems of knowledge if you just knew where to look? For example:

- Explore purchasing networks and isolate the most active customers
- Explore patterns of collaboration in your company's network of employees
- Detect which transactions are likely to be fraudulent
- Isolate groups in your company which are overperforming or underperforming
- Model the transportation chain necessary to produce and disseminate your product
- And more

Whatever the reason, you have decided to learn about networks and implement their analysis in your projects. Great idea!

1.3 Objective and Approach

This book assumes you know next to nothing about how networks can be viewed as a statistical object. Its goal is to give you the concepts, the intuitions, and the tools you need to actually implement programs capable of learning from network data.

The book is intended to give you the best introduction you can possibly get to explore and exploit network data. You might be a graduate student, doing research on biochemical networks or trade networks in ancient Mesopotamia. Or you might be a professional interested in an introduction to the field of network data science, because you think it might be useful for your company. Whoever you are, we think you'll find a lot of things that are useful and interesting in this book!

We'll cover the fundamentals of network data science, focusing on developing intuition on networks as statistical objects, doing so while paired with relevant Python tutorials. By the end of this book, you will be able to utilize efficient and easy to use tools available for performing analyses on networks. You will also have a whole new range of statistical techniques in your toolbox, such as representations, theory, and algorithms for networks.

We'll spend this book learning about network algorithms by showing how they're implemented in production-ready Python frameworks:

- Numpy and Scipy are used for scientific programming. They give you access to array objects, which are the main way we'll represent networks computationally.
- Scikit-Learn is very easy to use, yet it implements many Machine Learning algorithms efficiently, so it makes a great entry point for downstream analysis of networks.
- Graspologic is an open-source Python package developed by Microsoft and the NeuroData lab at Johns Hopkins University which gives you utilities and algorithms for doing statistical analyses on network-valued data.

The book favors a hands-on approach, growing an intuitive understanding of networks through concrete working examples and a bit of theory. While you can read this book without picking up your laptop, we highly recommend you experiment with the code examples available online as Jupyter notebooks at http://docs.neurodata.io/graph-stats-book/index.html.

1.4 Prerequisites

We assume you have a basic knowledge of mathematics. Because network science uses a lot of linear algebra, requiring a bit of linear algebra knowledge is unfortunately unavoidable. (You should know what an eigenvalue is!)

If you care about what's under the hood mathematically, we have certain sections marked as "advanced material" - you should have a reasonable understanding of college-level math, such as calculus, linear algebra, probability, and statistics for these sections.

You should also probably have some background in programming - we'll mainly be using Python to build and explore our networks. If you don't have too much of a Python or math background, don't worry - we'll link some resources to give you a head start.

If you've never used Jupyter, don't worry about it. It is a great tool to have in your toolbox and it's easy to learn. We'll also link some resources for you if you are not familiar with Python's scientific libraries, like numpy, scipy, networkx, and scikit-learn.

1.5 Roadmap

This book is organized into three parts.

Part I, Foundations, gives you a brief overview of the kinds of things you'll be doing in this book, and shows you how to solve a network data science problem from start to finish. It covers the following topics:

- What a network is and where you can find networks in the wild
- All the reasons why you should care about studying networks
- Examples of ways you could apply network data science to your own projects
- An overview of the types of problems Network Machine Learning is good at dealing with
- The main challenges you'd encounter if you explored Network Learning more deeply
- Exploring a real network data science dataset, to get a broad understanding of what you might be able to learn.

Part II, Representations, is all about how we can represent networks statistically, and what we can do with those representations. It covers the following topics:

- Ways you can represent individual networks
- Ways you can represent groups of networks
- The various useful properties different types of networks have
- Types of network representations and why they're useful
- How to represent networks as a bunch of points in space
- How to represent multiple networks
- How to represent networks when you have extra information about your nodes

Part III, Applications, is about using the representations from Part II to explore and exploit your networks. It covers the following topics:

- Figuring out if communities in your networks are different from each other
- Selecting a reasonable model to represent your data
- Finding nodes, edges, or communities in your networks that are interesting
- Finding time points which are anomalies in a network which is evolving over time
- What to do when you have new data after you've already trained a network model
- How hypothesis testing works on networks
- Figuring out which nodes are the most similar in a pair of networks

1.6 Conventions Used In This Book

1.7 Using Code Examples

1.8 About the Authors

Dr. Joshua Vogelstein is an Assistant Professor in the Department of Biomedical Engineering at Johns Hopkins University, with joint appointments in Applied Mathematics and Statistics, Computer Science, Electrical and Computer

Engineering, Neuroscience, and Biostatistics. His research focuses on the statistics of networks in brain science (connectomes). His lab and collaborators have developed the leading computational algorithms and libraries to perform statistical analysis on networks.

Alex Loftus is a master's student at Johns Hopkins University in the Department of Biomedical Engineering, with an undergraduate degree in neuroscience. He has worked on implementing network spectral embedding and clustering algorithms in Python, and helped develop an MRI pipeline to produce brain networks from diffusion MRI data.

Eric Bridgeford is a PhD student in the Department of Biostatistics at Johns Hopkins University. Eric's background includes Computer Science and Biomedical Engineering, and he is an avid contributor of packages to CRAN and PyPi for nonparametric hypothesis testing. Eric studies general approaches for statistical inference in network data, with applications to problems with network estimation in MRI connectomics data, including replicability and batch effects.

Dr. Carey E. Priebe is Professor of Applied Mathematics and Statistics, and a founding member of the Center for Imaging Science (CIS) and the Mathematical Institute for Data Science (MINDS) at Johns Hopkins University. He is a leading researcher in theoretical, methodological, and applied statistics / data science; much of his recent work focuses on spectral network analysis and subsequent statistical inference. Professor Priebe is Senior Member of the IEEE, Elected Member of the International Statistical Institute, Fellow of the Institute of Mathematical Statistics, and Fellow of the American Statistical Association.

Dr. Christopher M. White is Managing Director, Microsoft Research Special Projects. He leads mission-oriented research and software development teams focusing on high risk problems. Prior to joining Microsoft, he was a Fellow at Harvard for network statistics and machine learning. Chris's work has been featured in media outlets including Popular Science, CBS's 60 Minutes, CNN, the Wall Street Journal, Rolling Stone Magazine, TEDx, and Google's Solve for X. Chris was profiled in a cover feature for the Sept/Oct 2016 issue of Popular Science.

Weiwei Yang is a Principal Development Manager at Microsoft Research. Her interests are in resource efficient alt-SGD ML methods inspired by biological learning. The applied research group she leads aims to democratize AI by addressing issues of sustainability, robustness, scalability, and efficiency in ML. Her group has applied ML to address social issues such as countering human trafficking and to energy grid stabilizations.

1.9 Acknowledgements

First of all, big thanks to everybody who has been reading the book as we write and giving feedback. So far, this list includes Dax Pryce, Ross Lawrence, Geoff Loftus, Alexandra McCoy, Olivia Taylor, and Peter Brown.

1.10 Finished Sections

(lots more in progress…)

1. Preface: *Preface*
2. Why Use Statistical Models: *Why Use Statistical Models?*
3. Single-Network Models:
4. Multi-Network Representation Learning: *Multiple-Network Representation Learning*
5. Joint Representation Learning: *Joint Representation Learning*

CHAPTER

TWO

TERMINOLOGY AND MATH REFRESHER

In this section, we outline some background terminology which will come up repeatedly throughout the book. This section attempts to standardize some background material that we think is useful going in. It is important to realize that many of the concepts discussed below are only crucial for understanding the advanced, starred sections. If you aren't familiar with some (or any!) of the below concepts, we don't think this would detract from your understanding of the broader content.

2.1 Vectors, Matrices, and Numerical Spaces

Throughout this book, we will need some level of familiarity with numerical spaces, and the grammar that we use to describe them. Taking the time to understand this notation will better help you understand many of the concepts in the rest of the book.

2.1.1 Numerical Spaces

Numerical spaces are everywhere. If you have taken any calculus or algebra courses, you are likely familiar with the natural numbers - these are just your basic one, two, three, and so on. This constitutes the most basic numerical space and is denoted by the symbol $\mathbb{N}$. Formally, the natural numbers describes the set:

$$\mathbb{N} = \{1, 2, 3, ...\}$$

and continues infinitely (notice that neither negative numbers nor numbers with decimal points appear in $\mathbb{N}$). On a similar note, we will frequently resort to short hand to describe subsets of the natural numbers. We will use the symbol $\in$ (read, "in") to denote that one quantity is found within a particular set. For example, since 5 is a natural number, we would say that $5 \in \mathbb{N}$, which can be thought of as "5 is in the set of natural numbers". To describe a subset of the first 5 natural numbers, we would use the notation $[5]$, which denotes the set:

$$[5] = \{1, 2, 3, 4, 5\}$$

In the more general case where we have some variable n where $n \in \mathbb{N}$ (again, n is some arbitrary natural number), then:

$$[n] = \{1, 2, ..., n\}$$

The next most basic numerical space is known as the integers, which is just the natural numbers combined with the negative numbers and zero. Specifically:

$$\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$$

From $-\infty$ up to $+\infty$.

There are many more numerical spaces, but in this book we'll focus on one in particular: real numbers, denoted $\mathbb{R}$. The real numbers can be thought of as all the numbers that can be represented by a finite or infinite number of decimal places in between (and including) the integers. We won't go into too many details; if you want more details on the real numbers,

a good place to start would be coursework in **real analysis**. Particularly, the real numbers include any natural number, and integer, any decimal, or any irrational number (such as π or $\sqrt{2}$). The main thing that is interesting about the real numbers that we will *indirectly* use throughout the book is that if we have any two real numbers x and y (remember, this would be written $x, y \in \mathbb{R}$), then the products, ratios, or sums of them are also real numbers:

$$x \cdot y, \frac{x}{y}, x + y \in \mathbb{R}$$

Throughout the book, we will build upon some of these numerical spaces and introduce several new ones along the way that are interesing for network machine learning. We will do this by attempting to relate them back to the basic numerical spaces we have introduced here.

2.1.2 One-Dimensional Quantities

We will frequently see the term "dimensional" come up in this book, and we will attempt to give some insight into what this means here. If we were to say that $x \in \mathbb{R}$, we know from the above description that this means that x is a real number, and is therefore "in" the set of real numbers. A one-dimensional quantity is a quantity which is described by a single element from one numerical space. In this instance, x is described by one real number, and is therefore one-dimensional. We will use a lowercase letter (for instance, x, a, b, α, β; the letters may be Roman or Greek) to denote that a quantity is one-dimensional.

2.1.3 Vectors

Building off the concept of one-dimensional variables, what if we had some variable that existed in two dimensions? For instance, consider the following:

$$\vec{x} = \begin{bmatrix} 1.5 \\ 2 \end{bmatrix}$$

As we can see here, $\vec{x}$ is now described by two real numbers (namely, 1.5 and 2). This means that $\vec{x}$ is now a two-dimensional quantity, since we have two separate values needed to describe $\vec{x}$. In this case, $\vec{x}$ no longer is "in" the real numbers, it is instead in the two-dimensional real vectors, or $\mathbb{R}^2$. Here, $\vec{x}$ is called a **vector**, and each of its dimensions are defined using the notation $x_1 = 1.5$ and $x_2 = 2$. The subscript x_j just means the j^{th} element of $\vec{x}$, which is numbered by counting downwards from the first row ($j = 1$) to however many rows $\vec{x}$ has in total. Since $\vec{x}$ is two-dimensional, we would say that $j \in [2]$, which means j can be either 1 or 2. In general, we will assume that all vectors are **column vectors** unless otherwise stated, which means that $\vec{x}$ will be assumed to be vertically aligned. This will not make much of a conceptual difference, but it will play a role when we define operations between vectors and matrices later on. On the other hand, a **row vector** will typically be denoted by using the **transpose** symbol, which we will learn about later on in the section on operators. Unlike a column vector, a row vector is aligned horizontally. For example, a row vector with entries identical to $\vec{x}$ will be denoted:

$$\vec{x}^\top = \begin{bmatrix} 1.5 & 2 \end{bmatrix}$$

In the general case, for any set $\mathcal{S}$, we would say that $\vec{s} \in \mathcal{S}^d$ if (think through this notation!) for any $j \in [d]$, $s_j \in \mathcal{S}$. The key aspects are that the symbol for the vector will be a lower case letter (in this example, s) like the one-dimensional quantity, but will add the $\vec{}$ symbol to denote that it is a vector with more than one dimension. The quantity d that you see in the superscript is referred to as the dimensionality. In this example, we would say that $\vec{s}$ is a d-dimensional $\mathcal{S}$-vector.

2.1.4 Matrices

Matrices come up a lot in network science because we often represent networks as matrices: the adjacency matrix, for instance, is a way to represent a network in terms of its edge connections. Because networks can be represented as matrices, we'll sometimes just talk about matrices directly.

We will see a variety of different types of matrices throughout this book, so let's start with a simple example. Consider the following marix:

$$X = \begin{bmatrix} 1.5 & 1.7 \\ 2 & 1.8 \end{bmatrix}$$

Here, we can see that X is described by four real numbers, with a particular arrangement. This time, we say that X is an element of the set of all possible 2×2 (2 rows, and 2 columns) matrices with real entries. In symbols, we would describe this as $\mathbb{R}^{2\times 2}$, where $\mathbb{R}$ says that the elements of the matrix are real numbers, and 2×2 means that the matrix has two rows and two columns. We can describe the entries of a matrix using indexing, very similar to what we did for vectors. In matrices, the rows and columns matter. In this case, the rows go from left to right horizontally, and the columns go from top to bottom vertically. The rows will be numbered from the top of the matrix to the bottom, and the columns will be numbered from the left-most column to the right-most column. For instance, the first row of the matrix X is the row-vector $\begin{bmatrix} 1.5 & 1.7 \end{bmatrix}$, and the second column of the matrix X is the column-vector $\begin{bmatrix} 1.7 \\ 1.8 \end{bmatrix}$. This subscripts x_{ij} means the entry of the matrix X in the i^{th} row aand the j^{th} column. In this instance, we would describe that $x_{11} = 1.5$, $x_{12} = 1.7$, $x_{21} = 2$, and $x_{22} = 1.8$.

In the general case, for a set $\mathcal{S}$, we would say that $S \in \mathcal{S}^{r\times c}$ if (think this through!) for any $i \in [r]$ and any $j \in [c]$, $s_{ij} \in \mathcal{S}$. Like before, the key aspects are that the symbol for a matrix will be a capital letter (in this example, S) to denote that it is a matrix, and its entries s_{ij} will be denoted using a lowercase letter. The quantity r is known as the row count and the quantity c is known as the column count of the matrix S. In this example, we would say that S is a $\mathcal{S}$-matrix with r rows and c columns.

Another thing we will see arise periodically is that vectors can be denoted as matrices with a single column. For example, in our example above in the vector section, we might equivalently write that $\vec{s} \in \mathcal{S}^{d\times 1}$. The "1" for the columns just denotes that $\vec{s}$ is a column vector with d rows in total. This will be useful when we define functions for matrices, and use the same notation for functions on vectors.

2.2 Useful Functions

Throughout the book, we will deal with many types of functions which take mathematical objects (potentially multiple) that exist in one numerical space and produce a mathematical object (potentially in a different) numerical space. You are probably familiar with several of these, such as the addition or multiplication operators on one-dimensional quantities. We will touch on some of the more fancy ones that we will see arise throughout the book.

The **sum**, denoted by a fancy capital epsilon $\sum$, denotes that we are summing a bunch of items which can be easily indexed. For instance, consider if we have a vector $\vec{x} \in \mathbb{R}^d$, so $\vec{x}$ is a d-dimensional vector. If we wanted to take the sum of all of the elements of $\vec{x}$, we would write:

$$\sum_{i=1}^{d} x_i = x_1 + x_2 + ... + x_d$$

The *summand* of the sum, the x_is next to the $\sum$ symbol, are the terms that will be summed up. Further, note that the $\sum$ symbol also indicates the indices of $\vec{x}$ that will be summed. Note that on the bottom, we see that the sum says from $i = 1$ and above it says d. This means that we sum all the elements of x_i starting from below at 1 and going up until d. We could say the exact same thing using our shorthand for this set, which we described in the section on natural numbers,

$[d]$:

$$\sum_{i \in [d]} x_i = \sum_{i=1}^{d} x_i = x_1 + x_2 + ... + x_d$$

We could similarly define **any** indexing set, such as $\mathcal{I} = \{1, 3\}$, and write:

$$\sum_{i \in \mathcal{I}} x_i = x_1 + x_3$$

The key is that the notation above or below the summand just tells us which elements we are applying the sum over. For instance, if $\vec{x}$ was a 3-dimensional vector:

$$\vec{x} = \begin{bmatrix} 1.7 \\ 1.8 \\ 2 \end{bmatrix}$$

We would have that:

$$\sum_{i=1}^{3} x_i = 5.5$$

if we were to use $\mathcal{I} = \{1, 3\}$, then:

$$\sum_{i \in \mathcal{I}} x_i = 3.7$$

the **product**, denoted by a capital pi $\prod$, behaves extremely similarly to the sum, except insted of applying sums, it applies multiplication. For instance, if we instead wanted to multiply all the elements of $\vec{x}$, we would write:

$$\prod_{i=1}^{d} x_i = x_1 \times x_2 \times ... \times x_d$$

Where $\times$ is just multiplication like you are probably used to. Again, we have the exact same indexing conventions, where:

$$\prod_{i \in [d]} x_i = \prod_{i=1}^{d} x_i = x_1 \times x_2 \times ... \times x_d$$

We can again just use indexing sets, too:

$$\prod_{i \in \mathcal{I}} x_i = x_1 \times x_3$$

With $\vec{x}$ defined as above in the sum example, we would have that:

$$\prod_{i=1}^{3} x_i = 6.12$$

if we were to use $\mathcal{I} = \{1, 3\}$, then:

$$\prod_{i \in \mathcal{I}} x_i = 3.4$$

The **Euclidean inner product**, or the *inner product* we will refer to in our book, is obtained by multiplying two vectors element-wise, and summing the result. Suppose we have two vectors $\vec{x}$ and $\vec{y}$, which are each d-dimensional real vectors (both x and y must have the same number of elements). The inner product is the quantity:

$$\langle \vec{x}, \vec{y} \rangle = \sum_{i=1}^{d} x_i y_i$$

as we will see in a second, in matrix notation, this is exactly equivalent to writing:

$$\langle \vec{x}, \vec{y} \rangle = \vec{x}^T \vec{y}$$

Matrix multiplication, denoted by a circle $\cdot$ (or in most cases, just two matrices side by side, with no separation), is an operation which takes a matrix which has r rows and c columns and another matrix which has c rows and l columns, and produces a matrix with r rows and l columns. Suppose we have a matrix $A \in \mathbb{R}^{r \times c}$, and $B \in \mathbb{R}^{c \times l}$. Here, r, c, and l could be *any* natural numbers. A matrix multiplication produces a matrix $D \in \mathbb{R}^{r \times l}$, where:

$$d_{ij} = \sum_{k=1}^{c} a_{ik} b_{kj}$$

What does this mean intuitively? Well, let's think about it. Let's imagine that the *rows* of A are indexed from 1 to r, like this:

$$A = \begin{bmatrix} \vec{a}_1^T \\ \vec{a}_2^T \\ \vdots \\ \vec{a}_r^T \end{bmatrix}$$

Note that the vectors $\vec{a}_i$ are transposed when oriented in the matrix A, because they are each c-dimensional vectors (and by convention in our book, all vectors will be *column* vecors. So to comprise the rows of A, they must be "flipped"). Similarly, let's imagine that the columns of B are indexed from 1 to l, like this:

$$B = \begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \dots & \vec{b}_l \end{bmatrix}$$

So what is the marix D? Note that each entry, $d_{ij} = \langle \vec{a}_i, \vec{b}_j \rangle = \vec{a}_i^T \vec{b}_j$. So the matrix D is the matrix whose entries are the *inner products of the rows of A with the columns of B*. In a diagram, D is like this:

$$D = \begin{bmatrix} \vec{a}_1^T \vec{b}_1 & \dots & \vec{a}_1^T \vec{b}_l \\ \vdots & \ddots & \vdots \\ \vec{a}_r^T \vec{b}_1 & \dots & \vec{a}_r^T \vec{b}_l \end{bmatrix}$$

As a matter of notation, we might often have the case where we want to discuss or interpret a single element which is a product of two matrices. For instance, suppose we care about the entry (i, j) of AB. We might also describe the resulting quantity d_{ij} using the notation $(AB)_{ij}$. The reason we adopt this notation is that we want to emphasize that the matrix multiplication operation is performed first (it is in *parentheses*), and then we look at the (i, j) entry of the resulting matrix.

