

Chapter 4

Primer

Network data calls for—and is analyzed with—many computational and mathematical tools. One needs good working knowledge in programming, including *data structures and algorithms* to effectively analyze networks. In addition to graph theory (Ch. 1), *probability theory* is the foundation for any statistical modeling and data analysis. *Linear algebra* provides another foundation for network analysis and modeling because matrices are often the most natural way to represent graphs. Although this book assumes that readers are familiar with the basics of these topics, here we review the computational and mathematical concepts and notation that will be used throughout the book. You can use this chapter as a starting point for catching up on the basics, or as reference while delving into the book.

4.1 Coding and computation

Studying network data requires creating and using computer code. We assume the reader is familiar with writing code using basic programming concepts, including loops, conditional statements, and functions, as well as basic data structures and algorithms. Here we provide some additional background and notation.

i For vs. foreach.

One programming concept worth discussing is the *foreach loop*. A traditional for loop uses a numeric index variable that increments between a start and end value in constant steps. Here is a for loop written with a C-style code:

```
for (int i=0; i<10; i++){  
    [...]  
}
```

The loop (or index) variable *i* begins at 0 and increases by 1 while it is less than 10.

By comparison, a foreach loop generalizes a for loop by defining a sequence of values, not necessarily a range of numbers, that are iterated over. For example:

```
A = ['a', 'b', 'c']
foreach x in A:
    [...]
done
```

Here *A* is a list of characters, and our variable *x* will be equal to 'a', then 'b', then 'c', as the loop progresses. In practice, *foreach* loops are very useful, as we can focus on what we are looping over and not have to deal with maintaining and using index variables.

The concepts are simple but confusion can set in when syntax is succinct: often a *foreach* loop is clear from the context and only “for” is written. Indeed, some programming languages, chiefly Python, use *foreach* loops exclusively. When first using such a language, think of the *for* as really being “foreach” when reading or writing code.

Working with files

Working with data computationally necessarily requires working with data files.¹ We assume familiarity with basic file read and write operations in your programming language of choice. Usually files are “opened” for reading and writing and a file “handle” is created that provides access to the file from your code. In Python, for instance, a file can be opened for writing with `fout = open("filename", 'w')` and we can use the *fout handle* to send data to the file:

```
values = ['a', 'b', 'c']
for s in values:
    fout.write(s)
    fout.write("\n")
fout.close()
```

In the example, we explicitly add newline characters (“\n”) after writing each string *s* and we “closed” the file handle when finished. Python, and most modern languages, provide great functionality to handle such bookkeeping automatically.

Working with files also requires specifying the locations of those files. For that, *paths* are used, which specify where the file is located within your computer’s file structure by describing the sequence of folders (directories) and sub-folders we would follow to find the file. Paths can be *global* (or *absolute*), specified from the very beginning of your file system, or *relative*, specified in relation to your program’s *working directory*. Managing a working directory can be a chore for beginners, but by allowing your code to use relative paths, your code will be portable, and work without being changed on other computers with different file structures.



Debugging tip: find out how to print your code’s working directory and add this to the top of your script if you are struggling with file errors. In Python:

```
import os
print(os.getcwd())
```

¹ And databases too!

and in R:

```
print(getwd())
```

Algorithms and pseudocode

An *algorithm* is a sequence of instructions to solve a problem or perform a computational task. Many network calculations boil down to a combination of algorithms. For instance, the problem of calculating the shortest paths between two nodes in a network can be achieved by the *breadth-first search* (BFS) algorithm or *Dijkstra algorithm*.

To describe algorithms, we use *pseudocode*, which is a simplified notation of instructions that resemble spoken language.² Readers who are familiar with programming in any common programming language should be able to read pseudocode, regardless of what language they know. Algorithm 4.1 is pseudocode of the BFS algorithm that returns all nodes that can be reached along paths starting from node s .