The **Euclidean distance** is the most common distance between vectors we will see in this book. The Euclidean distance effectively tells us how far apart two points in d-dimensional space are. Given $\vec{x}, \vec{y} \in \mathbb{R}^d$ ($\vec{x}$ and $\vec{y}$ are d-dimensional real vectors), the Euclidean distance is the quantity:

$$\delta(\vec{x}, \vec{y}) = \langle \vec{x} - \vec{y}, \vec{x} - \vec{y} \rangle = \sum_{i=1}^{d} (x_i - y_i)^2$$

In particular, if we check the distance between a vector and the origin (the **zero-vector**, denoted 0_d, which is a d-dimensional vector where all entries are 0), we end up with a very useful quantity, called the squared Euclidean norm. We will use a special notation for the Euclidean norm, which is:

$$||\vec{x}||_2^2 = \delta(\vec{x}, 0_d) = \sum_{i=1}^{d} x_i^2$$

The subscript $_2$ just means that this is the "2"-norm, which is a concept outside of the scope of this book. The superscript ^{2} means that this is the squared Euclidean norm. Therefore, the Euclidean norm itself is:

$$||\vec{x}||_2 = \sqrt{\delta(\vec{x}, 0_d)} = \sqrt{\sum_{i=1}^{d} x_i^2}$$

What does this mean interpretation wise? The "square" operation basically means, if there are dimensions of $\vec{x}$ that are big, the norm will end up being big. If the dimensions of $\vec{x}$ are small, they will not contribute very much to the norm.

Based on the equation we saw above for the Euclidean distance, we could also understand the Euclidean distance to be the squared Euclidean norm of the vector which is the difference between $\vec{x}$ and $\vec{y}$. Using this convention:

$$\delta(\vec{x}, \vec{y}) = ||\vec{x} - \vec{y}||_2^2$$

In this sense, we can see that the Euclidean distance and the Euclidean norms are attributing a concept of "length" and "how far" a vector is from another (whether that is the origin or an arbitrary real vector). Next, we will see a related concept for matrices. The **squared Frobenius norm** is the quantity, given a matrix $A \in \mathbb{R}^{r \times c}$:

$$||A||_F^2 = \sum_{i=1}^{r} \sum_{i=1}^{c} a_{ij}^2$$

Note that this is very similar to the squared Euclidean norm of a vector, except it is applied to both the rows *and* the columns of A. Again, we have a similar interpretation to the Euclidean norm. If an entry of A is big, it will contribute much to the Frobenius norm due to the squared a_{ij} term. If an entry is smaller, it will not contribute as much. The Frobenius norm itself is just the square root of this:

$$||A||_F = \sqrt{\sum_{i=1}^{r} \sum_{i=1}^{c} a_{ij}^2}$$

2.3 Probability

Throughout this book, we will be very concerned with probabilities and probability distributions. For this reason, we will introduce some basic notation that we will be concerned with. In probability analyses, we are concerned with describing things that occur in the real world with some level of uncertainty. We capture this uncertainty using probability, which in essence, describes how likely (or unlikely) a particular outcome is compared to all of the possible outcomes that could be realized. In general, we will call the most basic objects which occur with some uncertainty **random variables**, which is a variable whose values that we get to see in the real world (the *realizations* of the random variable) depend on some random phenomenon. We will denote a random variable using a similar notation to a one-dimensional variable, with the exception that we will *bold face* the variable to make clear that it is random. For instance, for a one-dimensional random variable, we will use notation like $\mathbf{x}$.

Like before, we can also have random vectors and random matrices. Like for the random variable, we will denote these with bold faces too. A random vector will be denoted using a bold faced variable with the vector symbol; for example, $\vec{\mathbf{x}}$. Likewise, a random matrix will be denoted using a bold faced upper case letter; for example, $\mathbf{X}$. Similar to how we indexed vectors and matrices, the index positions of random vectors and random matrices are random variables, too. That is, $\vec{\mathbf{x}}$ is a d-dimensional random vector whose entries are the random variables $\mathbf{x}_i$ for all i from 1 to d:

$$\vec{\mathbf{x}} = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_d \end{bmatrix}$$

And $\mathbf{X}$ is a $(r \times c)$ random matrix whose entries are the random varaiables $\mathbf{x}_{ij}$ for all i from 1 to r and j from 1 to c:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_{11} & \cdots & \mathbf{x}_{1c} \\ \vdots & \ddots & \vdots \\ \mathbf{x}_{r1} & \cdots & \mathbf{x}_{rc} \end{bmatrix}$$

A probability distribution, denoted by $\mathbb{P}$, is a function which gives the probability of a particular value being attained by a random quantity. To state this another way, the probability distribution is concerned with fixing probabilities to realizations of random quantities. to make this a little more concrete, we will give an example with the simplest possible probability distribution, the Bernoulli distribution, denoted $Bernoulli(p)$. For the sake of this example, we will say

that $\mathbf{x}$ is a random variable which is $Bernoulli(p)$ distributed, which we denote by $\mathbf{x} \sim Bernoulli(p)$. The Bernoulli distribution describes that the probability of the random variable $\mathbf{x}$ taking a realization of 1 is p, whereas the probability of the random variable $\mathbf{x}$ taking a realization of 0 is $1 - p$. Using the probability distribution, we would say that:

$$\mathbb{P}(\mathbf{x} = 0) = 1 - p$$
$$\mathbb{P}(\mathbf{x} = 1) = p$$

2.4 Advanced Probability*

the probability distribution for a random vector or a random matrix is described very similarly. The caveat is that with a random vector/matrix, we affix a probability of *every element* of the random vector/matrix equaling the realized vector/matrix. For instance, if $\vec{\mathbf{x}}$ is a random vector taking realizations which are d-dimensional vectors, and $\vec{x}$ is one such d-dimensional vector, then:

$$\mathbb{P}(\vec{\mathbf{x}} = \vec{x}) = \mathbb{P}(\mathbf{x}_1 = x_1, ..., \mathbf{x}_d = x_d)$$

and likewise, if $\mathbf{X}$ is a random matrix taking realizations which are $r \times c$ matrices, and X is one such $r \times c$ matrix, then:

$$\begin{aligned}\mathbb{P}(\mathbf{X} = X) &= \mathbb{P}(\mathbf{x}_{11} = x_{11}, ..., \mathbf{x}_{rc} = x_{rc}) \\ &= \mathbb{P}(\mathbf{x}_{ij} = x_{ij} \text{ for any } i \text{ and } j)\end{aligned}$$

A probability concept we will see arise frequently in the advanced sections of the book is one called independence. A pair of random variables are independent if for any x which is a possible realization of $\mathbf{x}$ and y is a possible realization of $\mathbf{y}$, then:

$$\mathbb{P}(\mathbf{x} = x, \mathbf{y} = y) = \mathbb{P}(\mathbf{x} = x)\mathbb{P}(\mathbf{y} = y)$$

A related concept that will be very important in our study of random matrices is the idea of mutual independence. If we have a set of n random variables $\mathbf{x}_i$ for all $i = 1, ..., n$, this set of random variables is said to be mutually independent if for any x_1 which is a possible realization of $\mathbf{x}_1$, any x_2 which is a possible realization of $\mathbf{x}_2$, and so on up to $\mathbf{x}_n$, then:

$$\mathbb{P}(\mathbf{x}_1 = x_1, ..., \mathbf{x}_n = x_n) = \prod_{i=1}^{n} \mathbb{P}(\mathbf{x}_i = x_i)$$

The ways in which this is useful will become more obvious through some of the advanced material of later chapters.

Another important concept we will see arise in some of the advanced material is the idea of conditional distributions. Given x which is a possible realization of $\mathbf{x}$ and y is a possible realization of $\mathbf{y}$, then the conditional distribution of $\mathbf{x}$ on $\mathbf{y}$ is the quantity:

$$\mathbb{P}(\mathbf{x} = x|\mathbf{y} = y) = \frac{\mathbb{P}(\mathbf{x} = x, \mathbf{y} = y)}{\mathbb{P}(\mathbf{y} = y)}$$

While outside the scope of this book, it can be shown that this is a proper probability distribution function, but we mainly are concerned with the fact that this is simply a useful notation for an intuitive idea. What this allows us to capture is the idea of attributing a probability for a random variable $\mathbf{x}$ obtaining the value x, given that we already know that $\mathbf{y}$ obtains the value y. A related concept, Baye's Rule, uses a simple consequence of this theorem. Note that we could flip the probability statement above, and would obtain that:

$$\mathbb{P}(\mathbf{y} = y|\mathbf{x} = x) = \frac{\mathbb{P}(\mathbf{x} = x, \mathbf{y} = y)}{\mathbb{P}(\mathbf{x} = x)}$$

a simple rearrangement of terms by multiplying both sides by $\mathbb{P}(\mathbf{x} = x)$ gives us that:

$$\mathbb{P}(\mathbf{x}, \mathbf{y}) = \mathbb{P}(\mathbf{y} = y|\mathbf{x} = x)\mathbb{P}(\mathbf{x} = x)$$

Substituting this in to our first definition for a conditional distribution of $\mathbf{x}$ on $\mathbf{y}$ gives:

$$\mathbb{P}(\mathbf{x} = x|\mathbf{y} = y) = \frac{\mathbb{P}(\mathbf{y} = y|\mathbf{x} = x)\mathbb{P}(\mathbf{x} = x)}{\mathbb{P}(\mathbf{y} = y)}$$

which is Baye's Rule.

Part II

Foundations

CHAPTER

THREE

THE NETWORK MACHINE LEARNING LANDSCAPE

3.1 What Is A Network?

I would say start with a task: I have a bunch of edges. how can I find the most similar nodes to a given node, ranked? if these aren't connected, should they be? did we just link predict? (actual question I feel like link prediction is handwavy af)

3.1.1 links for inspiration

- ez intro on graps for ML https://towardsdatascience.com/graph-theory-and-deep-learning-know-hows-6556b0e9891b

3.2 Why Study Networks?

- motivation for network analysis: https://arxiv.org/pdf/0912.5410.pdf

3.3 Examples of applications

3.4 Types of Networks

3.5 Types of Network Learning Problems

3.6 Main Challenges of Network Learning

3.7 Exercises

CHAPTER

FOUR

END-TO-END BIOLOGY NETWORK MACHINE LEARNING PROJECT

CHAPTER

FIVE

END-TO-END BUSINESS NETWORK MACHINE LEARNING PROJECT

5.1 Look at the Big Picture

5.2 Get the Data

5.3 Discover and Visualize the Data to Gain Insights

5.4 Prepare the Data for Network Algorithms

Part III

Representations

CHAPTER

SIX

PROPERTIES OF NETWORKS AS A STATISTICAL OBJECT

6.1 Matrix Representations Of Networks

When we work with networks, we need a way to represent them mathematically and in our code. A network itself lives in network space, which is just the set of all possible networks. Network space is kind of abstract and inconvenient if we want to use traditional mathematics, so we'd generally like to represent networks with groups of numbers to make everything more concrete.

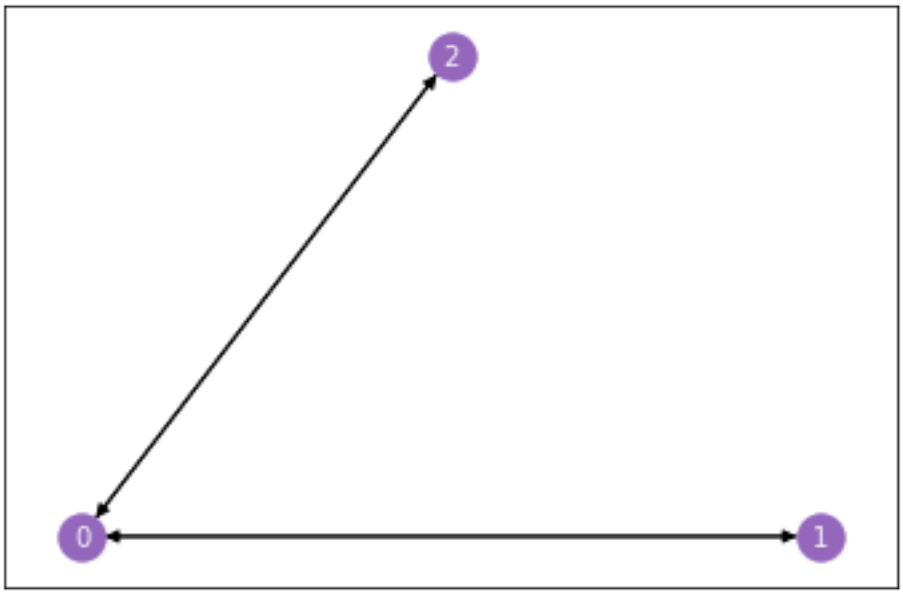
More specifically, we would often like to represent networks with *matrices*. In addition to being computationally convenient, using matrices to represent networks lets us bring in a surprising amount of tools from linear algebra and statistics. Programmatically, using matrices also lets us use common Python tools for array manipulation like numpy.

The most common matrix representation of a network is called the Adjacency Matrix, and we'll learn about that first.

6.1.1 The Adjacency Matrix

The beating heart of matrix representations for networks throughout this book is the adjacency matrix. The idea is pretty straightforward: Let's say you have a network with n nodes. You give each node an index – usually some value between 0 and n – and then you create an $n \times n$ matrix. If there is an edge between node i and node j, you fill the $(i, j)_{th}$ value of the matrix with an entry, usually 1 if your network has unweighted edges. In the case of undirected networks, you end up with a symmetric matrix with full of 1's and 0's, which completely represents the topology of your network.

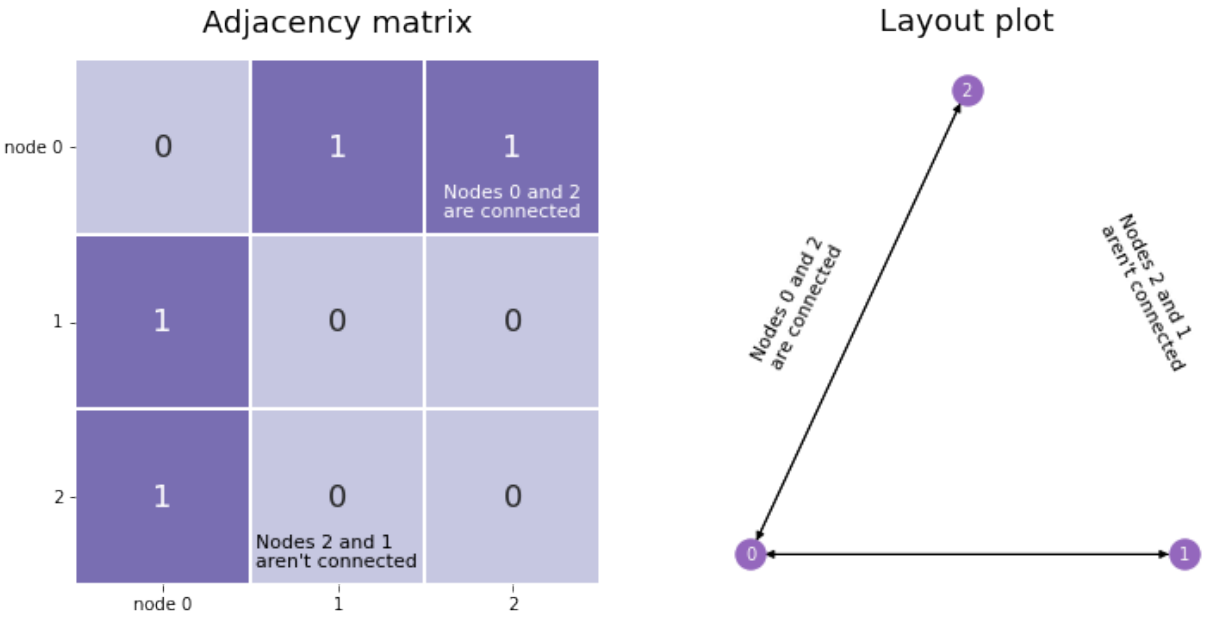
Let's see this in action. We'll make a network with only three nodes, since that's small and easy to understand, and then we'll show what it looks like as an adjacency matrix.



Our network has three nodes, labeled 1, 2, and 3. Each of these three nodes is either connected or not connected to each of the two other nodes. We'll make a square matrix A, with 3 rows and 3 columns, so that each node has its own row and column associated to it.

So, let's fill out the matrix. We start with the first row, which corresponds to the first node, and move along the columns. If there is an edge between the first node and the node whose index matches the current column, put a 1 in the current location. If the two nodes aren't connected, add a 0. When you're done with the first row, move on to the second. Keep going until the whole matrix is filled with 0's and 1's.

The end result looks like the matrix below. Since the second and third nodes aren't connected, there is a 0 in locations $A_{2,1}$ and $A_{1,2}$. There are also zeroes along the diagonals, since nodes don't have edges with themselves.

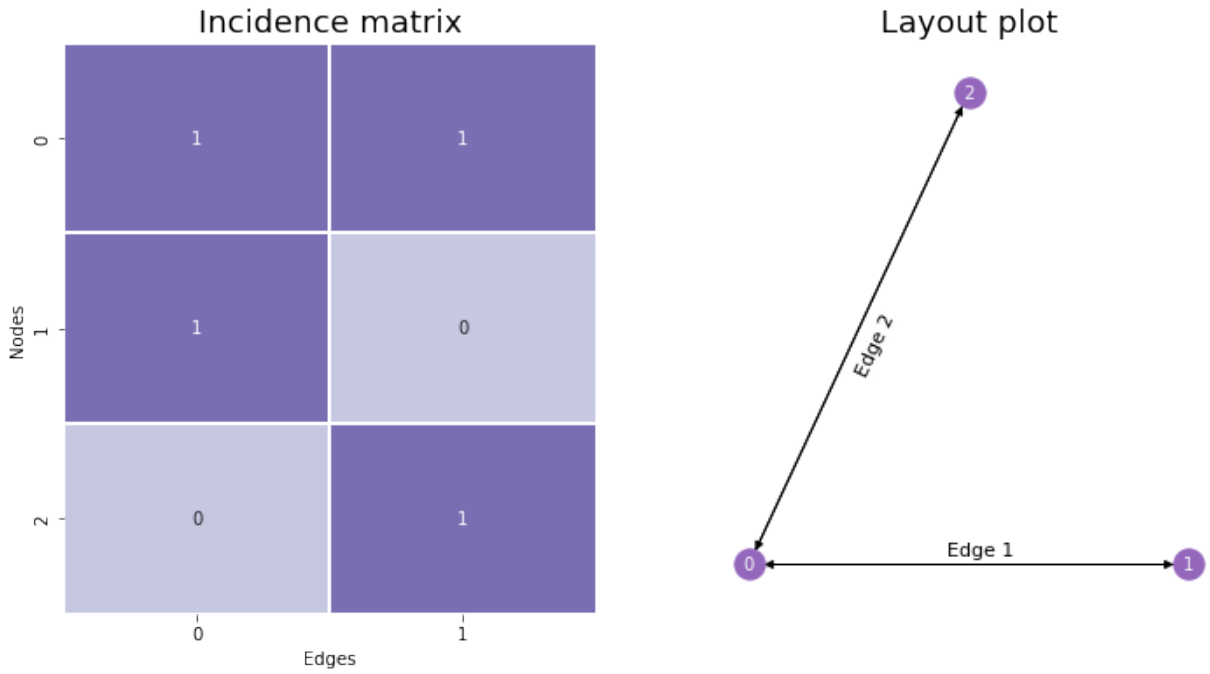


Although the adjacency matrix is straightforward and easy to understand, it isn't the only way to represent networks.

6.1.2 The Incidence Matrix

Instead of having values in a symmetric matrix represent possible edges, like with the Adjacency Matrix, we could have rows represent nodes and columns represent edges. This is called the *Incidence Matrix*, and it's useful to know about – although it won't appear too much in this book. If there are n nodes and m edges, you make an $n \times m$ matrix. Then, to determine whether a node is a member of a given edge, you'd go to that node's row and the edge's column. If the entry is nonzero (1 if the network is unweighted), then the node is a member of that edge, and if there's a 0, the node is not a member of that edge.

You can see the incidence matrix for our network below. Notice that with incidence plots, edges are (generally arbitrarily) assigned indices as well as nodes.



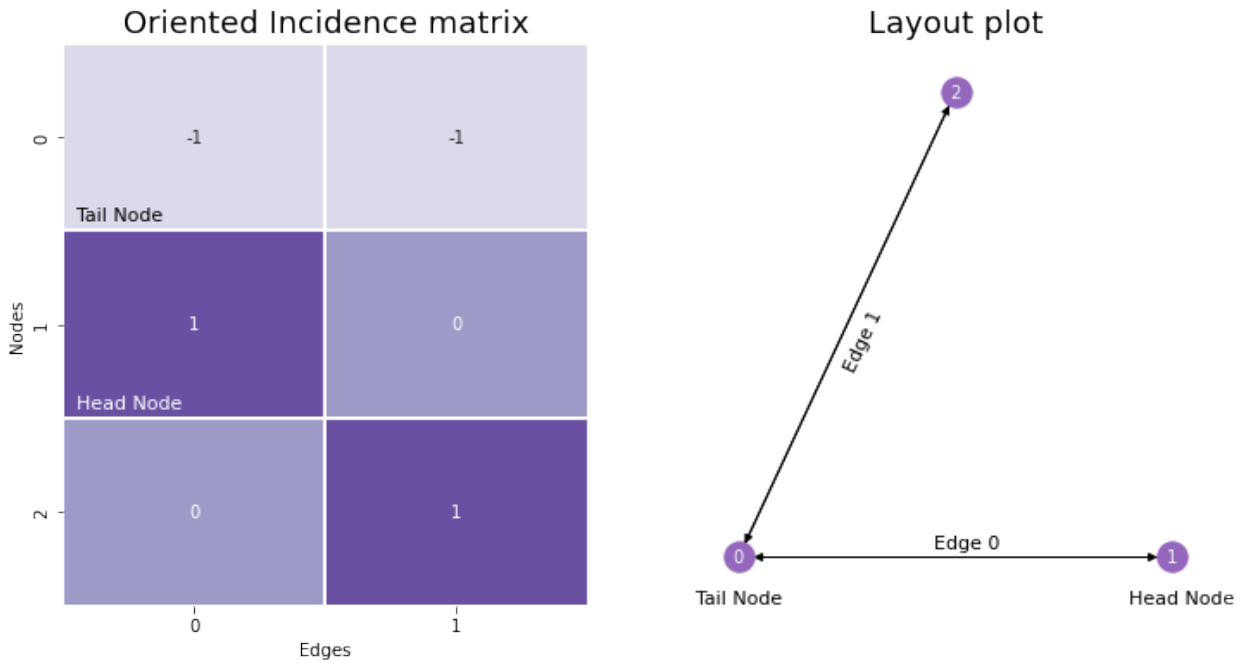
When networks are large, incidence matrices tend to be extremely sparse – meaning, their values are mostly 0's. This is because each column must have exactly two nonzero values along its rows: one value for the first node its edge is connected to, and another for the second. Because of this, incidence matrices are usually represented in Python computationally as scipy's *sparse matrices* rather than as numpy arrays, since this data type is much better-suited for matrices which contain mostly zeroes.

You can also add orientation to incidence matrices, even in undirected networks, which we'll discuss next.

6.1.3 The Oriented Incidence Matrix

The oriented incidence matrix is extremely similar to the normal incidence matrix, except that you assign a direction or orientation to each edge: you define one of its nodes as being the head node, and the other as being the tail. For undirected networks, you can assign directionality arbitrarily. Then, for the column in the incidence matrix corresponding to a given edge, the tail node has a value of -1, and the head node has a value of 0. Nodes who aren't a member of a particular edge are still assigned values of 0.

```
Text(0.9, -0.05, 'Head Node')
```



Although we won't use incidence matrices, oriented or otherwise, in this book too much, we introduced them because there's a deep connection between incidence matrices, adjacency matrices, and a matrix representation that we haven't introduced yet called the Laplacian. Before we can explore that connection, we'll discuss one more representation: the degree matrix.

6.1.4 The Degree Matrix

The degree matrix isn't a full representation of our network, because you wouldn't be able to reconstruct an entire network from a degree matrix.

6.1.5 The Laplacian Matrix

The Symmetric Laplacian

The Random-Walk Laplacian

6.2 Representations of Networks

6.3 Properties of Networks

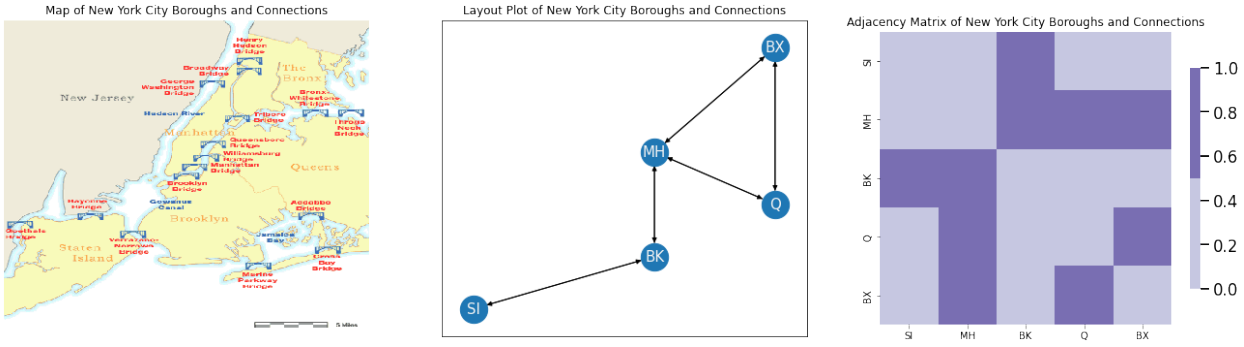
6.3.1 Descriptive Properties of Networks

Remember that a network topology, a collection of nodes $\mathcal{V}$, edges $\mathcal{E}$, can be represented as an $n \times n$ adjacency matrix, where n is the total number of nodes. The adjacency matrix looks like this:

$$A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix},$$

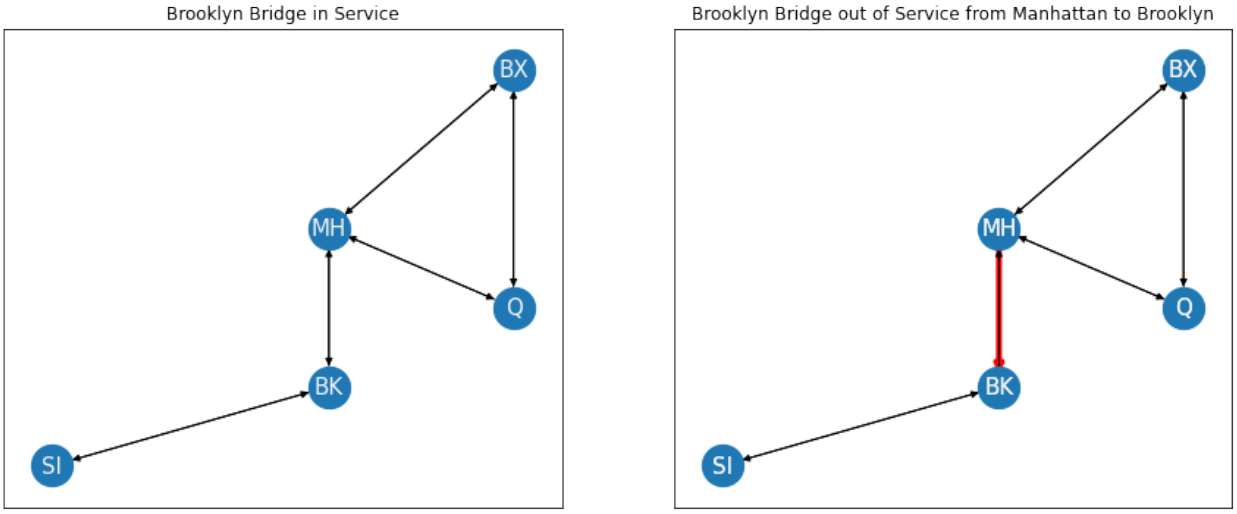
Let's say we have a network representing the five boroughs of New York (Staten Island SI, Brooklyn BK, Queens Q, the Bronx BX, and Manhattan MH). The nodes in our network are the five boroughs. The edges (i, j) of our network exist if one can travel from borough i to borough j along a bridge.

Below, we will look at a map of New York City, with the bridges connecting the different boroughs. In the middle, we look at this map as a network layout plot. The arrows indicate the direction of travel. On the right, we look at this map as an adjacency matrix:

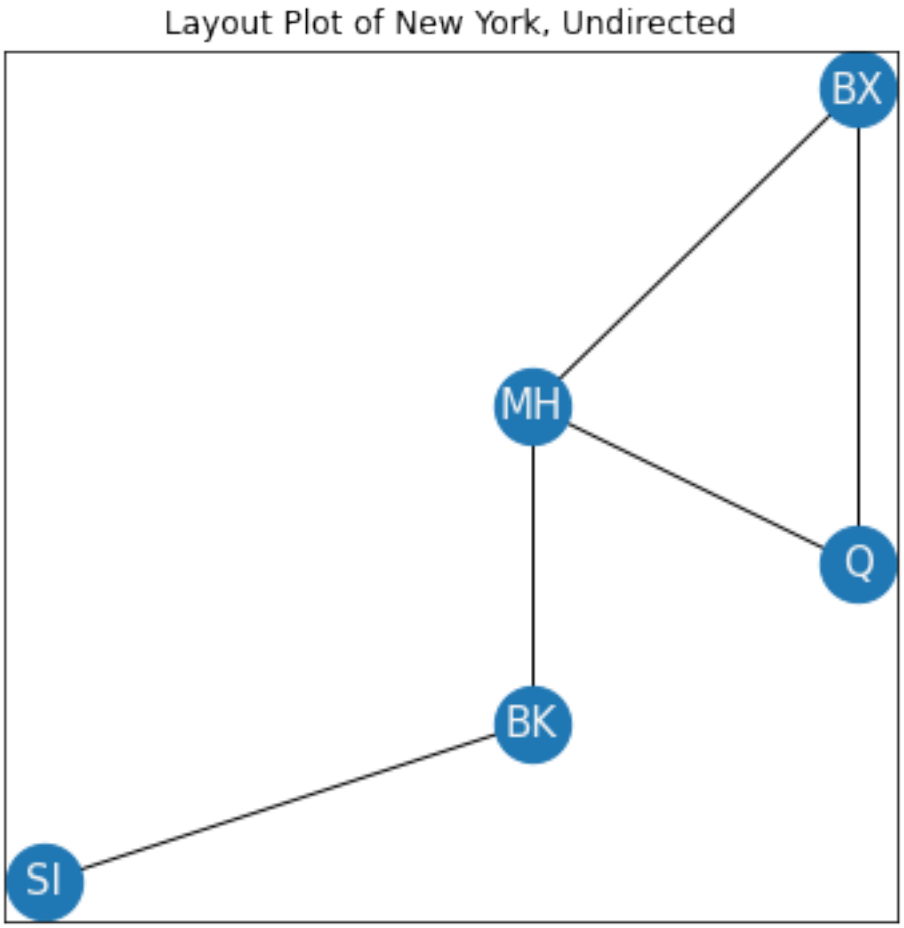


The edges of undirected networks are bi-directional

When you decide to travel from borough i to borough j, you care about whether you can *actually drive* on that bridge! In a similar way, the concept of directedness describes whether we need to worry about one-way bridges and bridge closures. If there are one-way bridges in our network, then a bridge from borough i to borough j doesn't *necessarily* imply that a bridge from borough j to borough i exists (just ask New York drivers). If, for instance, the Brooklyn bridge was closed from Manhattan to Brooklyn, our network might change like this. Note that the red arrow going from Manhattan (MH) to Brooklyn (BK) is no longer present:



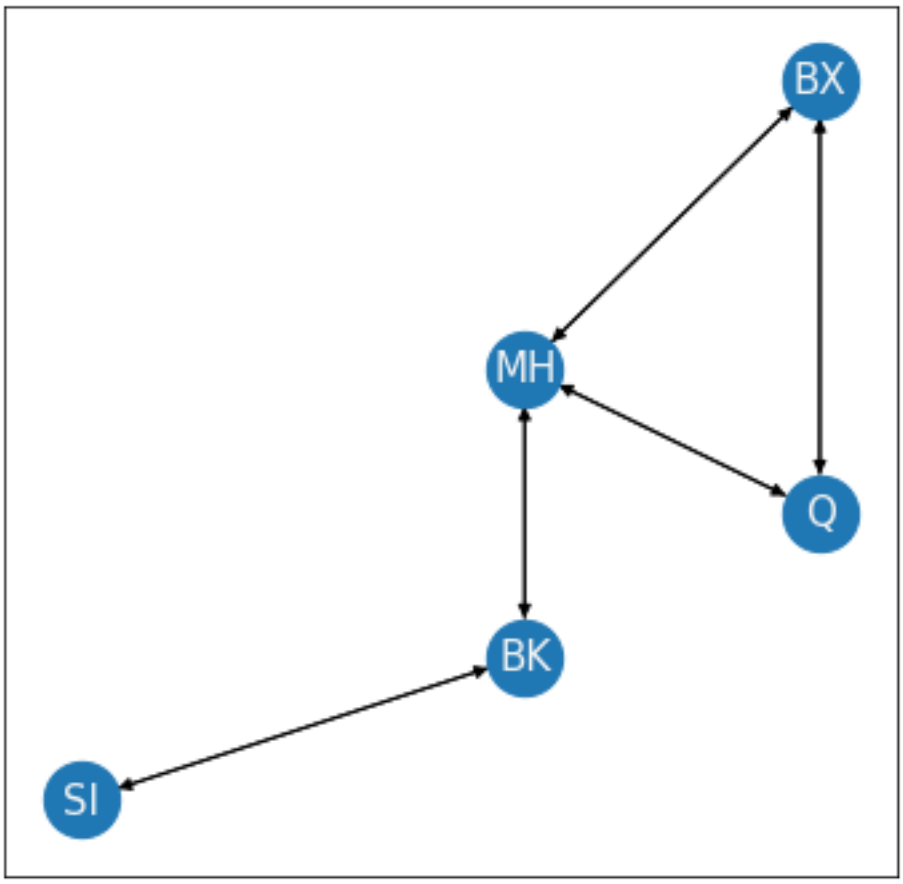
Fortunately, in the context of this book, we will usually only worry about the undirected case, or when the presence of an arrow implies that the other direction exists, too. A network is **undirected** if a connection between node i and node j implies that node j is also connected to node i. For this reason, we will usually omit the arrows entirely, like we show below:



For the adjacency matrix A, remember a connection between nodes i and j is represented by the adjacency a_{ij}. This means that if the network is undirected, $a_{ij} = a_{ji}$, for all pairs of nodes i and j. By definition, this tells us that the adjacency matrix A is symmetric, so $A = A^\top$.

Loopless networks do not have self-loops

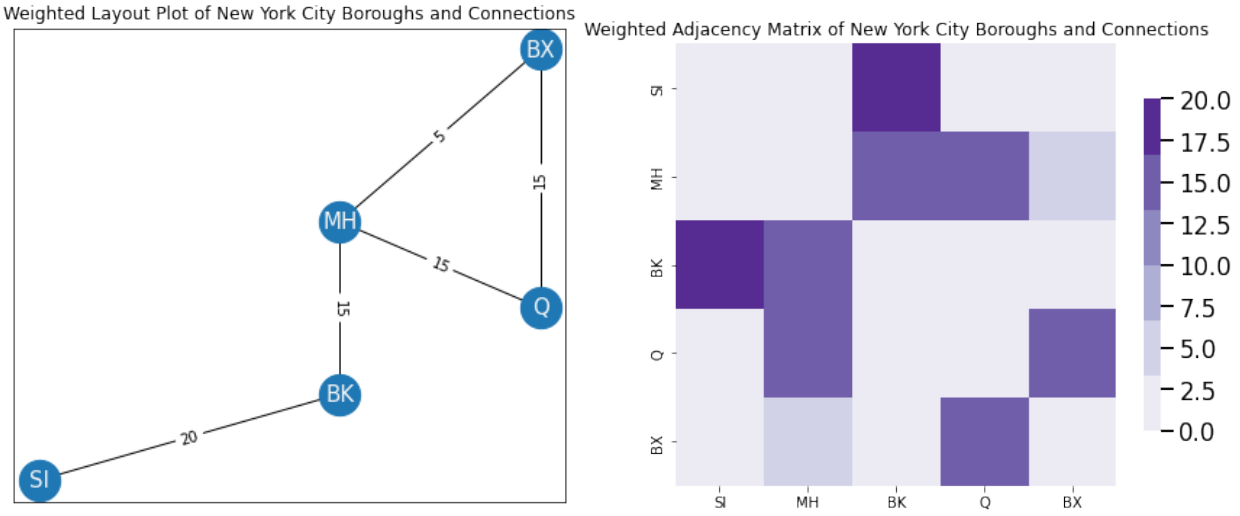
If we are already in a borough, why would we want to take a bridge to that same borough? This logic relates to the concept of *self-loops* in a network. A **self-loop** in a network describes whether nodes can connect back to themselves. For instance, consider the following loop from Staten Island back to itself. This would have the interpretation of a bridge which connects Staten Island back to itself:



In this example, the concept of self-loops is a little trite, but it is worth mentioning as you might see it arise elsewhere. A network is **loopless** if self-loops are not possible. For the adjacency matrix A, a self-loop would be represented by the adjacencies a_{ii} for all nodes i. Note that these entries a_{ii} are all of the *diagonal* entries of A. Therefore, for a network which is loopless, all adjacencies a_{ii} on the diagonal are 0. You might also see this property abbreviated by stating that the diagonal of the adjacency matrix is 0, or $diag(A) = 0$.

Unweighted networks either have an edge, or they don't

Do we need to convey information about how long it takes to get from borough i to borough j with our network? This fundamental question underlies the concept of *weightedness* in networks. We could use things called *edge-weights* $w(i, j)$ could be used to describe the amount of time it takes to get from borough i to borough j. An **edge-weight** $w(i, j)$ assigns a weight to an edge between nodes i and j if that edge exists. If we care about weightedness in the network, the network is called *weighted*. The adjacencies a_{ij} of A for a weighted network take the value of the edge-weight; that is, $a_{ij} = w_{ij}$ for any edge which exists between nodes i and j. In the below plot, edge-weight indicates the approximate time to travel from one borough to the other. The network is undirected, so we don't have to worry about directionality differences. The edge-weight is indicated by the number along the corresponding edge. We can also visualize edge-weights in terms of the adjacency matrix, which we show on the right:



For most examples in this book, we will usually discuss *unweighted* or *binary* networks. A network is **unweighted** or **binary** if we only care about whether edges are *present* or *absent*. In a network which is unweighted, an adjacency a_{ij} takes the value 1 if there is an edge from node i to node j, and takes the value 0 if there is *not* an edge from node i to node j.