Algorithm 4.1 Reachable nodes with breadth-first search (BFS). With a few modifications, this algorithm can also compute the distances (or shortest path lengths) between nodes in G .

```

1: Input: Graph  $G = (V, E)$ , starting node  $s$ 
2: Define  $Q$  as a new queue with  $s$  as its only element
3: Define visited as a new set with  $s$  as its only element
4: while  $Q$  is not empty do
5:   current =  $Q$ .dequeue()           ▷ Get a node to visit
6:   for  $u \in \text{neighbor}(\text{current})$  do   ▷ Get the neighbors
7:     if  $u \notin \text{visited}$  then
8:       visited.add( $u$ )
9:        $Q$ .enqueue( $u$ )
10: Return visited

```

Notice that BFS as written in Alg. 4.1 depends on a *queue*, a data structure that makes it easy and fast to retrieve items (in this case nodes) in a particular order—algorithms and data structures go hand-in-hand. An alternative to BFS is called DFS—Depth-First Search. Based on the names and Alg. 4.1, can you judge what makes them different?

Computational complexity

Often there are multiple ways to solve a computational problem and the solutions (algorithms) can wildly vary in terms of their efficiency, which is concretely defined and studied as *computational complexity*. For instance, imagine that we are sorting an array of numbers. Among numerous sorting algorithms, one particularly interesting—and incredibly simple—algorithm is called “Bogosort,” which can be expressed with just two lines of pseudocode (see Alg. 4.2).

² There is an old joke saying that “Python is executable pseudocode” given its syntax resembling pseudocode and natural language.

Algorithm 4.2 Bogosort

```

while not sorted(deck) do
    shuffle(deck)

```

If the shuffle³ is random and all elements are distinct, given n total elements, the expected number of comparisons and swaps that should be performed (in the average case) roughly scales with $n!$. In other words, if we have an array with just one million items, we are looking at about $10^{5,565,709}$ operations. Even if a supercomputer can perform 10^{20} operations per second, we'll need to wait. . . for some time.

Yes, this is indeed a ridiculous example. But, if you are not equipped with at least a basic understanding of computational complexity, you *will* accidentally implement your own bogo-algorithms. Pretty much everyone who codes had the experience of waiting for a program to finish, realizing only later that it will not finish within their lifetime (or until the end of our solar system). The difference between an efficient algorithm and an inefficient one can determine—not just *how long* we should wait—but whether the computation is even *possible*.

Can't we just use faster programming languages or more powerful computers? This is a valid point but only to some extent. In most cases, the *fundamental* efficiency of the algorithm easily tops the power of the hardware and programming language.

Time complexity and space complexity The two major components of computational complexity are *time complexity* and *space complexity*. Time complexity is about how the amount of operations increases (scales) with the size of problem (e.g., the length of an array in a sorting problem or the number of nodes and edges in a network calculation). In other words, the question is: would we do 10 times more computation if our array becomes 10 times longer? Would we do 100 times more computation? Space complexity is similar, but about how the amount of memory scales with the size of the problem. There are often—not always—tradeoffs between these two; we can sometimes speed up a computation by putting more data into memory or save space by doing more computation.

Big-O Since an algorithm's performance matters most when the size of input is large, we usually focus on the limiting behavior of algorithms. (This also abstracts away pesky implementation details like how fast the CPU is and how efficient the programming language is.) These are conceptualized based on the “Big-O notation” that captures the limiting behavior of mathematical functions. As $x \rightarrow \infty$,

$$f(x) = O(g(x)) \quad (4.1)$$

if there exists a positive real number M and a real number x_0 such that

$$|f(x)| \leq M g(x), \text{ for all } x \geq x_0. \quad (4.2)$$

³ By “shuffle” we mean to randomly permute all the elements of a sequence.

For instance, if the time complexity of an algorithm, given the size of input n , can be written as $T(n) = 2n^2 + 5n + 100$, we say $T(n) = O(n^2)$. Because, for any $n > 1$,

$$2n^2 + 5n + 100 < 2n^2 + 5n^2 + 100n^2 = 107n^2, \quad (4.3)$$

where $M = 107$ and $g(n) = n^2$.

In most cases, the big-O complexity can be calculated by simply keeping only the fastest-growing term in the full complexity expression. Note that big-O notation specifies an upper bound, not a precise scaling relationship. In other words, if $f(n) = O(n)$, then we could also write $f(n) = O(n^2)$. However, in the context of algorithm analysis, big-O notation usually denotes the tighter bounds.