This book considers *simple networks*

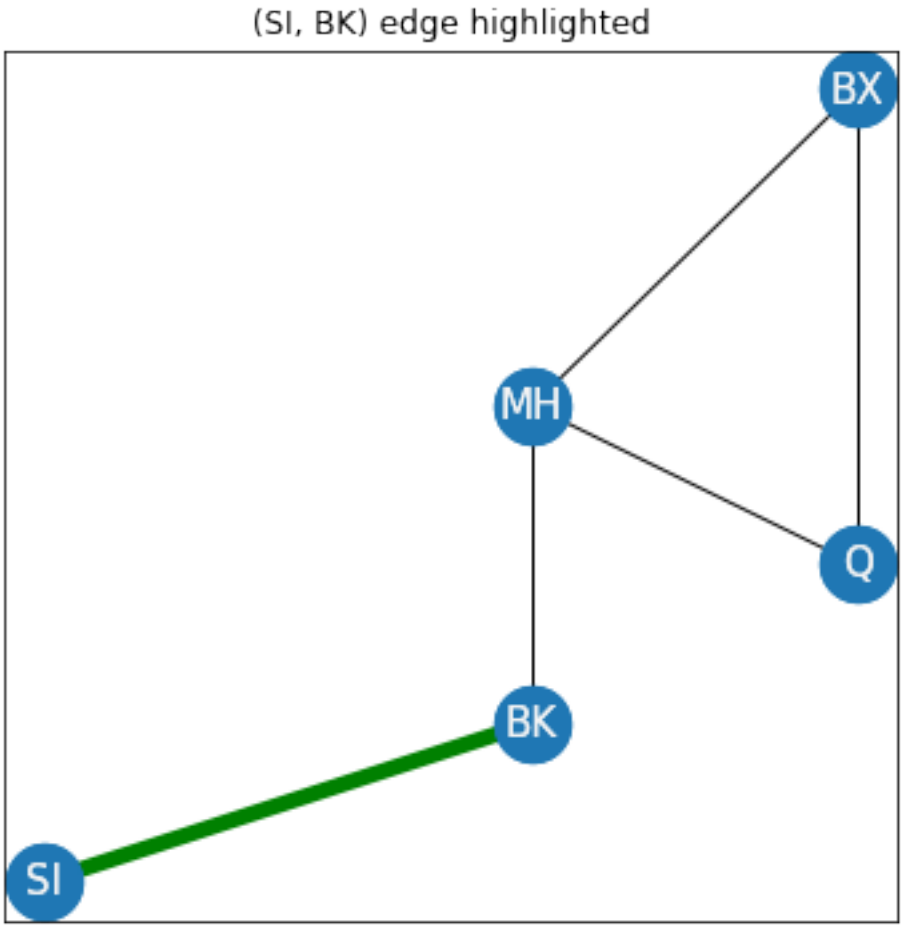
This point is a *really* big deal conceptually for our study of network machine learning. A **simple network** is loopless, undirected, and unweighted. Most of the examples and techniques we look at in this book are developed in the context of simple networks. Fortunately, this note is largely conceptual, and doesn't really impact much from an implementation perspective. All the techniques and packages we use will make sensible choices, or will directly extend, to cases that fall outside of this particular setup. If your networks don't satisfy one or any of these properties, most of the approaches discussed herein will still work. If the technique will not work for the network you have provided, the software package used, `graspologic`, will either give you a warning or an explicit error if there is a substantial issue with the network you have provided.

6.3.2 Descriptive Properties of Nodes

Just like we have many words and properties which describe the network itself, we also have special vocabulary in network machine learning to describe properties about the individual nodes in the network. Remember that the nodes of the network are the n-element set $\mathcal{V}$, which is just the collection $\{v_1, ..., v_n\}$, where v_1 is node 1, v_2 is node 2, so on and so forth. We will tend to use the short-hand v_i to describe the node i, for all nodes from 1 to n.

Node adjacencies and incidences

We begin by descrcibing properties of single nodes in a simple network. The simplest property of a network is *adjacency*. A pair of nodes i and j in an undirected network are **adjacent** or are **neighbors** if an edge exists between them. In terms of the adjacency matrix, two nodes i and j are adjacent/neighbors if the element a_{ij} has a value of one. For instance, in the New York City example, the nodes SI and BK are adjacent/neighbors due to the presence of the green edge, shown in the figure. A related property is known as *incidence*. A node i is **incident** an edge (i, j) or an edge (j, i) if it is one of the two nodes which the edge connects. The adjacencies corresponding to this edge, a_{ij} and a_{ji}, will both take a value of one. For instance, the nodes SI and BK are incident the green edge shown in the figure, as this edge connects SI to BK:



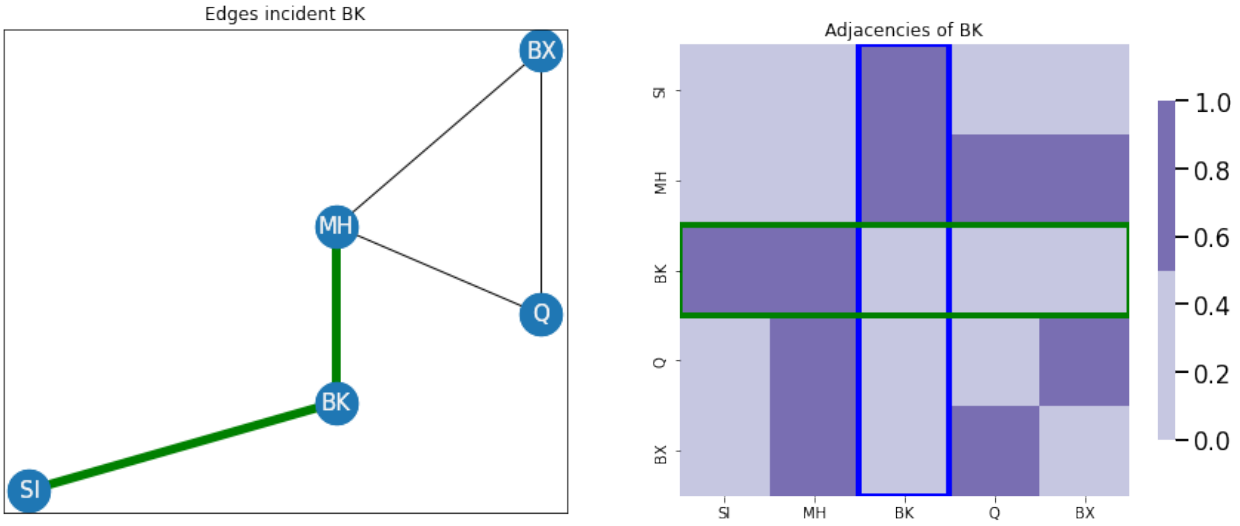
These two nodes are *adjacent* one another due to the fact that an edge exists between them.

Node degree quantifies the number of incidences

The simplest summary statistic for a node is known as the *node degree*. The **node degree** of a node i in a simple network is the number of edges incident to it. Since every edge incident (i, j) which is incident node i takes the value of 1, we can count the adjacencies that correspond to the edges incident node i. If an edge does not exist, the adjacency corresponding to this *potential* edge takes a value of zero. Therefore, we can just sum along the i^{th} row or the i^{th} column of the adjacency matrix, since the row (column) correspond to the edges incident node i:

$$degree(v_i) = \sum_{j=1}^{n} a_{ij} = \sum_{j=1}^{n} a_{ji}$$

This means we will sum all of the potential edges which do *not* exist (any of the a_{ij}s which take a value of zero, and therefore no edge exists between nodes i and j) with all of the edges which *do* exist and are incident node i (since these a_{ij}s will take a value of one). For instance, if we consider the node BK in our example, we have two incident edges, indicated in green, so $degree(v_{BK}) = 2$. When we look at the corresponding adjacency matrix, if we sum the adjacencies for node v_{BK}, we also get two. The adjacencies which would be summed $\sum_{i=1}^{n} a_{ji}$ are shown in blue, and the adjacencies which would be summed $\sum_{j=1}^{n} a_{ij}$ are shown in green:

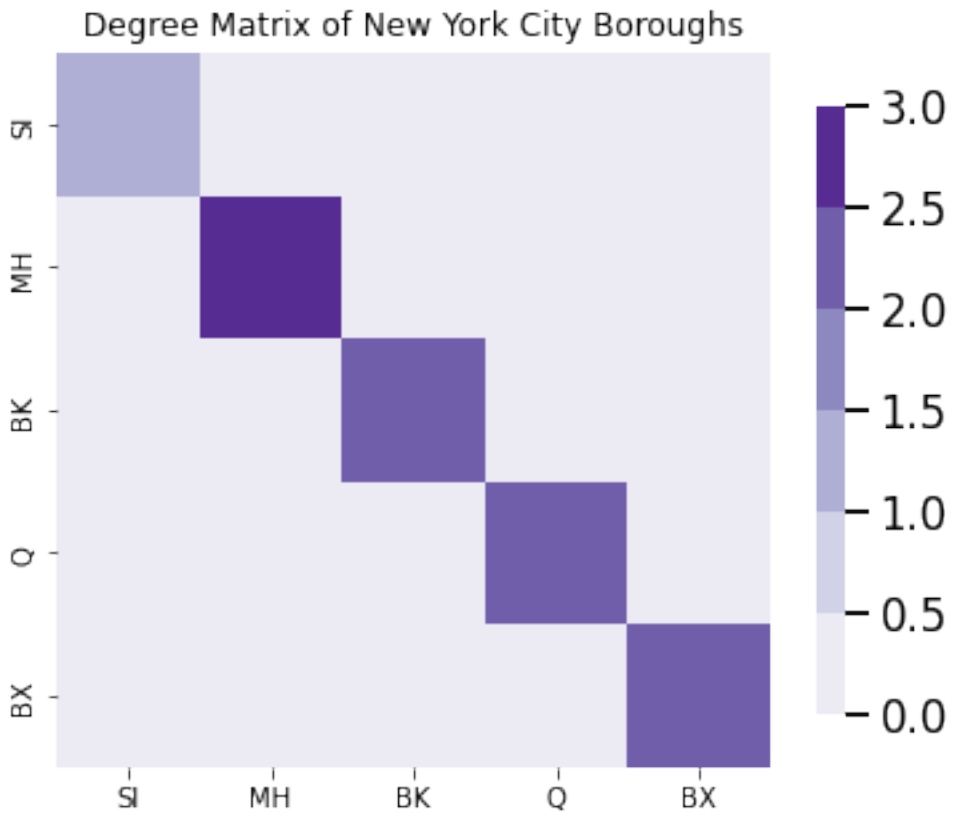


The degree matrix indicates the degrees of each node

A useful quantity which we will come across in many of the later chapters of this book is called the *degree matrix* of the network. The degree matrix is the *diagonal* matrix:

$$D = \begin{bmatrix} d_1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & d_n \end{bmatrix}, \quad d_i = degree(v_i)$$

This matrix D is called **diagonal** because all of the entries $d_{ij} = 0$ unless $i = j$. The diagonal entries d_{ii} of the degree matrix are simply the node degrees $degree(v_i)$ for each node i. Using the counting procedure we described above, we can see that the node SI has degree one, the node BK has degree two, the node MH has degree three, the node Q has degree two, and the node BX has degree two. Therefore, the degree matrix is:



6.3.3 Network summary statistics tell us useful attributes about networks

When we learn about networks, it is often valuable to compute properties of the network so that we can get a better understanding of the relationships within it. We will caall these properties *network summary statistics*. Although this book will focus more on finding and using *representations* of networks than using summary statistics, they're useful to know about. We will introduce two network summary statistics, the network density and the clustering coefficient, and then show an example as to why we do not find summary statistics all that useful for network machine learning.

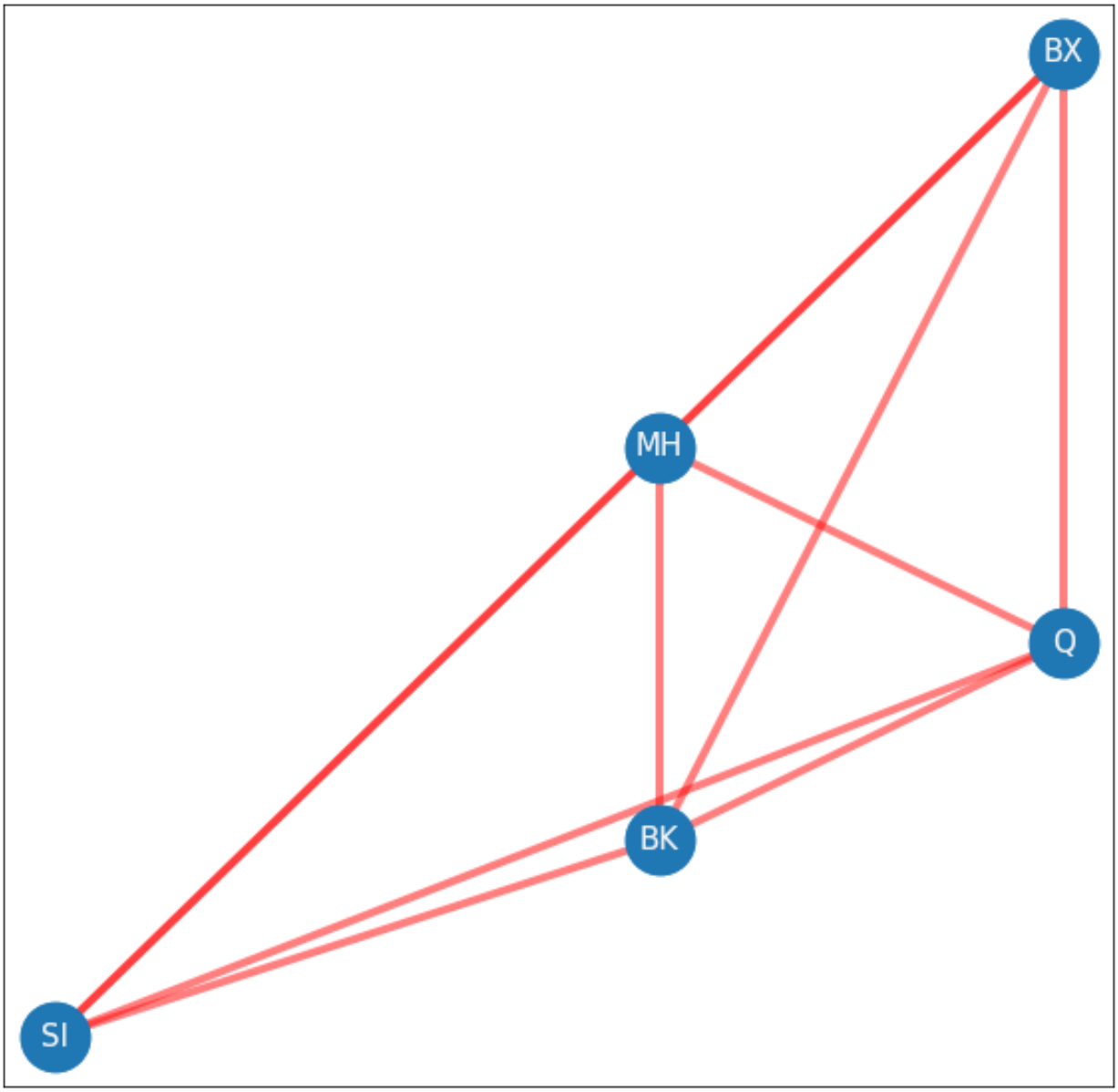
The network density indicates the fraction of possible edges which exist

Given athe adjacency matrix A of a simple network, what fraction of the possible edges *actually* exist?

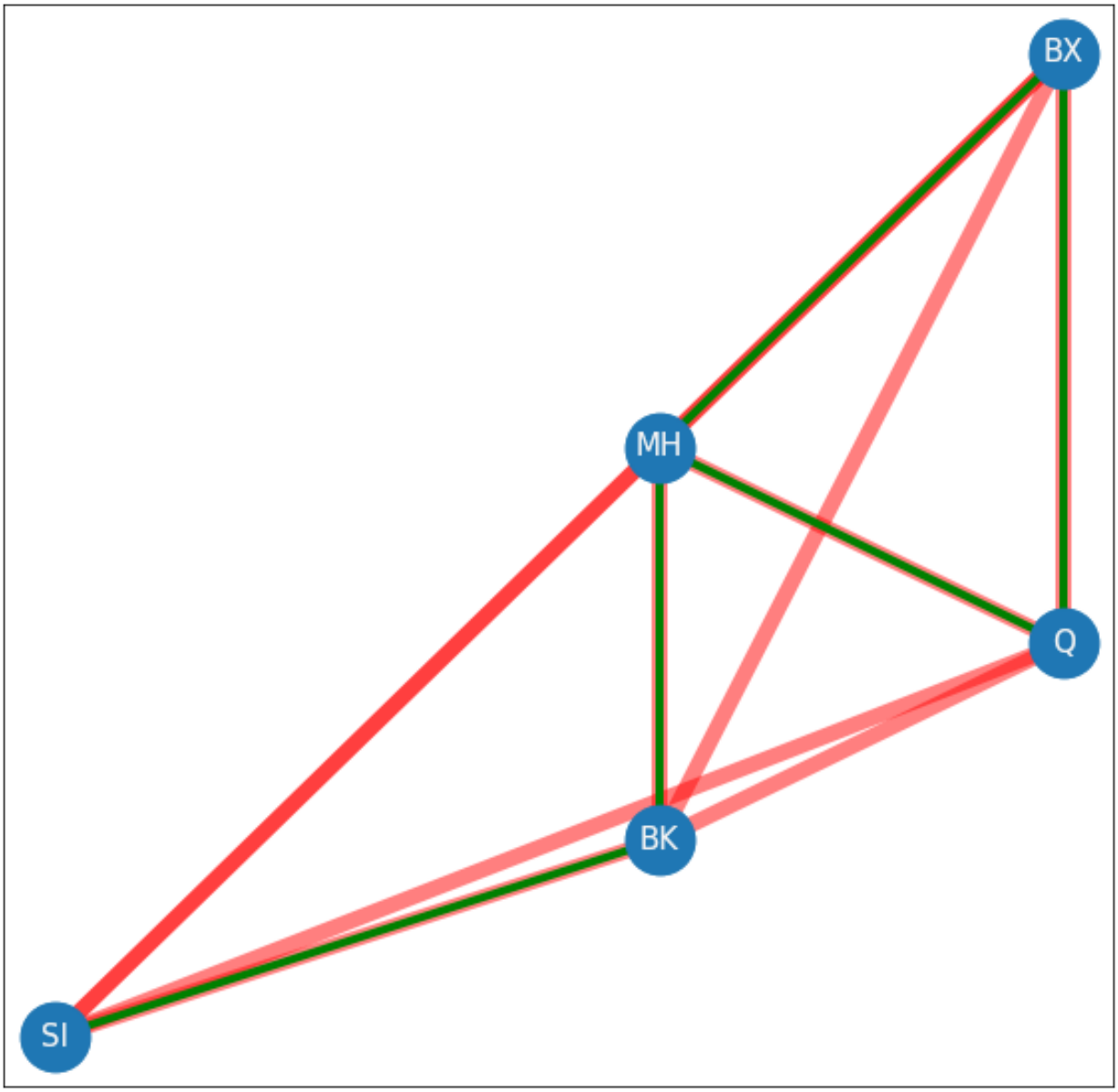
To understand this quantity, first we need to understand how many edges are possible in a network. We have n total nodes in the network, so A is an $n \times n$ matrix. Therefore, A has n^2 total entries. However, it turns out that over *half* of these entries are redundant. Since we said that the network was loopless, this means that every entry is *by default* 0 along the diagonal. Since each node i has a corresponding diagonal entry a_{ii}, this comes to n entries in total that we do not need to count. This leaves our total possible number of edges at n^2 (the total number of entries in the matrix A) minus n (the total number of entries which are automatically 0), or $n^2 - n = n(n-1)$. This quantity represents the total number of possible edges which are *not* in the diagonal.