Another crucial consideration is that algorithms' performance can vary immensely based on the input. Even a terrible sorting algorithm may work very well if the input is already almost sorted. Therefore, it is critical to consider multiple scenarios, especially worst cases. It is customary to report both the average and worst case complexity when they differ. For instance, the worst-case time complexity of the Quicksort algorithm is $O(n^2)$ although its average-case time complexity is $O(n \log n)$.

A sneak peek into the complexity zoo $O(1)$ refers to the case where the algorithm does not depend on the size of input data at all. Whether it's $n = 10$ items or $n = 10,000,000$, an $O(1)$ algorithm returns its result within a constant time that does not scale with n . For instance, the operation of obtaining the size of an array or a set is usually $O(1)$ because the data structure usually keeps track of the number of items.

$O(N)$ algorithms tend to be those that traverse the data at least once. For instance, if we want to identify the maximum value of an unsorted array, we need to scan the entire array and examine every element at least once. $O(N^2)$ algorithms tend to require repeat traversals. For instance, in terms of time complexity, bubble sort is a well-known example of an $O(N^2)$ algorithm. Given N elements, we go through every item but the last one and compare that item to every subsequent item in the array; the operations scale as N^2 . $O(\log N)$ algorithms tend to appear when the data are "nicely organized" and we can eliminate major portions of the data at every step of the computation. For instance, if we have an already sorted array, finding the location of a target number within the array is easy: we can recursively bisect the array. If the value at the middle of the array is, say, larger than the target value, then we can safely ignore the second half of the array because all those values will be larger than the target value. This bisection reduces the computation drastically and we need only the order of $O(\log N)$ computation. Having a sublinear algorithm for a problem is fantastic!⁴

Data structures

Algorithms (and their complexity) are tightly coupled with *data structures*—the ways to computationally organize and manipulate collections of data. For instance, sorting algorithms like *heap sort* depend on a clever binary-tree data structure called a *heap* that allows us to access the smallest (or the largest) item in the collection in constant time.

⁴ On the other hand, an exponential or even *nondeterministic polynomial* (NP) algorithm is very challenging. This is the famous P vs. NP question.

Let's describe—mathematically and as “computational objects”—some of the common data structures that we'll rely on throughout this book.

Arrays and lists

In the abstract sense, an *array* or a *list* is a sequence of items, which do not have to be unique, usually denoted with square brackets: $[a, b, \dots]$. An *array* usually refers to the most primitive type of lists that stores the same type of data in a consecutive block of memory. An array can be indexed; we can obtain the first item or 100th item (e.g., $a[0]$ and $a[100]$) in $O(1)$. The size of an array is usually fixed when we allocate it. It is not possible to quickly find out whether an item is in the array or not. To do so, either we should scan the whole array (unsorted) or otherwise do a search (e.g., a binary search on a sorted array).

Lists usually refer to a data structure that can be dynamically lengthened, shrunk, or modified. A *linked list* is a data structure constructed by creating a chain of items that each point to the next (and/or previous) item in the list. We can traverse the linked list sequentially by following these *pointers*. Items can be easily added or removed from the list by modifying these pointers. However, a linked list does not allow us to access an item by its index.

The number of items in a list is its *length*. Lists can be homogeneous (every item in the list is the same type) or inhomogeneous, containing multiple types of items. A list can even contain elements that are themselves lists, making a *list-of-lists*.

Tuples, which are usually denoted with parentheses: (x, y, z) , are similar to lists. Like a list, a tuple is still a sequentially ordered list of items. Usually the distinction is whether we can change (list) or not (tuple) the data. Some computer languages implement list and tuple data structures slightly differently; lists in Python, for instance, are mutable, while tuples are immutable.

i A *mutable* data structure (or variable) is one whose contents can change after it has been created while an *immutable* data structure cannot be changed once it is created. Using an immutable data structure guarantees that it stays the same once initialized. We can pass that data structure to a function and be sure the function won't have any side effects on the data. While immutable data limits the computational tasks and algorithms that can be pursued, it is safer and, when changes aren't needed, good practice to make data immutable.

Dictionaries

The next data structure worth considering, and one that is used throughout most computer code that works with networks, is the *dictionary*. Dictionaries or “dicts,” also called associative arrays, maps, or hashes,⁵ allow us to efficiently map *keys* to *values*.

Dictionaries typically require the keys to be unique, distinct from one another. The values need not be unique. The cardinality, size, or length of a dictionary is the number of keys it contains. Dictionaries are denoted with curly braces: $\{k_1 : v_1, k_2 : v_2, \dots\}$ by

⁵ You can tell something is important when it has a lot of names!

writing each element of the dictionary in the form *key:value*, using a colon (:) to denote the mapping from key to value. You may on occasion see a dictionary's key–value pair written equivalently as a tuple: (key, value).

Dictionaries are especially powerful data structures. We can efficiently determine whether a key is a member of a dictionary. (Usually this requires immutable keys.) This means we can efficiently retrieve values when we have a key, giving us a lookup table that works at (approximately) the same speed regardless of how long it is. A use case with networks is defining a dictionary where the nodes in the network are the keys and the values are the sets of neighbors (Ch. 8). Then we can quickly retrieve any node's neighbors regardless of how big the network becomes.

Sets

Once we have a dictionary, we can also implement a *set* (Sec. 4.2.1), which can be considered (and sometimes implemented as) a dictionary with the item as the key.⁶ Unlike a list where $[a, b] \neq [b, a]$, a set is unordered, $\{a, b\} = \{b, a\}$, and its elements are unique. The usual implementations of sets guarantee low time complexity ($O(1)$ or $O(\log N)$) operations for item lookup (by using a hashtable, sorted list, or tree-based data structure) and addition and deletion of items. But it does not allow indexing.

Sets are usually mutable as we can insert or remove elements from them but implementations of sets typically forbid using mutable objects as elements because data structures work by identifying each element based on its contents and, if those contents change, the program will lose track of the element. Not only does forbidding mutable set elements avoid these errors, it allows for data structures to work very efficiently, meaning that we can tell if a given item is a member of a set without looking at every element in the set first (i.e., fast item lookup). Regardless of how big the set is, testing for membership requires the same amount of work.⁷ This efficient test for set membership is very powerful, and complicated (network) algorithms can often be expressed very easily with set data structures.

4.2 Mathematics

Here we describe some notation we'll use throughout the book and discuss various concepts from linear algebra and probability, both of which are central to studies of network data.

4.2.1 Sets and other notation

A *set* is an unordered collection of unique elements, which can be any mathematical objects, including other sets.⁸ A set that contains no elements is called the *empty set*,

⁶ The value can be `true`, 1, or anything that indicates the existence; the value does not matter much when we use a dictionary as a set.

⁷ Strictly speaking this is only approximately true. Sometimes, depending on what algorithm is used to implement the set, there may be more computations needed for bigger sets than smaller sets, but it is usually less than having to look at every element.

⁸ As a mathematical object, this is different than the set data structure we just discussed. Some set data structures, depending on how they are implemented, cannot support sets as members of sets.

denoted \emptyset or sometimes $\{\}$. The number of elements in a set is the size or *cardinality* of the set; for a set s , its size is denoted $|s|$. We denote a set when listing out its elements using curly braces: $\{a, b, c\}$ or $\{a_i\}_i^n := \{a_1, a_2, \dots, a_n\}$. An element x that belongs to a set A is denoted by $x \in A$; likewise, x not in A is denoted $x \notin A$. The union $A \cup B$ of two sets A and B is the set of elements that appear in either or both of the sets. Conversely, the intersection $A \cap B$ is the set of elements that appear in both A and B . The difference $A - B$ is the set of elements in A that are not in B , and likewise, $B - A$ is the set of elements in B that are not in A . One set can be a *subset* of another set: $A \subset B$ means that every element in A is also in B . Conversely, $A \supset B$ means that A is a *superset* of B : every element in B is also in A . A is a *proper subset* of B if and only if A is a subset of B and B contains at least one element not in A , that is, $B - A \neq \emptyset$.

Some sets of *numbers* are common enough that standard symbols are used to denote them. These include the set of integers \mathbb{Z} , the set of real numbers \mathbb{R} , and the set of complex numbers \mathbb{C} . (Note that these sets are infinite and each is a superset of the one that came before: $\mathbb{Z} \subset \mathbb{R} \subset \mathbb{C}$.) Another convention is to use superscripts to denote positive and negative numbers: \mathbb{R}^+ represents the positive real numbers, for example.