What else are we overcounting? Well, as it turns out, since the network is also *undirected*, every node that is *not* in the diagonal is also being double counted. Why is this? Remember that an undirected network has an adjacency matrix where for every pair of nodes i and j, $a_{ij} = a_{ji}$. This means that we overcount the number of possible edges not in the diagonal by a factor of *two*, since each off-diagonal entry a_{ij} has a corresponding entry a_{ji}. This leaves the total number of possible edges in the network as $\frac{1}{2}n(n-1)$, or the total number of possible edges not in the diagonal reduced by a factor of two. This quantity is equivalent to the notation $\binom{n}{2}$, which is read as "n *choose* 2". You might see this notation

arise in the study of *combinatorics*, where it is used to answer the question of, "In how many ways can we *choose* two items from n items?" In the network below, we see all of the *possible* edges indicated in red. If you count them up, there are $\frac{1}{2} \cdot 5 \cdot (5-1) = 10$ red edges, in total:



Now, how many edges *actually* exist in our network? The sum of all of the entries of A can be represented by the quantity $\sum_{i=1}^{n} \sum_{i=1}^{n} a_{ij}$, however, there are some redundancies. Remember that A is loopless, so we don't need to count the diagonal entries at all. This brings our quantity to $\sum_{i=1}^{n} \sum_{i \neq j} a_{ij}$, since we don't need to count any edges along the diagonal of A. Next, remember that if an edge in A exists between nodes i and j, that *both* a_{ij} and a_{ji} take the value of 1, due to the undirected property. This means that to obtain the edge count of A, that we only need to count *either* a_{ij} *or* a_{ji}. Somewhat arbitrarily in this book, we will always count the adjacencies a_{ij} in the upper triangle of A, which are the entries where $j > i$. This brings our quantity to $\sum_{i=1}^{n} \sum_{j>i} a_{ij}$, which we can write $\sum_{j>i} a_{ij}$ for short. The edges which exist in our network will be indicated with green, in the following figure, of which there are 6 total. Remember that the red edges were the *possible* edges:



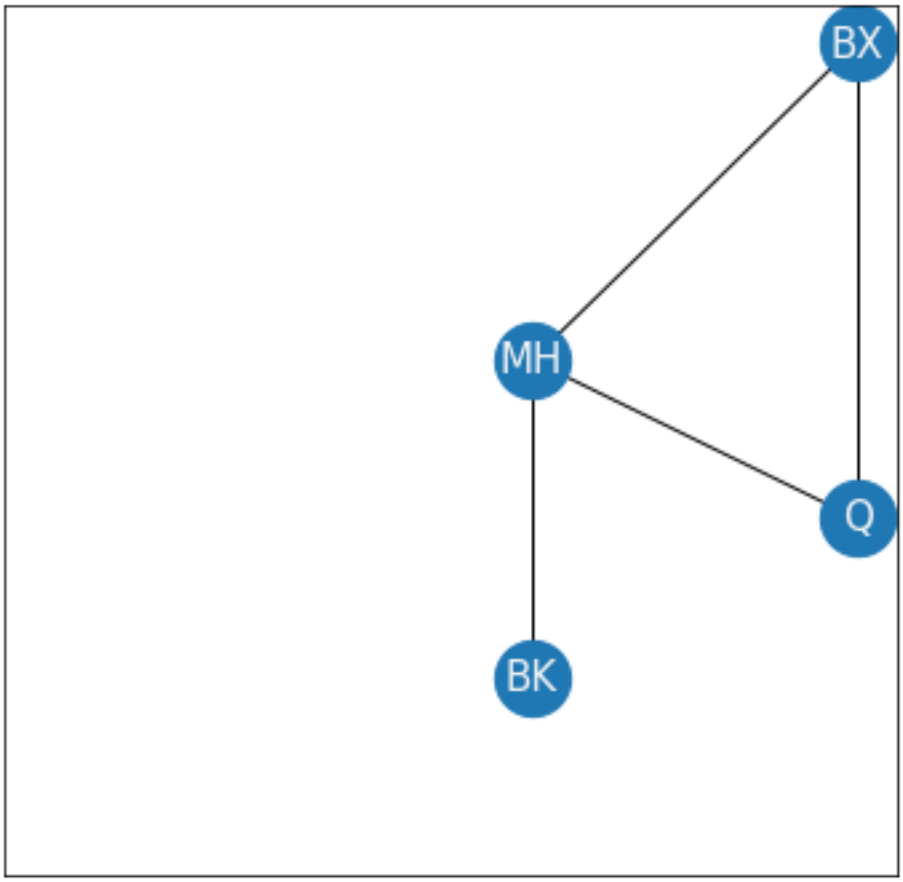
To put it lal together, the **network density** is a summary statistic which indicates the *density of edges* which are present in the network. For a simple network, the network density can be defined as the ratio between the total number of edges in A and the total number of edges possible in A:

$$density(A) = \frac{\sum_{j>i} a_{ij}}{\frac{n(n-1)}{2}} = \frac{2\sum_{j>i} a_{ij}}{n(n-1)}$$

In our example, this is simply the ratio of green edges which *actually* exist to red edges which could *possibly* exist, which is $\frac{5}{10} = 0.5$.

The clustering coefficient indicates how much nodes tend to cluster together

The clustering coefficient inddicates the fraction of triplets of nodes which are closed. What the heck is that? Let's look at only Brooklyn, Manhattan, Queens, and the Bronx:



To begin to define the clustering coefficient, we first must understand what a *triplet* is. A **triplet** is an ordered tuple of three nodes which are connected by two or three edges. For instance, in the above network, we have the following triplets of nodes:

1. (BX, MH, BK), (BX, BK, MH), (MH, BX, BK), (MH, BK, BX), (BK, BX, MH), (BK, MH, BX): two edges,
2. (MH, BK, Q), (MH, Q, BK), (BK, MH, Q), (BK, Q, MH), (Q, MH, BK), (Q, BK, MH): two edges,
3. (BX, MH, Q), (BX, Q, MH), (MH, BX, Q), (MH, Q, BX), (Q, BX, MH), (Q, MH, BX): three edges,

and one three-node sets which has no triplets between {BK, BX, Q}, which has no triplets because there is only a single edge between BX andd Q amongst the three nodes. A triplet is *closed* if there are three edges, and is open if there are only two edges. In our example, there are six closed triplets amongst the nodes {BX, MH, Q}, and there are twele open triplets across {BK, MH, Q} and {BK, MH, BX}. The global clustering coefficient is defined as:

$$C = \frac{\text{number of closed triplets}}{\text{number of closed triplets} + \text{number of open triplets}}$$

In our example, this comes to $C = \frac{6}{6+12} = \frac{1}{3}$. This equation can also be understood in terms of the adjacency matrix. Note that if a triplet between nodes i, j, and k is closed, then all three of the adjacenies a_{ij}, a_{jk}, and a_{ki} have a value of 1. Therefore, if we could the number of times that $a_{ij}a_{jk}a_{ki} = 1$, we also count the number of closed triplets! This means that the number of closed triplets can be expressed as $\sum_{i,j,k} a_{ij}a_{jk}a_{ki}$.

Further, note that for a given node i, that we can find an arbitrary triplet (either open or closed) through the following procedure.

1. Pick a single neighbor j for node i. Note that the node i has a number of neighbors equal to $degree(v_i) = d_i$, so there are d_i possible neighbors to choose from.
2. Pick a different neighbor k for node i. Note that since node i had d_i neighbors, it has $d_i - 1$ neighbors that are not node j.
3. Since we know that nodes j and k are both neighbors of node i, we know that a_{ij} and a_{ik} both have values of one, and therefore the edges (i, j) and (i, k) exist. Therefore, the tuple of nodes (i, j, k) is a triplet, because *at least* two edges exist amongst the three nodes. This tuple is closed if the edge (j, k) exists, and open if the edge (j, k) does not exist.
4. Therefore, there are $d_i(d_i - 1)$ triplets in which node i is the leading node of the triplet.

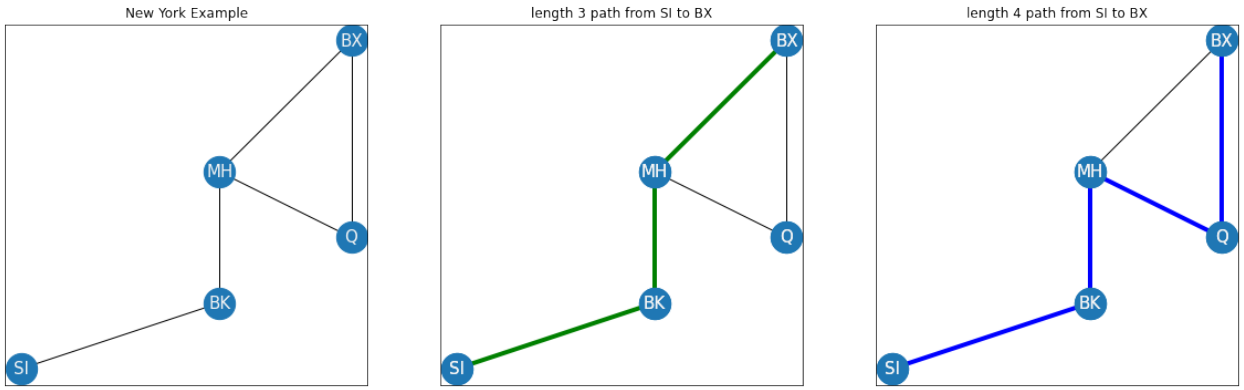
As it turns out, since triplets are *ordered tuples*, we can repeat this procedure for all nodes, and if we count how many triplets we get in total, we get the *total number of triplets* for the entire network. Therefore, the number of open and closed triplets in the network is the quantity $\sum_i d_i(d_i - 1)$. Then we could express the clustering coefficient in terms of the adjacency matrix as:

$$C = \frac{\sum_{i,j,k} a_{ij}a_{jk}a_{ki}}{\sum_i d_i(d_i - 1)}, \quad d_i = degree(v_i)$$

which is a bit easier to implement programmatically.

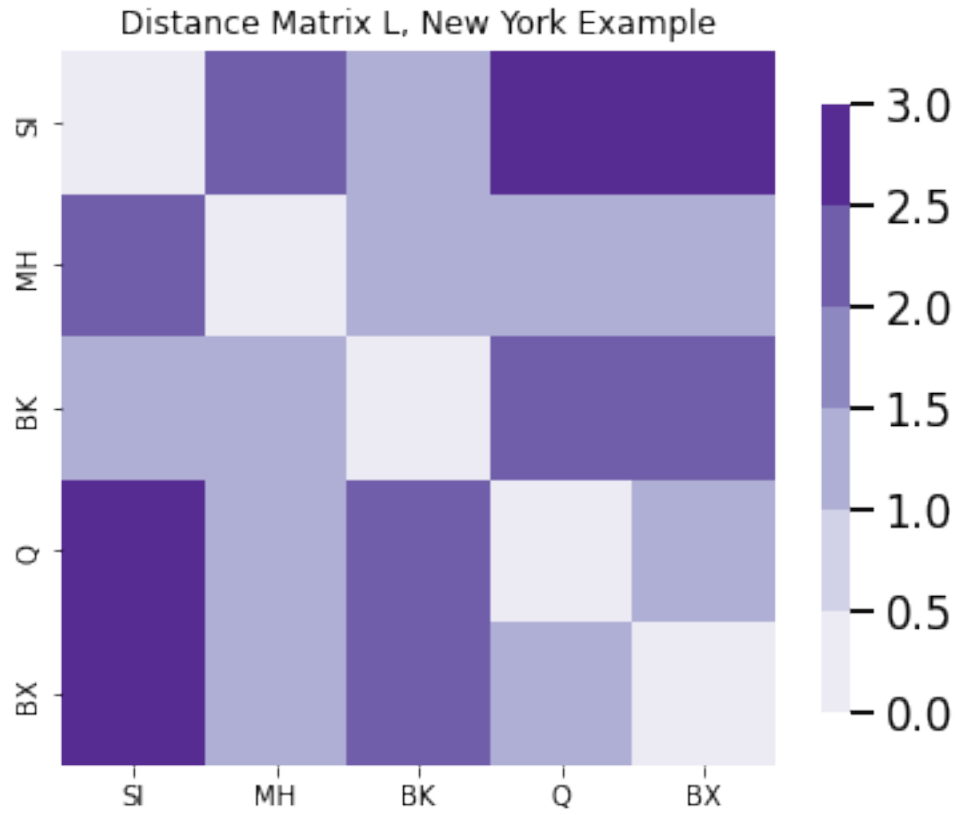
The path length describes how far two nodes are

How many bridges would we need to cross to get from Staten Island to Bronx? This concept relates directly to the concept of the *path length* in a network. A **path** between two nodes i and j is a sequence of edges which starts at node i, and traverses through other nodes in the network until reaching node j. Two nodes are described as **connected** if a path exists between them. The **path length** is the number of edges in the path. For instance, if we remember our network from the New York example, we could get from Staten Island to Bronx in two possible ways, indicated in green and blue in the following example:



In this case, there are only two paths from SI to BX which do not visit the same node more than once, but in a larger network, there may be *many* possible paths from one node to another. For this reason, we will usually be interested in one particular path, the *shortest path*. The **shortest path** or **distance** between nodes i and j is the path with the smallest path length that connects nodes i and j. In our example, the shortest path is indicated by the green edges, and the shortest path length is therefore three. If it is not possible to get from node i to node j using edges of the network, the shortest path length is defined to be infinite. The shortest path between nodes i and j will often be abbreviated using the notation l_{ij}.

A common summary statistic is to view the *distance matrix* L, which is the $n \times n$ matrix whose entries l_{ij} are the shortest path lengths between all pairs of nodes in the network. For our New York example, the distance matrix is:



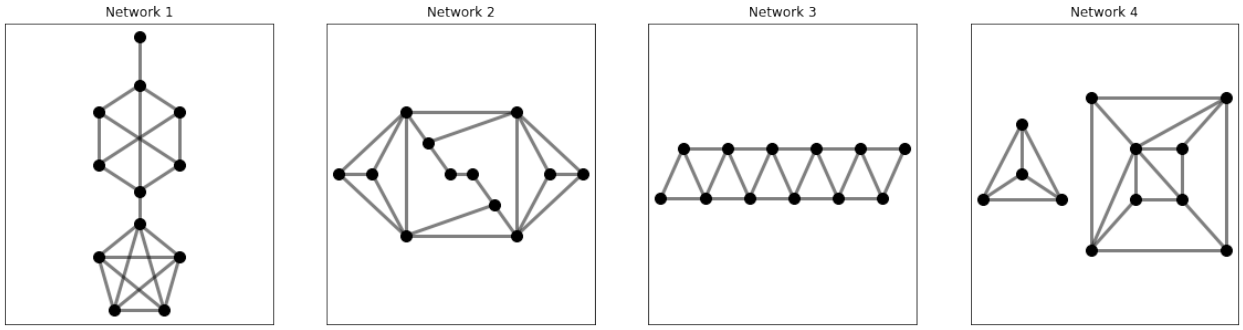
A common network statistic we can compute using the distance matrix is the *average shortest path length*. The average shortest path length l of a simple network is simply the average of all of the shortest paths between two distinct nodes i and j of the distance matrix:

$$l = \frac{1}{n(n-1)} \sum_{i \neq j} l_{ij}$$

Network summary statistics can be misleading when comparing networks

When we perform network machine learning, we want the data we are analyzing to be *sensitive* in the sense that, if two networks are *different* (we use the term *different* a little loosely here, but we will be more specific in a second!) we want the data to reflect that. Let's say we had the following four networks:

```
0.5222222222222223
```



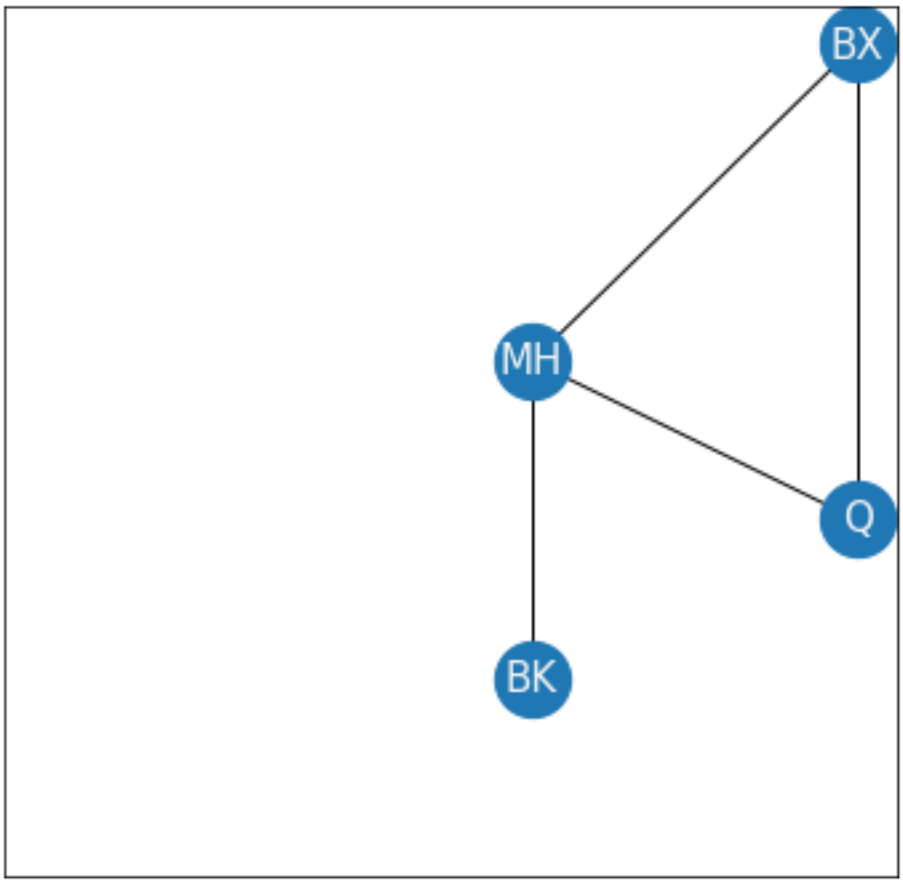
As it turns out, all of these networks share the same number of nodes, the same network density, and the same clustering coefficient:

Network	Network Density	Clustering Coefficient
Network 1	$\frac{1}{3}$	0.6
Network 2	$\frac{1}{3}$	0.6
Network 3	$\frac{1}{3}$	0.6
Network 4	$\frac{1}{3}$	0.6

To conclude our discussion on network properties, we will turn to a final property of a network, known as a subnetwork, which will be useful as a pre-processing step for networks. The concept of a subnetwork will introduce the idea of a *connected network*, which is a network in which a path exists between all pairs of nodes in the network. Network machine learning methods may exhibit unexpected behavior when the network is not connected, so reducing the network to a connected component is often aa useful pre-processing step to prepare data.

6.3.4 Subnetworks are subsets of larger networks

When we think of an entire network, it is often useful to consider it in smaller bits. For instance, when we were looking at the clustering coefficient, we found it useful to break out the nodes {BK, Q, BX, MH} so we could count triplets:



This portion of the network is called a *subnetwork*. A **subnetwork** is a network topology whose nodes and edges are *subsets* of the nodes and edges for another network topology. In this case, the network toplogy of the New York example is $(\mathcal{V}, \mathcal{E})$ defined by the sets:

1. The nodes $\mathcal{V}$: $\{SI, BK, Q, MH, BX\}$,
2. The edges $\mathcal{E}$: $\{(SI, BK), (BK, MH), (MH, Q), (MH, BX), (Q, BX)\}$.

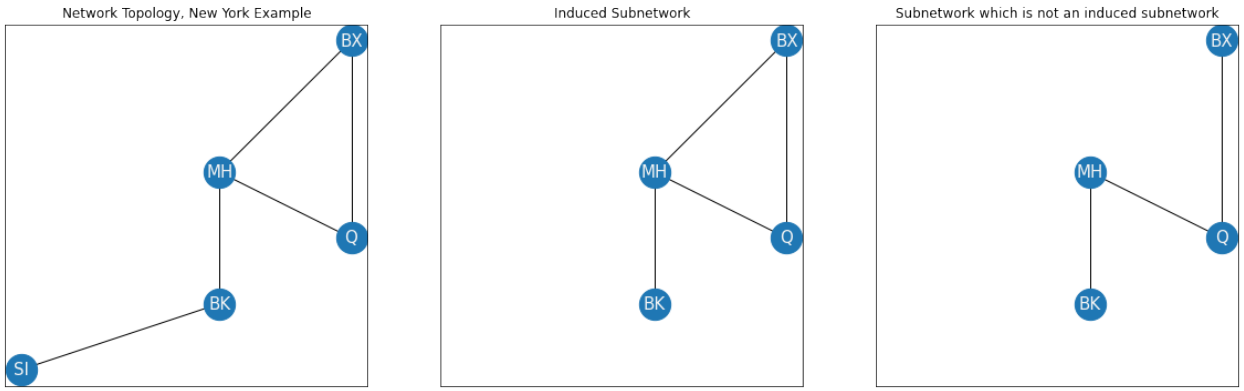
and the subnetwork is the network:

1. The nodes $\mathcal{V}_s$: $\{BK, Q, MH, BX\}$,
2. The edges $\mathcal{E}_s$: $\{(BK, MH), (MH, Q), (MH, BX), (Q, BX)\}$.