How can we express sets that have complex definitions? For example, the set of all integers between 0 and 100 that are divisible by 4? We can write this out element-by-element, of course, but the following notation is common:

$$\{a \mid a \in \mathbb{Z}, 0 \leq a \leq 100, a = 0 \bmod 4\}. \quad (4.4)$$

You can read this kind of expression as “the set of all a such that [these conditions hold]”. In this case, we’d read Eq. (4.4) as “the set of all values of a such that” (“|”) “ a is an integer” ($a \in \mathbb{Z}$), “ a is between 0 and 100 inclusive” ($0 \leq a \leq 100$) and “ a is divisible by 4” ($a = 0 \bmod 4$). This notation is called *set-builder notation* or just *set notation*.

On occasion, we will distinguish identities or definitions from statements of equality by using “:=” instead of “=” (“ \equiv ” is another popular alternative).

4.2.2 Linear algebra

Linear algebra provides a powerful language to organize, represent, and manipulate collections of numbers and variables. As network data are all about collections of nodes and edges, it is natural to use the language of linear algebra.

Basic definitions A *scalar* x is an individual number. A d -dimensional *vector* \mathbf{x} is a collection of scalars x_i , $i = 1, \dots, d$, where x_i is the i th *element* of \mathbf{x} . (We use boldface variables to distinguish vectors and matrices from scalars.) Vectors of numbers of the same length can be added, subtracted, and multiplied, with a variety of definitions for scalar and vector multiplication. For instance, the *dot product* between two vectors is the sum of the products of their elements: $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^d a_i b_i$.

A matrix \mathbf{X} of size $n \times m$ is a collection of scalars arranged into an array or grid of n rows and m columns. The elements of a matrix \mathbf{X} , typically written⁹ X_{ij} and

⁹ It is sometimes common to use an uppercase letter for the matrix and the corresponding lowercase letter for an element of that matrix. We dispense with that formality.

identified by a pair of indices, represent the row and column of the element: X_{ij} is the element located in row i and column j . A *square matrix* is one where $n = m$; otherwise, the matrix is *rectangular*. Vectors can also be distinguished as row vectors or column vectors. A *row vector* is then a $1 \times d$ matrix and a *column vector* is a $d \times 1$ matrix. Given row and column vectors, a matrix of size $n \times m$ can also be considered as a collection of row vectors of length m arranged in n rows or a collection of column vectors of length n arranged in m columns. Sometimes we need to *transpose* a matrix: \mathbf{A}^\top is the transpose of \mathbf{A} , formed by swapping rows and columns, or $A_{ij}^\top = A_{ji}$. Transposing a column vector will create a row vector, and vice versa, which we can use to represent a dot product: $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^\top \mathbf{b}$ using matrix multiplication (described shortly).

Matrices of numbers are endowed with a variety of mathematical operations. Two matrices \mathbf{A} and \mathbf{B} can be added or subtracted by adding or subtracting their elements element-wise: $[\mathbf{A} \pm \mathbf{B}]_{ij} = A_{ij} \pm B_{ij}$. This requires \mathbf{A} and \mathbf{B} to be the same size (have the same numbers of rows and columns). A matrix can be multiplied by a scalar along the same lines: $[c\mathbf{A}]_{ij} = cA_{ij}$.

Lastly and most importantly is matrix multiplication. A matrix $\mathbf{C} = \mathbf{AB}$ is defined as a collection of dot products between the rows of \mathbf{A} and the columns of \mathbf{B} : the element $C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$ of \mathbf{C} is the dot product between the i th row of \mathbf{A} and the j th column of \mathbf{B} . This definition has a variety of consequences and is fundamental to all areas of science, engineering, and mathematics. It requires the matrices to be compatible sizes; if \mathbf{A} has n columns, \mathbf{B} should have n rows.