As we can see, the subnetwork topology $(\mathcal{V}_s, \mathcal{E}_s)$ is such that every element in $\mathcal{V}_s$ is an element of $\mathcal{V}$, and therefore the nodes of the subnetwork are a subset of the nodes of the complete network. Further, every element in $\mathcal{E}_s$ is an element of $\mathcal{E}$, and therefore the edges of the subnetwork are a subset of the edges of the complete network. So the subnetwork toplogy $(\mathcal{V}_s, \mathcal{E}_s)$ is a subnetwork of the network topology $(\mathcal{V}, \mathcal{E})$. This particular subnetwork can be described further as an **induced** subnetwork. A subnetwork of a network is **induced** by a set of vertices as follows:

1. The nodes $\mathcal{V}_s$ are a subset of the nodes of the network $\mathcal{V}$,
2. The edges $\mathcal{E}_s$ consist of *all* of the edges from the original network which are incident pairs of node in $\mathcal{V}_s$.

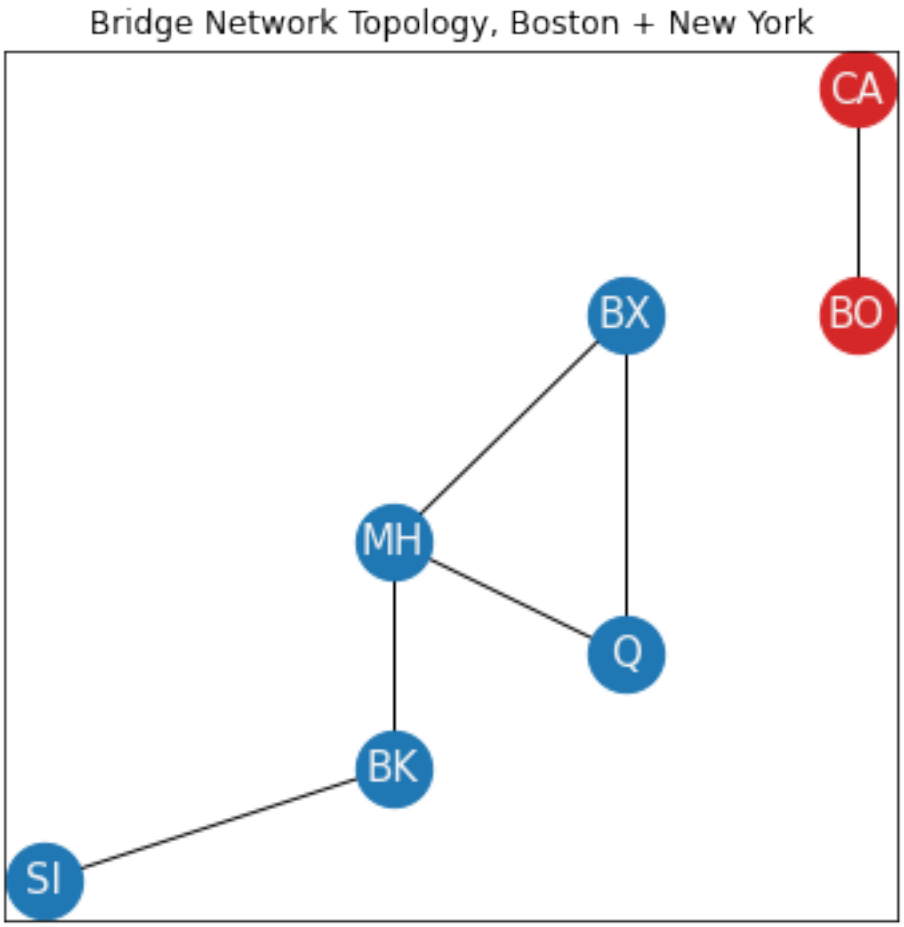
To see an example of a subnetwork which is *not* an induced subnetwork, we can consider a subnetwork which removes one of the edges that exist in the original network:



A particular induced subnetwork that we will often be concerned with is known as the largest connected component (LCC).

The largest connected component (LCC) is the largest subnetwork of connected nodes

To define the largest connected component, we'll modify our example slightly. Let's say our network also includes the Boston area, and we have two new nodes, Boston (BO) and Cambridge (CA). Boston and Cambridge are incident several bridges between one another, so an edge exists between them. However, there are no bridges between boroughs of New York and the Boston area, so there are no edges from nodes in the Boston area to nodes in the New York area:



The entire network topology can be described by the sets:

1. $\mathcal{V} = \{SI, MH, BK, BX, Q, CA, BO\}$,
2. $\mathcal{E} = \{(SI, BK), (MH, BK), (MH, Q), (MH, BX), (MX, Q), (CA, BO)\}$.

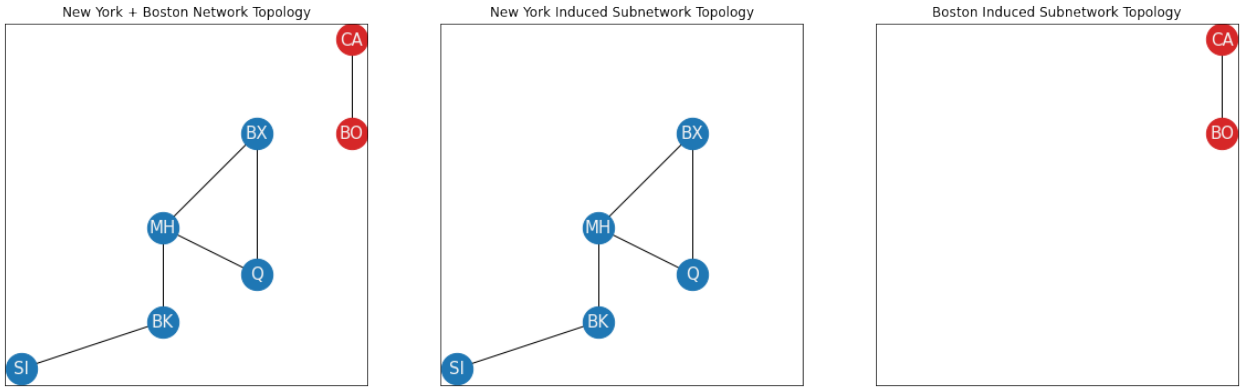
Notice that we have two distinct sets of nodes, those of New York and those of Boston, which are *only* connected amongst one another. Formally, these two sets of nodes can be described as inducing *connected components* of the network topology $(\mathcal{V}, \mathcal{E})$. A **connected component** is an induced subnetwork in which any two nodes are connected to each other by a path through the network. The two connected components are the New York induced subnetwork:

1. The nodes $\mathcal{V}_N$: $\{SI, BK, Q, MH, BX\}$,
2. The edges $\mathcal{E}_N$: $\{(SI, BK), (BK, MH), (MH, Q), (MH, BX), (Q, BX)\}$.

and the Boston induced subnetwork:

1. The nodes $\mathcal{V}_B$: $\{CA, BO\}$,
2. The edges $\mathcal{E}_B$: $\{(CA, BO)\}$.

which we can represent visually here:



The **largest connected component** (LCC) of a network is the connected component with the most nodes. In our example, the New York connected component has five nodes, whereas the Bosston connected component has two nodes. Therefore, the New York connected component is the LCC.

6.4 Regularization

In practice, many networks we will encounter in network machine learning will *not* be simple networks. As we discussed in the preceding discussion, many of the techniques we discuss will be just fine to use with weighted networks. Unfortunately, real world networks are often extremely noisy, and so the analysis of one real world network might not generalize very well to a similar real world network. For this reason, we turn to *regularization*. **Regularization** is defined as, "the process of adding information in order to solve an ill-posed problem or to prevent overfitting." In network machine learning, what this usually will entail is modifying the network (or networks) themselves to allow better generalization of our statistical inference to new datasets. For each section, we'll pose an example, a simulation, and code for how to implement the desired regularization approach. It is important to realize that you might use several of these techniques simultaneously in practice, or you might have a reason to use these techniques that go outside of our working examples.

To start this section off, we're going to introduce an example that's going to be fundamental in many future sections we see in this book. We have a group of 50 local students who attend a school in our area. The first 25 of the students polled are athletes, and thhhe second 25 of the students polled are in marching band. We want to analyze how good of friends the students are, and to do so, we will use network machine learning. The nodes of the network will be the students. Next, we will describe how the two networks are collected:

1. Activity/Hobby Network: To collect the first network, we ask each student to select from a list of 50 school activities and outside hobbies that they enjoy. For a pair of students i and j, the weight of their interest alignment will be a score between 0 and 50 indicating how many activities or hobbies that they have in common. We will refer to this network as the common interests network. This network is obviously undirected, since if student i shares x activities or hobbies with student j, then student j also shares x activities or hobbies with student i. This network is weighted, since the score is between 0 and 50. Finally, this network is loopless, because it would not make sense to look at the activity/hobby alignment of a student with themself, since this number would be largely uninformative as every student would have perfect alignment of activities and hobbies with him or herself.
2. Friendship Network: To collect the second network, we ask each student to rate how good of friends they are with other students, on a scale from 0 to 1. A score of 0 means they are not friends with the student or do not know the student, and a score of 1 means the student is their best friend. We will refer to this network as the friendship network. This nework is clearly directed, since two students may differ on their understanding of how good of friends they are. This network is weighted, since the score is between 0 and 1. Finally, this network is also loopless, because it would not make sense to ask somebody how good of friends they are with themself.

Our scientific question of interest is how well activities and hobbies align with perceived notions of friendship. We want to use the preceding networks to learn about a hypothetical third network, a network whose nodes are identical to the two networks above, but whose edges are whether the two individuals are friends (or not) on facebook. To answer this question, we have quite the job to do to make our networks better suited to the task! We begin by simulating some example data, shown below as adjacency matrix heatmaps:

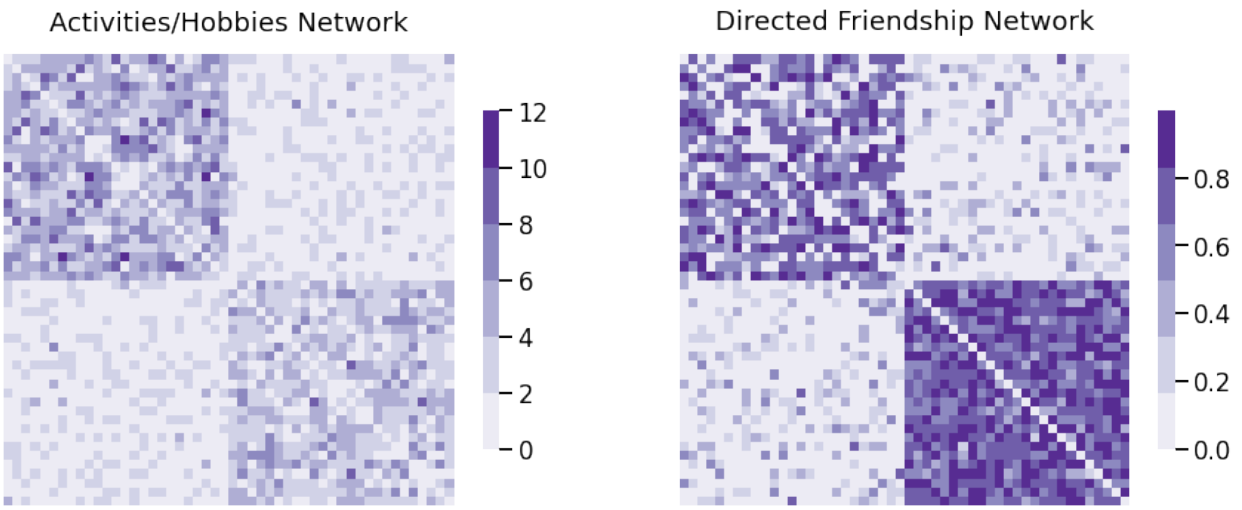
```
from graspologic.simulations import sbm
import numpy as np

wtargsa = [[dict(n=50, p=.09), dict(n=50, p=.02)],
           [dict(n=50, p=.02), dict(n=50, p=.06)]]

wtargsf = [[dict(a=4, b=2), dict(a=2, b=5)],
           [dict(a=2, b=5), dict(a=6, b=2)]]

# human brain network
A_activity = sbm(n=[25, 25], p=[[1,1], [1,1]], wt=np.random.binomial, wtargs=wtargsa,␣
↪loops=False, directed=False)

# alien brain network
A_friend = sbm(n=[25, 25], p=[[.8, .4], [.4, 1]], wt=np.random.beta, wtargs=wtargsf,␣
↪directed=True)
```



6.4.1 Regularization of the Nodes

The Largest Connected Component is the largest subnetwork of connected nodes

We have already learned about the LCC in the preceding section, so we won't cover the in-depth, but it is important to realize that this is a node regularization technique.

Degree trimming removes nodes with low degree

Let's imagine that in our friendship network, there were an additional three athlete students from a nearby school. Perhaps one of these students had a friend in the first school he met at a sports camp, so these students are not a separate component of the network entirely. Even though these students are not *totally* disconnected from the rest of the network entirely, and therefore would not be removed by computing the LCC, their presence in our analysis might still lead to stability issues in future network machine learning tasks. For this reason, it may be advantageous to remove nodes whose degrees are much different from the other nodes in the network.

6.4.2 Regularizing the Edges

Symmetrizing the network gives us undirectedness

If we wanted to learn from the friendship network about whether two people were friends on facebook, a reasonable first place to start might be to *symmetrize* the friendship network. The facebook network is *undirected*, which means that if a student i is friends on facebook with student j, then student j is also friends with student i. On the other hand, as we learned above, the friendship network was directed. Since our question of interest is about an undirected network but the network we have is directed, it might be useful if we could take the directed friendship network and learn an undirected network from it. This relates directly to the concept of *interpretability*, in that we need to represent our friendship network in a form that will produce an answer or us about our facebook network which we can understand.

Another reason we might seek to symmetrize the friendship network is that we might think that asymmetries that exist in the network are just *noise*. We might assume that the adjacency entries a_{ij} and a_{ji} relate to one another, so together they might be able to produce a single summary number that better summarizes their relationship all together.

Remember that in a symmetric network, $a_{ij} = a_{ji}$, so in an *asymmetric* network, $a_{ij} \neq a_{ji}$. To symmetrize the friendship network, what we want is a *new* adjacency value, which we will call w_{ij}, which will be a function of a_{ij} and a_{ji}. Then, we will construct a new adjacency matrix A', where each entry a'_{ij} *and* a'_{ji} are set equal to w_{ij}. The little apostrophe just signifies that this is a potentially different value than either a_{ij} or a_{ji}. Note that by construction, A' is in fact symmetric, because $a'_{ij} = a'_{ji}$ due to how we built A'.

Ignoring a "triangle" of the adjacency matrix

The easiest way to symmetrize a network A is to just ignore part of it entirely. In the adjacency matrix A, you will remember that we have an upper and a lower triangular part of the matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n} \\ a_{n1} & \cdots & a_{n,n-1} & a_{nn} \end{bmatrix},$$

The entries which are listed in red are called the **upper right triangle of the adjacency matrix above the diagonal**. You will notice that for the entries in the upper right triangle of the adjacency matrix, a_{ij} is such that j is *always* greater than i. Similarly, the entries which are listed in blue are called the **lower left triangle of the adjacency matrix below the diagonal**. In the lower left triangle, i is *always* greater than j. These are called *triangles* because of the shape they

make when you look at them in matrix form: notice, for instance, that in the upper right triangle, we have a triangle with three corners of values: a_{12}, a_{1n}, and $a_{n-1,n}$.

So, how do we ignore a triangle all-together? Well, it's really quite simple! We will visually show how to ignore the lower left triangle of the adjacency matrix. We start by forming a triangle matrix, Δ, as follows:

$$\Delta = \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n} \\ 0 & \dots & 0 & 0 \end{bmatrix},$$

Notice that this matrix *keeps* all of the upper right triangle of the adjacency matrix above the diagonal the same as in the matrix A, but replaces the lower left triangle of the adjacency matrix below the diagonal and the diagonal with 0s. Notice that the transpose of Δ is the matrix:

$$\Delta^\top = \begin{bmatrix} 0 & 0 & \dots & 0 \\ a_{12} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ a_{1n} & \dots & a_{n-1,n} & 0 \end{bmatrix}$$

So when we add the two together, we get this:

$$\Delta + \Delta^\top = \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ a_{12} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n} \\ a_{1n} & \dots & a_{n-1,n} & 0 \end{bmatrix},$$

We're almost there! We just need to add back the diagonal of A, which we will do using the matrix $diag(A)$ which has values $diag(A)_{ii} = a_{ii}$, and $diag(A)_{ij} = 0$ for any $i \neq j$:

$$A' = \Delta + \Delta^\top + diag(A) = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{12} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n} \\ a_{1n} & \dots & a_{n-1,n} & a_{nn} \end{bmatrix},$$

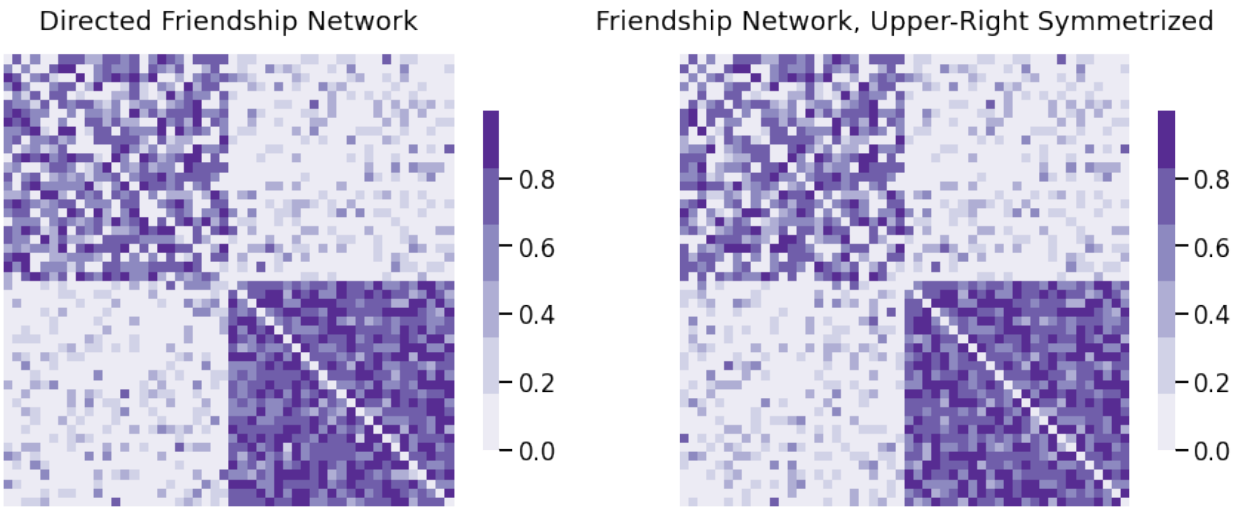
Which leaves A' to be a matrix consisting *only* of entries which were in the upper right triangle of A. A' is obviously symmetric, because $a'_{ij} = a'_{ji}$ for all i and j. Since the adjacency matrix is symmetric, the network A' represents is undirected.

So what does this mean in terms of the network itself? What this means is that the network originally had edge weights a_{ij}, where a_{ij} might not be equal to a_{ji}. This means student i might perceive their friendship with student j as being stronger or weaker than student j perceived about student i. What we did here was we basically just ignored any perceived friendships a_{ji} when j exceeded i (the lower left triangle), and simply "replaced" that perceived friendship with the corresponding entry a_{ij} in the upper right triangle of the adjacency matrix. This produced for us a single friendship strength a'_{ij} where $a'_{ij} = a'_{ji}$.