Eigenvalues and eigenvectors For an $n \times n$ square matrix \mathbf{A} , when an $n \times 1$ vector \mathbf{v} and scalar λ satisfies the following equation:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v},$$

we call \mathbf{v} an *eigenvector* of \mathbf{A} and λ an associated *eigenvalue*. When we consider \mathbf{A} as a linear transformation, its eigenvectors are the vectors that do not change direction under this transformation. Eigenvectors and eigenvalues play a role when diagonalizing a matrix. If a square matrix \mathbf{A} is diagonalizable, we can write it as $\mathbf{A} = \mathbf{VDV}^{-1}$, where the columns of \mathbf{V} are the eigenvectors of \mathbf{A} and \mathbf{D} is a diagonal matrix with the eigenvalues of \mathbf{A} on the diagonal ($D_{ii} = \lambda_i$). Often, but not always, we will work with real, symmetric ($\mathbf{A}^\top = \mathbf{A}$) matrices, which is especially convenient because, by the spectral theorem, they can always be diagonalized by an orthonormal basis of real eigenvectors. The eigendecomposition is one of several important *matrix factorizations*; singular value decomposition (SVD), for rectangular matrices, is another we will encounter. As we will see, matrix algebra and spectral analysis play pivotal roles in many problems in network analysis.

4.2.3 Probability

Probability is the fundamental mathematical tool of statistics and data science. Let us introduce the basic probability concepts and notation.

Conceptually, *probability* captures the chances for a random event to occur. An *event* A is when a *random variable* takes on a certain value from its *sample space* Ω ,

defined as the set of all possible outcomes for the event. For instance, A can be the event where a coin is flipped and takes on the value heads, represented as $A = H$. The sample space of A is $\Omega_A = \{H, T\}$. A single event can encompass multiple acts; event B could be the results from tossing a coin twice in a row, which has a sample space of $\{HH, HT, TH, TT\}$. Two or more events are said to be *mutually exclusive* (or *disjoint*) if at most one of them can occur. We say the events are *collectively exhaustive* if at least one of the events must occur.

An event A is said to occur with *probability* $\Pr(A)$ (sometimes written $P(A)$). This $\Pr(A)$, defined for the sample space of A , is a *probability distribution function* if it satisfies:

1. $\Pr(A) \geq 0$ for every A ,
2. $\Pr(\Omega_A) = 1$,
3. $\Pr(\cup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} \Pr(A_i)$ for mutually exclusive A_1, A_2, \dots

These axioms of probability—non-negativity, unitarity, and additivity—are the foundation for other properties of probability, such as Bayes' theorem, described below.

The *joint probability* for multiple events A, B, \dots all occurring is denoted by $\Pr(A, B, \dots)$. For instance, imagine throwing two dice A and B . Then, $\Pr(A = 1, B = 1)$ is the joint probability that both dice land on 1. The joint probability is the product of the individual probabilities if and only if the random variables are *independent* from each other:

$$\Pr(A, B) = \Pr(A) \Pr(B) \iff A \text{ and } B \text{ are statistically independent.} \quad (4.5)$$

As a notation, the joint distribution is symmetric, meaning $\Pr(A, B) = \Pr(B, A)$.

The *conditional probability* $\Pr(A|B)$ ¹⁰ is the probability to observe A given that we observe B . For instance, the probability of rain during the next hour $\Pr(\text{rain})$ can depend heavily on whether there are dark clouds or not. In other words, $\Pr(\text{rain at 4pm} \mid \text{clouds}) > \Pr(\text{rain at 4pm} \mid \text{no clouds})$. Note that the conditional probability does not have to describe a temporal or causal relationship. For instance, it is totally valid to think about $\Pr(\text{clouds at 2pm} \mid \text{rain at 4pm})$ (given a future event, we can look back on past events) or $\Pr(\text{wearing rain boots} \mid \text{carrying an umbrella})$.

The conditional probability and the joint probability are related. If we ask for the probability of both events A and B occurring, it is the same as asking (i) if event B occurs *and* (ii) given B occurred, if event A occurs. Formally, we write this relationship¹¹ as

$$\Pr(A, B) = \Pr(A \mid B) \Pr(B) \quad (4.6)$$

or

$$\Pr(A \mid B) = \frac{\Pr(A, B)}{\Pr(B)}. \quad (4.7)$$

This is called the *chain rule* (or *general product rule*) of probability. This rule plays an essential role in probability theory by allowing us to reduce any joint probability distribution into a product of conditional probabilities.