In graspologic, we can implement this as follows:

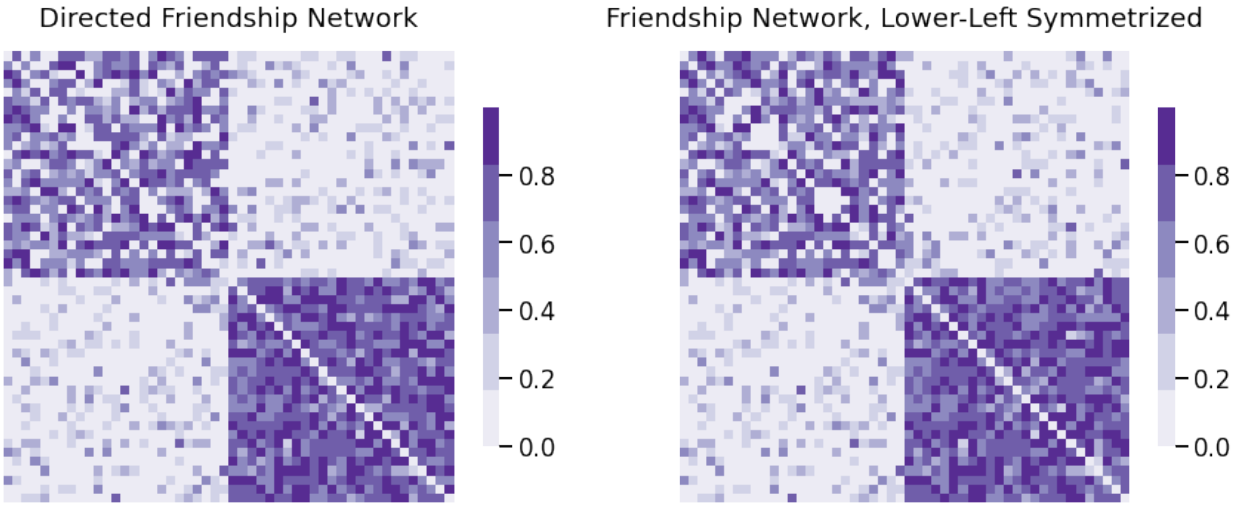
```
from graspologic.utils import symmetrize

# symmetrize with upper right triangle
A_friend_upright_sym = symmetrize(A_friend, method="triu")
```



Likewise, we can lower-left symmetrize as well:

```
# symmetrize with lower left triangle
A_friend_lowleft_sym = symmetrize(A_friend, method="tril")
```



Taking a function of the two values

There are many other ways we can also take a function of a_{ij} and a_{ji} to get a symmetric matrix. One is to just average the two. That is, we can let the matrix A' be the matrix with entries $a'_{ij} = \frac{a_{ij}+a_{ji}}{2}$ for all i and j. In matrix form, this

operation looks like this:

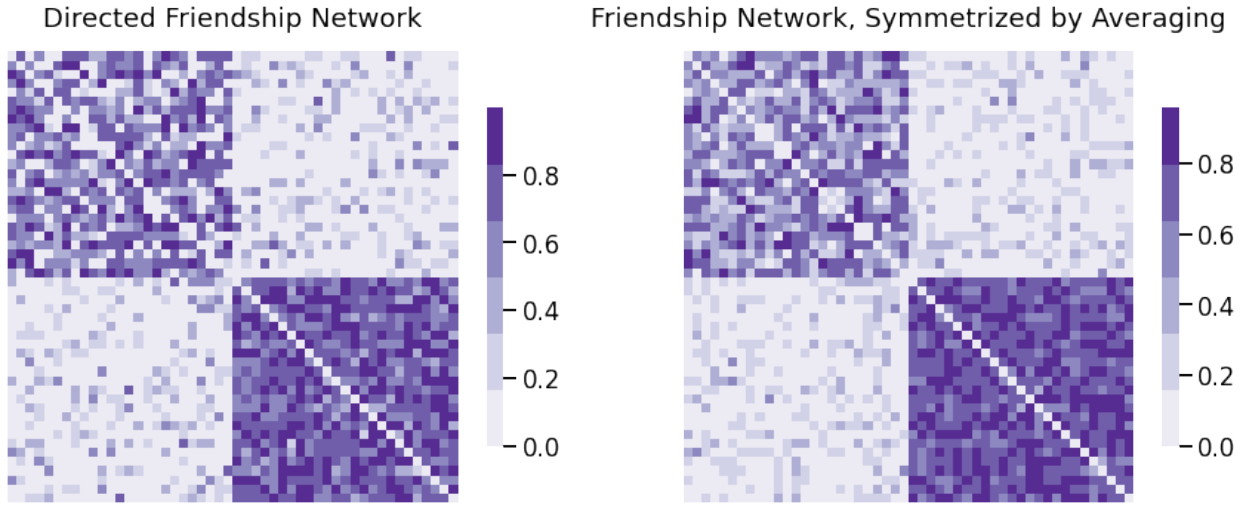
$$
\begin{aligned}
A' &= \frac{1}{2}(A + A^\top) \\
&= \frac{1}{2}\left(\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} + \begin{bmatrix} a_{11} & \dots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1n} & \dots & a_{nn} \end{bmatrix}\right) \\
&= \begin{bmatrix} \frac{1}{2}(a_{11} + a_{11}) & \dots & \frac{1}{2}(a_{1n} + a_{n1}) \\ \vdots & \ddots & \vdots \\ \frac{1}{2}(a_{n1} + a_{1n}) & \dots & \frac{1}{2}(a_{nn} + a_{nn}) \end{bmatrix} \\
&= \begin{bmatrix} a_{11} & \dots & \frac{1}{2}(a_{1n} + a_{n1}) \\ \vdots & \ddots & \vdots \\ \frac{1}{2}(a_{n1} + a_{1n}) & \dots & a_{nn} \end{bmatrix}
\end{aligned}
$$

As we can see, for all of the entries, $a'_{ij} = \frac{1}{2}(a_{ij} + a_{ji})$, and also $a'_{ji} = \frac{1}{2}(a_{ji} + a_{ij})$. These quantities are the same, so $a'_{ij} = a'_{ji}$, and A' is symmetric. As the adjacency matrix is symmetric, the network that A' represents is undirected.

Remember that the asymmetry in the friendship network means student i might perceive their friendship with student j as being stronger or weaker than student j perceived about student i. What we did here was instead of just arbitrarily throwing one of those values away, we said that their friendship might be better indicated by averaging the two values. This produced for us a single friendship strength a'_{ij} where $a'_{ij} = a'_{ji}$.

We can implement this in graspologic as follows:

```
# symmetrize with averaging
A_friend_avg_sym = symmetrize(A_friend, method="avg")
```



We will use the friendship network symmetrized by averaging in several of the below examples, which we will call the "undirected friendship network".

Diagonal augmentation

In our future works with network machine learning, we will come across numerous techniques which operate on adjacency matrices which are *positive semi-definite*. This word doesn't mean a whole lot to us for network machine learning, but it has a big implication when we try to use algorithms on many of our networks. Remember that when we have a loopless network, a common practice in network science is to set the diagonal to zero. What this does is it leads to our adjacency matrices being *indefinite* (which means, *not* positive semi-definite). For us, this means that many network machine learning techniques simply cannot operate on these adjacency matrices. However, as we mentioned before, these entries are not actually zero, but simply *do not exist* and we just didn't have a better way to represent them. Or do we?

Diagonal augmentation is a procedure for imputing the diagonals of adjacency matrices for loopless networks. This gives us "placeholder" values that do not cause this issue of indefiniteness, and allow our network machine learning techniques to still work. Remember that for a simple network, the adjacency matrix will look like this:

$$A = \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ a_{21} & \ddots & & \vdots \\ \vdots & & \ddots & a_{n-1,n} \\ a_{n1} & \dots & a_{n,n-1} & 0 \end{bmatrix}$$

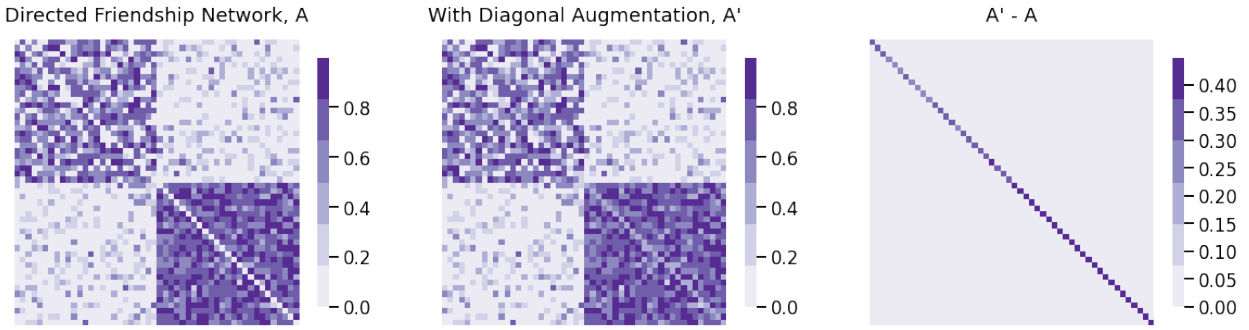
What we do is impute the diagonal entries using the *fraction of possible edges which exist* for each node. This quantity is simply the node degree d_i (the number of edges which exist for node i) divided by the number of possible edges node i could have (which would be node i connected to each of the other $n-1$ nodes). Remembering that the degree matrix D is the matrix whose diagonal entries are the degrees of each node, the diagonal-augmented adjacency matrix is given by:

$$A' = A + \frac{1}{n-1}D = \begin{bmatrix} \frac{d_1}{n-1} & a_{12} & \dots & a_{1n} \\ a_{21} & \ddots & & \vdots \\ \vdots & & \ddots & a_{n-1,n} \\ a_{n1} & \dots & a_{n,n-1} & \frac{d_n}{n-1} \end{bmatrix}$$

When the matrices are directed or weighted, the computation is a little different, but fortunately `graspologic` will handle this for us. Let's see how we would apply this to the directed friendship network:

```
from graspologic.utils import augment_diagonal

A_friend_aug = augment_diagonal(A_friend)
```



As we can see, the diagonal-augmented friendship network and the original directed friendship network differ only in that the diagonals of the diagonal-augmented friendship network are non-zero.

Lowering edge bias

As you are probably aware, in all of machine learning, we are always concerned with the *bias/variance tradeoff*. The **bias/variance tradeoff** is an unfortunate side-effect that concerns how well a learning technique will generalize to new datasets.

1. **Bias** is a simplifying assumption of a model that makes the task easier to estimate. For instance, if we have a friendship network, we might make simplifying assumptions, such as an assumption that two athletes from different sports have an equally likely chance of being friends with a member of the band.
2. On the other hand, the **variance** is the degree to which the an estimate of a task will change when given new data. An assumption that if a player is a football player he has a higher chance of being friends with a band member might make sense given that the band performs at football games.

The "trade-off" is that these two factors tend to be somewhat at odds, in that raising the bias tends to lower the variance, and vice-versa:

1. **High bias, but low variance**: Whereas a lower variance model might be better suited to the situation when the data we expect to see is noisy, it might not as faithfully represent the underlying dynamics we think the network possesses. A low variance model might ignore that athletes might have a different chance of being friends with a band member based on their sport all together. This means that while we won't get the student relationships *correct*, we might still be able to get a reasonable estimate that we think is not due to overfitting.
2. **Low bias, but high variance**: Whereas a low bias model might more faithfully model true relationships in our training data, it might fit our training data a little *too* well. Fitting the training data too well is a problem known as **overfitting**. If we only had three football team members and tried to assume that football players were better friends with band members, we might not be able to well approximate this relationship because of how few individuals we have who reflect this situation.

Here, we show several strategies to reduce the bias due to edge weight noise in network machine learning.

Thresholding converts weighted networks to binary networks

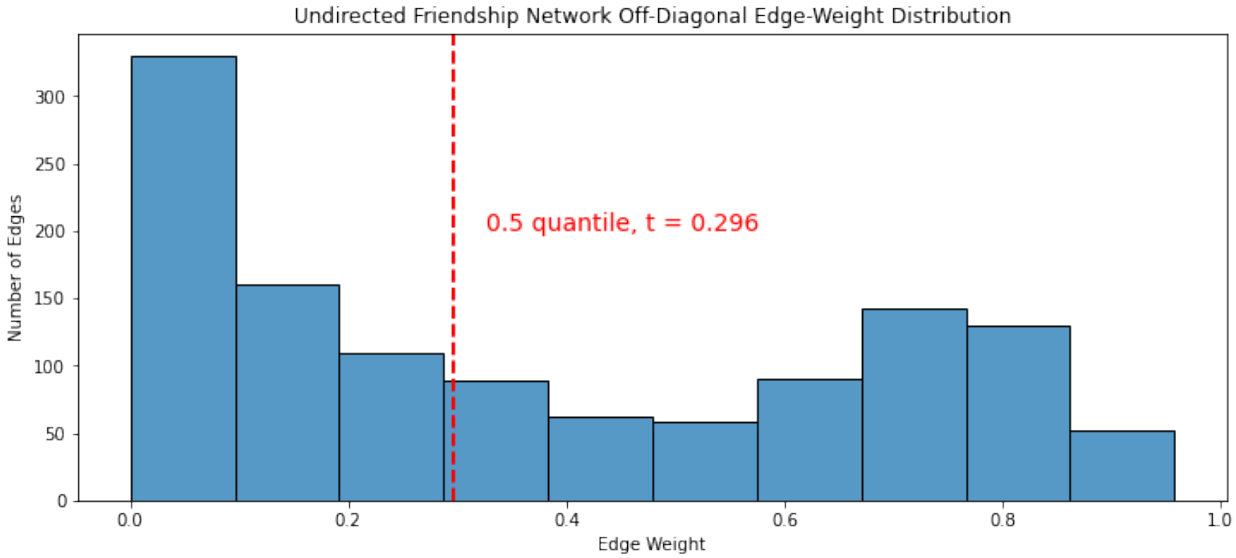
The simplest way to reduce edge bias is the process of *thresholding*. Through thresholding, we choose a threshold value, t. Next, we simply set all of the entries of the adjacency matrix less than or equal to t to zero, and the entries of the adjacency matrix above t to one.

Some of the most common approaches to choosing this threshold are:

1. Set the threshhold to zero: set all non-zero weighted entries to one, and all zero-weight entries to zero. This is most commonly used when we see zero-inflated networks, or networks where the adjacency matrix takes values that are either zero or some quantity different from one,
2. Set te threshold to be the mean: set all values below the mean edge-weight to zero, and all values above the mean edge-weight to one,
3. Use a quantile: A quantile is a percentile divided by 100. In this strategy, we identify a target quantile of the edge-weight distribution. What this means is that we are selecting the lowest *fraction* of the edge-weights (where that fraction is the quantile that we choose) and setting these edges to 0, and selecting the remaining edges to 1. If we select a quantile of 0.5, this means that we take the smallest 50% of edges and set them to zero, and the largest 50% of edges and set them to 1.

We will show how to use the percentile approach to binarization, with both our activity/hobby and friendship networks. We will threshold using the edge-weight in the 50^{th} percentile. Our example networks of activity/hobby and friendship were loopless, as you could see above. Remember as we learned in the preceding section, that if the network itself is loopless, the diagonal entries simply *do not exist*; 0 is simply a commonly used placeholder. For this reason, when we compute percentiles of edge-weights, we need to *exclude the diagonal*. Further, since this network is undirected, we also need to restrict our attention to one triangle of the corresponding adjacency matrix. We choose the upper-right triangle

arbitrarily, as the adjacency matrix's symmetry means the upper-right triangle and lower-right triangle have identical edge-weight distributions. We begin by using this procedure on the friendship network. To complete this processs, we first look at the edge-weight distribution for the friendship network, which is shown below, and identified the edge-weight at the 0.5 quartile:



The 0.5 quantile, it turns out, is about 0.3. This is because about 50% of the edges are less than this threshold, and about 50% of the edges are greater than this threshold. There is exactly one more edge in less than or equal to t, because this edge is exactly the median (an alternative name for the 0.5 quartile) value:

```
Number of edges less than or equal to t: 613
Number of edges greater than or equal to t: 612
```

Next, we will assign the edges less than or equal to t to zero, and the edges greater than or equal to t to one:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-14-fb4cba753867> in <module>
----> 1 A_friend_thresh = copy(A_friend_avg_sym)  # copy the network over
      2
      3 # threshold using t
      4 A_friend_thresh[A_friend_thresh <= t] = 0
      5 A_friend_thresh[A_friend_thresh > t] = 1

NameError: name 'copy' is not defined
```

Since the friendship network is now undirected (we made the adjacency matrix symmetric through averaging), loopless (because we defined it that way), and binary (because we thresholded the edges), we have now turned it into a *simple* network! Great job. Next, we will discuss an important property as to *why* thresholding using a quantile tends to be a very common tactic to obtaining simple networks from networks which are undirected and loopless. Remember that in the last section, we defined the network density for a simple network as:

$$density(A) = \frac{2\sum_{j>i} a_{ij}}{n(n-1)}.$$

Since we have thresholded at the 50^{th} percentile for the symmetric friendship network, this means that about 50 percent of the possible edges will exist (the *largest* 50 percent of edges), and 50 percent of the possible edges will not exist (the *smallest* 50 percent of edges). Remembering that the number of possible edges was $\frac{1}{2}n(n-1)$ for an undirected network,

this means that $\sum_{j>i} a_{ij}$ must be half of $\frac{1}{2}n(n-1)$, or $\frac{1}{4}n(n-1)$. Therefore:

$$\begin{aligned} density(A) &= \frac{2\sum_{j>i} a_{ij}}{n(n-1)}, \\ &= \frac{2 \cdot \frac{1}{4}n(n-1)}{n(n-1)}, \quad \sum_{j>i} a_{ij} = \frac{1}{4}n(n-1) \\ &= 0.5. \end{aligned}$$

So when we threshold the network at a quantile t, we end up with a network of density also equal to t! Let's confirm that this is the case for our symmetric friendship network:

```
from graspologic.utils import is_unweighted, is_loopless, is_symmetric

def simple_network_dens(X):
    # make sure the network is simple
    if (not is_unweighted(X)) or (not is_loopless(X)) or (not is_symmetric(X)):
        raise TypeError("Network is not simple!")
    # count the non-zero entries in the upper-right triangle
    # for a simple network X
    nnz = np.triu(X, k=1).sum()
    # number of nodes
    n = X.shape[0]
    # number of possible edges is 1/2 * n * (n-1)
    poss_edges = 0.5*n*(n-1)
    return nnz/poss_edges

print("Network Density: {:.3f}".format(simple_network_dens(A_friend_thresh)))
```

This is desirable for network machine learning because many network properties (such as the summary statistics we have discussed so far, and numerous other properties we will discuss in later chapters) can vary when the network density changes. This means that a network of a different density might have a higher clustering coefficient than a network of a lower density simply due to the fact that its density is higher (and therefore, there are more opportunities for closed triangles because each node has more connections). This means that when we threshold groups of networks and compare them, thresholding using a quantile will be very valuable.

Note that a common pitfall you might run into with thresholding (and the broader class of techniques known as *sparsification* approaches) that rely on quantiles occurs when a weighted network can only take non-negative edge-weights. This corresponds to a network with an adjacency matrix A where every a_{ij} is greater than or equal to 0. In this case, one must be careful to choose a threshold which is not zero. Let's consider a network were 60% of the entries are zeros, and 40% of the entries take a random value between 5 and 10:

```
from graspologic.simulations import er_nm

# 10 nodes
n = 10
# total number of edges is 40% of the number of possible edges
# 1/2 * n * (n-1)
m = int(0.4*0.5*n*(n-1))
A = er_nm(n, m, wt=np.random.uniform, wtargs=dict(low=5, high=10))
```

If we threshold at the 50^{th} percentile, since 60 percent of the edges do not exist, then the 50^{th} percentile is still just zero:

```
# use the quantile function to obtain the threshold
t = quantile(A[np.where(np.triu(A.shape, k=1))], q=0.5)  # quantile = percentile / 100
A_thresh = copy(A)  # copy the network over
```

(continues on next page)

(continued from previous page)

```
# threshold using t
A_thresh[A_thresh <= t] = 0
A_thresh[A_thresh > t] = 1
print("Threshold for 50th percentile: {:d}".format(int(t)))
```

And we don't actually end up with a network having a density of 0.5, but rather, the same as the fraction of non-zero edges in the original network (which was 40%, or 0.4):

```
dens = simple_network_dens(A_thresh)
```

So the take-home message is that we need to be careful that if we want to conclude that two percentile-thresholded networks have the same network density (equal to the percentile we thresholded at), that we have enough non-zero entries to threshold with across both (or all) of the networks.

Sparsification removes potentially spurious low-weight edges

The next simplest edge-weight regularization technique is called *sparsificiation*. Remember that our undirected friendship network looked like this:

Notice that for a *lot* of the off-diagonal entries, many of the values are really tiny compared to the maximum value in the network which is almost 1. What if the way we measured these edges was very sensitive to high values, but had trouble discerning whether a value was actually zero, or was just really small?

For this particular situation, we turn to *sparsification*. Through sparsification, we proceed very similar to thresholding like we did above. Remember that we chose a threshold, t, and first set all adjacency values less than or equal to t to zero. Now, we're done! We simply skip the step of setting values greater than t to one. Let's try an example where we take the friendship network, and sparsify the network using the 0.7 quantile. Note that this will lead to the smallest 70 percent of edges to take the value of zero, and the largest 30 percent of edges will keep their original edge-weights:

```
q=0.7  # the quantile to sparsify with

# use the quantile function to obtain the threshold
t = quantile(A_friend_avg_sym[upper_tri_non_diag_idx], q=q)  # quantile = percentile /
↪ 100
A_friend_sparse = copy(A_friend_avg_sym)  # copy the network over

# sparsify using t
A_friend_sparse[A_friend_sparse <= t] = 0
```

Notice that many of the small entries in the off-diagonal areas now have a value of zero. Again, we have the same pitfalls for sparsification as we did with thresholding, where if the network takes only non-negative edge weights and the percentile we choose corresponds to a threshold of zero, we might not actually end up changing anything.

Edge-weight normalization

With weighted networks, it is often the case that we might want to reshape the distributions of edge-weights in our networks to highlight particular properties. Notice that the edge-weights for our human networks take values between 0 and 1, but for our alien network take values between 0 and almost 40. How can we possibly compare between these two networks when the edge-weights take such different ranges of values? We turn to standardization, which allows us to place values from different networks on the same scale.

z-scoring standardizes edge weights using the normal distribution

The first approach to edge-weight standardization is known commonly as z-scoring. Suppose that A is the adjacency matrix, with entries a_{ij}. With a z-score, we will rescale the weights of the adjacency matrix, such that the new edge-weights (called z-scores) are approximately normally distributed. The reason this can be useful is that the normal distribution is pretty ubiquitous across many branches of science, and therefore, a z-score is relatively easy to communicate with other scientists. Further, many things that exist in nature can be well-approximated by a normal distribution, so it seems like a reasonable place to start to use a z-score for edge-weights, too! The z-score is defined as follows. We will construct the z-scored adjacency matrix Z, whose entries z_{ij} are the corresponding z-scores of the adjacency matrix's entries a_{ij}. For a weighted, loopless network, we use an estimate of the *mean*, $\hat{\mu}$, and the *unbiased* estimate of the *variance*, $\hat{\sigma}^2$), which can be computed as follows:

$$\hat{\mu} = \frac{1}{n}\sum_{i \neq j} a_{ij},$$

$$\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i \neq j}(a_{ij} - \hat{\mu})^2.$$

The z-score for the (i, j) entry is simply the quantity:

$$z_{ij} = \frac{a_{ij} - \hat{\mu}}{\hat{\sigma}}$$

Since our network is loopless, notice that these sums are for all *non-diagonal* entries where $i \neq j$. If the network were not loopless, we would include diagonal entries in the calculation, and instead would sum over all possible combinations of i and j. the interpretation of the z-score z_{ij} is the *number of stadard deviations* that the entry a_{ij} is from the mean, $\hat{\mu}$.

We will demonstrate on the directed friendship network. We can implement z-scoring as follows:

```
from scipy.stats import zscore

def z_score_loopless(X):
    if not is_loopless(X):
        raise TypeError("The network has loops!")
    # the entries of the adjacency matrix that are not on the diagonal
    non_diag_idx = where(~eye(X.shape[0], dtype=bool))
    Z = np.zeros(X.shape)
    Z[non_diag_idx] = zscore(X[non_diag_idx])
    return Z

ZA_friend = z_score_loopless(A_friend)
```

Next, we will look at the edge-weight histogram for the directed friendship network before and after z-scoring. Remember that the network is loopless, so again we exclude the diagonal entries:

```
from seaborn import histplot

non_diag_idx = where(~eye(A_friend.shape[0], dtype=bool))
fig, axs = plt.subplots(1,2, figsize=(15, 4))
ax = histplot(A_friend[non_diag_idx].flatten(), ax=axs[0], bins=9)
ax.set_xlabel("Edge Weight");
ax.set_ylabel("Number of Edges");
ax.set_title("Directed Friendship Network, Before Z-score");
ax = histplot(ZA_friend[non_diag_idx].flatten(), ax=axs[1], bins=9)
ax.set_xlabel("Z-score");
ax.set_ylabel("Number of Edges");
ax.set_title("Directed Friendship Network, After Z-score");
```

The theory for when, and why, to use z-scoring for network machine learning tends to go something like this: many things tend to be normally distributed with the same mean and variance, so perhaps that is a reasonable expectation for our network, too. Unfortunately, we find this often to *not* be the case. In fact, we often find that the specific distribution of edge weights itself often might be lamost infeasible to identify in a population of networks, and therefore *almost* irrelevant all-together. To this end, we turn to instead *ranking* the edges.

Ranking edges preserves ordinal relationships

The idea behind ranking is as follows. We don't really know much useful information as to how the distribution of edge weights varies between a given pair of networks. For this reason, we want to virtually eliminate the impact of that distribution *almost* entirely. However, we know that if one edge-weight is larger than another edge-weight, that we do in fact trust that relationship. What this means is that we want something which preserves *ordinal* relationships in our edge-weights, but ignores other properties of the edge-weights. An ordinal relationship just means that we have a natural ordering to the edge-weights. This means that we can identify a largest edge-weight, a smallest edge-weight, and every position in between. When we want to preserve ordinal relationships in our network, we do something called *passing the non-zero edge-weights to ranks*. We will often use the abbreviation `ptr` to define this function because it is so useful for weighted networks. We pass non-zero edge-weights to ranks as follows:

1. Identify all of the non-zero entries of the adjacency matrix A.
2. Count the number of non-zero entries of the adjacency matrix A, n_{nz}.
3. Rank all of the non-zero edges in the adjacency matrix A, where for a non-zero entry a_{ij}, $rank(a_{ij}) = 1$ if a_{ij} is the smallest non-zero edge-weight, and $rank(a_{ij}) = n_{nz}$ if a_{ij} is the largest edge-weight. Ties are settled by using the average rank of the two entries.
4. Report the weight of each non-zero entry (i, j) as $r_{ij} = \frac{rank(a_{ij})}{n_{nz}+1}$, and for eachh zero entry as $r_{ij} = 0$.

Below, we pass-to-ranks the directed friendship network using `graspologic`, showing both the adjacency matrix and the edge-weight distribution before and after `ptr`:

```
from graspologic.utils import pass_to_ranks

RA_friend = pass_to_ranks(A_friend)
```

The edge-weights for the adjacency matrix R after `ptr` has the interpretation that each entry r_{ij} which is non-zero is the *quantile* of that entry amongst *the other non-zero entries*. This is unique in that it is completely *distribution-free*, which means that we don't need to assume anything about the distribution of the edge-weights to have a reasonably interpretable quantity. On the other hand, the z-score had the interpretation of the number of standard deviations from the mean, which is only a sensible quantity to compare if we assume the population of edge-weights are normally distributed.

Another useful quantity related to pass-to-ranks is known as the zero-boosted pass-to-ranks. Zero-boosted pass-to-ranks is conducted as follows:

1. Identify all of the non-zero entries of the adjacency matrix A.
2. Count the number of non-zero entries of the adjacency matrix A, n_{nz}, *and* the number of zero-entries of the adjacency matrix A, n_z. Note that since the values of the adjacency matrix are either zero or non-zero, that $n_{nz} + n_z = n^2$, as A is an $n \times n$ matrix and therefore has n^2 total entries.
3. Rank all of the non-zero edges in the adjacency matrix A, where for a non-zero entry a_{ij}, $rank(a_{ij}) = 1$ if a_{ij} is the smallest non-zero edge-weight, and $rank(a_{ij}) = n_{nz}$ if a_{ij} is the largest edge-weight. Ties are settled by using the average rank of the two entries.
4. Report the weight of each non-zero entry (i, j) as $r'_{ij} = \frac{n_z+rank(a_{ij})}{n^2+1}$, and for each zero entry as $r'_{ij} = 0$.

The edge-weights for the adjacency matrix R' after zero-boosted `ptr` have the interpretation that each entry r'_{ij} is the quantile of that entry amongst *all* of the entries. Let's instead use zero-boosted `ptr` on our network:

```
RA_friend_zb = pass_to_ranks(A_friend, method="zero-boost")
```

Logging reduces magnitudinal differences between edges

When we look at the distribution of non-zero edge-weights for the activity/hobby network or the friendship network, we notice a strange pattern, known as a *right-skew*:

Notice that *most* of the edges have weights which are comparatively small, between 0 and 34, but some of the edges have weights which are much (much) larger. A **right-skew** exists when the majority of edge-weights are small, but some of the edge-weights take values which are much larger.

What if we want to make these large values more similar in relation to the smaller values, but we simultaneously want to preserve properties of the underlying distribution of the edge-weights? Well, we can't use `ptr`, because `ptr` will throw away all of the information about the edge-weight distribution other than the ordinal relationship between pairs of edges. To interpret what this means, we might think that there is a big difference between sharing no interests compared to three interests in common, but there is not as much of a difference in sharing ten interests compared to thirteen interests in common.

To do this, we instead turn to the logarithm function. The logarithm function $log_{10}(x)$ is defined for positive values x as the value c_x where $x = 10^{c_x}$. In this sense, it is the "number of powers of ten" to obtain the value x. You will notice that the logarithm function looks like this:

What is key to noice about this function is that, as x increases, the log of x increases by a *decreasing* amount. Let's imagine we have three values, $x = .001$, $y = .1$, and $z = 10$. A calculator will give you that $log_{10}(x) = -3, log_{10}(y) = -1$, and $log_{10}(z) = 1$. Even though y is only .099 units bigger than x, its logarithm $log_{10}(y)$ exceeds $log_{10}(x)$ by two units. on the other hand, z is 9.9 units bigger than y, but yet its logarithm $log_{10}(z)$ is still the same two units bigger than $log_{10}(y)$. This is because thhe logarithm is instead looking at the fact that z is *one* power of ten, y is -1 powers of ten, and z is -3 powers of ten. The logarithm has *collapsed* the huge size difference between z and the other two values x and y by using exponentiation with *base* ten.

In this sense, we can also use the logarithm function for our network to reduce the huge size difference between the values in our activity/hobby network. However, we must first add a slight twist: to do this properly and yield an interpretable adjacency matrix, we need to *augment* the entries of the adjacency matrix *if* it contains zeros. This is because the $log_{10}(0)$ is *not defined*. To augment the adjacency matrix, we will use the following strategy:

1. Identify the entries of A which take a value of zero.
2. Identify the smallest entry of A which is not-zero, and call it a_m.
3. Compute a value ϵ which is an *order of magnitude* smaller than a_m. Since we are taking powers of ten, a single order of magnitude would give us that $\epsilon = \frac{a_m}{10}$.
4. Take the augmented adjacency matrix A' to be defined with entries $a'_{ij} = a_{ij} + \epsilon$.

Next, since our matrix has values which are now all greater than zero, we can just take the logarithm:

```
def augment_zeros(X):
    if np.any(X < 0):
        raise TypeError("The logarithm is not defined for negative values!")
    am = np.min(X[np.where(X > 0)])  # the smallest non-zero entry of X
    eps = am/10  # epsilon is one order of magnitude smaller than the smallest non-
 ↪zero entry
    return X + eps  # augment all entries of X by epsilon

A_activity_aug = augment_zeros(A_activity)
# log-transform using base 10
A_activity_log = np.log10(A_activity_aug)
```

When we plot the augmented and log-transformed data, what we see is that many of the edge-weights we originally might have thought were zero if we only looked at a plot were, in actuality, *not* zero. In this sense, for non-negative weighted networks, log transforming after zero-augmentation is often very useful for visualization to get a sense of the magnitudinal differences that might be present between edges.

Our edge-weight histogram becomes:

CHAPTER

SEVEN

WHY USE STATISTICAL MODELS?

7.1 Why Use Statistical Models?

In network data science, we typically begin with a question of interest, and a network we use to answer that question. To answer this question, we will turn to statistical models. Consider the simplest possible statistical model which has been used for years: the coin flip model. Let's say we are playing a game with a gambler, and we bet one dollar. If the coin lands on heads, we get our dollar back, and an additional dollar. If the coin lands on tails, we lose our dollar. We get to watch ten coin flips before deciding whether or not to join the game. So, should we play?

A coin has a probability, which we will denote p, of landing on heads. Since the coin either lands on heads or tails, this means that the probability that the coin lands on tails is $1 - p$. Reasonably, we might guess that the probability that this coin lands on heads is just 0.5 (the coin has an equal chance of landing on heads and tails). Unfortunatetly, the universe is a nefarious place! We could easily construct coins which slightly favor heads (perhaps a coin with a chance of landing on heads of 0.51) *or* a coin which slightly favors tails (perhaps a coin with a chance of landing on tails of 0.51).

To understand whether or not we should play this game, we turn to *statistical modelling*. A **statistical model** is a set of assumptions as to how a random system operates. The statistical model delineates what we think comes down to random chance in our system. In our coin flip example, this means that we describe the coin flip using the *Bernoulli model*, which means that the coin lands on heads with probability p and tails with probability $1 - p$. The statistical model is the set of all possible coins which could be used for the game.

A **random variable** is an object whose outcome comes down to random chance. In our coin flip example, the coin flip itself *is* the random variable. The coin possesses a probability p of landing on heads and a probability $1 - p$ of landing on tails. To learn about the coin, we conduct an *experiment*. The experiment here is watching the coin flip ten times, and observing the outcome of the flips. The outcome of the coin flip is called a *realization* of the random variable. We will never truly understand the coin flip perfectly (we can never say for sure whether the coin will definitely land on heads or tails unless it has two heads or two tails). If we observe enough realizations of coin flips, however, we might be able to describe the coin flip in a way which could work this game in our favor.

The coin flip example is obviously very trivial, but it extends directly to statistical network models which we will use for simple networks. Consider a network of 100 students, in which each of the students attends one of two schools. We are interested in understanding whether students are more likely to be friends on a social networking site with students in their school than with students from the other school. Unlike traditional machine learning, in network machine learning we do not observe n outcomes (or realizations) with d dimensions. Rather, we see an adjacency matrix A, whose nodes are the 100 students, and whose edges are the entries a_{ij} which take a value of one if the two students i and j are friends on the social networking site and a value of zero if the two students are not friends on the social networking site.

Like the coin flip example, there is randomness and uncertainty to our social network. Perhaps a pair of students might be friends in real life, but they never got around to adding each other on the social media site. Maybe our students had a fight and are no longer friends, but never bothered to delete one another as friends on the site. Other factors might exist that we don't know about (sports, hobbies, special interests) that influence whether two people are friends. Our social network might not capture all of the students, and we might be missing a large portion of the community all together. In

many additional ways, our social network is noisy, and in order to address our question of interest, we need procedures which account for this uncertainty.

In machine learning, we typically encounter situations in which we have n observations in d dimensions. Traditional statistical models include univariate statistical models (models for data with 1 dimension) and multivariate statistical models (models for data with $d > 1$ dimensions), which can capture this traditional data representation. These models are well suited for discovering new insights about individual observations or collections of individual observations. Why do we need special statistical models for networks? Our realizations are not n disparate observations in d-dimensions; a realization in network machine learning **is the full network itself**, consisting of nodes, edges, and potential network attributes. We seek to model a representation of the *entire* network so that we can convey insights about properties of the network. To address our question of interest above, we need to characterize how students relate to other students in the network, not describe individual stdents. To this end, we describe our random network using sets of statistical assumptions, referred to as the **statistical network model**. The coin flip model might haave felt really simple, but we will see how we can use collections of coins to describe pretty complicated random networks throughout this chapter. We break down the key aspects of the coin flip experiment because it is so crucial:

The Coin Flip Experiment

We had the following items we were concerned with in the coin flip example:

1. The outcomes: The outcomes are either heads or tails. These outcomes will be denoted by the letter x, which takes values which are H (Heads) or T (Tails). The value x is called a **realization**.
2. The coin which was used: The specific coin being used in the coin flip experiment has a probability p of landing on heads and a probability $1 - p$ of landing on tails. We will denote the specific coin being used by the letter **x**. The bold-faced font means that the coin being used has a random outcome (it might be heads, it might be tails) to differentiate it from the coin flips which we saw and have known outcomes indicated by x (which has a fixed value, since we flipped the coin and *realized* the outcome). We don't know anything about p just yet, so we can't describe the coin's specific random behavior just yet. The value **x** is called the **random variable**.
3. Feasible coins: A possible coin that could have been used in the coin flipping experiment is one which has a probability q (which might be different from p) of landing on heads, and $1 - q$ of landing on tails. A feasible coin will be denoted by $Bern(q)$, which just means that the coin lands on heads with probability q and tails with probability $1 - q$.
4. The Bernoulli model: The collection of all feasible coins which could have been used. This is described by the set $\{Bern(q) : q$

7.1.1 Models aren't Right. Why do we Care?

It is important to clarify that we must pay careful attention to the age old aphorism attributed to George Box, a pioneering British statistician of the 20^{th} century. George Box stated, "all models are wrong, but some are useful." In this sense, it is important to remember that the statistical model we select is, in practice, *never* the correct model (this holds for any aspect of machine learning, not just network machine learning). In the context of network science, this means that even if we have a model we think describes our network very well, it is *not* the case that the model we select actually describes the network precisely and correctly. Despite this, it is often valuable to use statistical models for the simple reason that assuming that a stochastic process (that is, some *random* process) which governs our data is what allows us to convey *uncertainty*. To understand the importance of leveraging uncertainty, consider the following scenarios:

1. Lack of information: In practice, we would never have all of the information about the system that produced the network we observe, and uncertainty can be used in place of that information. For instance, in our social network example, we might only know which school that people are from, but there are many other attributes that would impact the friend circle of a given student. We might not know things like which classes people have taken nor which grade they're in, but we would expect these facts to impact whether a given pair of people might have a

higher chance of being friends. We can use uncertainty in our model to capture the fact that we don't know the classes nor grades of the students.

2. We might think the network is deterministic, rather than stochastic: In the extreme case, we might think that if we had *all* of the information which governs the network, then we could determine exactly what realizations would look like with perfect accuracy. Even if we knew exactly what realizations of the network might look like, this description, too, isn't likely to be very valuable. If we were to develop a model on the basis of everything, our model would be extremely complex and require a large amount of data. For instance, in our social network example, to know whether two people were friends with perfect accuracy, we might need to have intimate knowledge of every single person's life (Did they just have a fight with somebody and de-connect with that person? Did they just go to a school dance and meet someone new?).
3. We learn from uncertainty and simplicity: When we do statistical inference, it is rarely the case that we prioritize a complex, convoluted model that mirrors our data suspiciously closely. Instead, we are usually interested in knowing how faithfully a simpler, more generally applicable model might describe the network. This relates directly to the concept of the bias-variance tradeoff from machine learning, in which we prefer a model which isn't too specific (lower bias) but still describes the system effectively (lower variance).

Therefore, it is crucial to incorporate randomness and uncertainty to understand network data. In practice, we select a model which is appropriate from a family of candidate models on the basis of three primary factors:

1. Utility: The model of interest possesses the level of refinement or complexity that we need to answer our scientific question of interest. What this means is that the coin flip model will allow us to determine whether or not we should gamble.
2. Estimation: The data has the level of breadth to facilitate estimation of the parameters of the model of interest. This means that we can use the outcomes of coin flips to guess what the probability that the coin will land on heads is.
3. Appropriateness: The model is appropriate for the data we are given. This means that there are not major factors which are unaccounted for by the statistical model, such as if the coin thrower holds a magnet which will alter the outcome of the coin flip.

For the rest of this section, we will develop intuition for the first point. Later sections will cover estimation of parameters and model selection.

7.2 Erdös-Rényi (ER) Random Networks

We will start our description with the simplest random network model. Consider a social network, with 50 students. Our network will have 50 nodes, where each node represents a single student in the network. Edges in the social network represent whether or not a pair of students are friends. What is the simplest way we could describe whether two people are friends?

In this case, the simplest possible thing to do would be to say, for any two students in our network, there is some probability (which we will call p) that describes how likely they are to be friends. In the below example, for the sake of argument, we will let $p = 0.3$. What does a realization from this network look like?