¹⁰ It can be read as “the probability of A given B .”

¹¹ If A and B are independent, then $\Pr(A \mid B) = \Pr(A)$, which again shows that $\Pr(A, B) = \Pr(A) \Pr(B)$.

What if there are more than two variables? We can iteratively apply the chain rule (Eq. (4.6)) by considering multiple variables as one. For instance, here with three variables, we first group A and B , then apply the chain rule again:

$$\begin{aligned}\Pr(A, B, C) &= \Pr(C \mid A, B) \Pr(A, B) \\ &= \Pr(C \mid A, B) \Pr(B \mid A) \Pr(A).\end{aligned}\quad (4.8)$$

More generally,

$$\Pr(A_n, \dots, A_1) = \Pr(A_n \mid A_{n-1}, \dots, A_1) \cdots \Pr(A_3 \mid A_2, A_1) \Pr(A_2 \mid A_1) \Pr(A_1). \quad (4.9)$$

When we consider multiple events, the *marginal probability* of a variable is simply the probability of individual events ($\Pr(A)$, $\Pr(B)$, etc.). Using the conditional probability, when we consider two variables, the marginal probability can be written as:

$$\Pr(A) = \sum_{B \in \Omega_B} \Pr(A, B) = \sum_{B \in \Omega_B} \Pr(A \mid B) \Pr(B), \quad (4.10)$$

which states that we need to consider all possible cases of the *other* variable to calculate the marginal probability. This process of summing/integrating out the other variable or variables (it holds for more than two variables) is called *marginalization*.

Finally, combining the symmetry of the joint distribution with the relationship between joint and conditional distributions reveals *Bayes' theorem*:

$$\begin{aligned}\Pr(A, B) &= \Pr(B, A) \\ \Pr(A \mid B) \Pr(B) &= \Pr(B \mid A) \Pr(A) \\ \Pr(A \mid B) &= \frac{\Pr(B \mid A) \Pr(A)}{\Pr(B)}.\end{aligned}\quad (4.11)$$

Equation (4.11), despite its simplicity, is one of humanity's most fundamental mathematical discoveries. It is also connected to many cognitive biases that plague human quantitative reasoning as many of those biases reflect misunderstandings of conditional probabilities.¹² Because Bayes' theorem is a fundamental way to approach *inference problems* and *learning from data* in general, we will be seeing this formula throughout this book.

4.2.4 Random variables and probability distributions

Random variables A *random variable* (RV) X is a variable endowed with a corresponding probability distribution $\Pr(X)$ governing the probability that X randomly takes on one of its *supported values*, denoted x . An event would be the assignment of x to X which would then occur with probability $\Pr(X = x)$. Given a probability distribution $f(x)$, often we use $X \sim f(x)$ to denote that X is *drawn from* the distribution $f(x)$ or equivalently that X is *distributed according to* $f(x)$.

¹² Bayes' theorem tells us we can't in general just "flip around" a conditional probability, $P(A \mid B) \neq P(B \mid A)$, something many intuitively wish to do.

For example, a coin flip can be represented by a random variable X that takes on value $x = 0$ when the coin lands tails and $x = 1$ when the coin lands heads. The probability distribution for this X can then be defined as $\Pr(X = 1) = p$, $\Pr(X = 0) = 1 - p$, where $p \in [0, 1]$ is a constant ($p = 1/2$ for an unbiased or fair coin).

Often distributions have *parameters* associated with them, non-random quantities that govern the scale or spread of the RV's values. For the coin flip, p served as a parameter. When discussing parameters generically, we commonly use the symbol θ to represent one or more generic parameters and when defining a function we distinguish between the values of the parameter(s) and the value of the RV with a semicolon: $f(x; \theta)$. (When describing an RV's probability it is also common to use a conditional probability, $\Pr(X | \theta)$, to distinguish between values and parameters.)

Lastly, when dealing with multiple variables, we distinguish whether they follow the same or different distributions. Suppose we independently flip a coin ($X \in \{0, 1\}$) and roll a die ($Y \in \{1 \dots 6\}$). Because they are independent, the joint probability of getting $X = x$ and $Y = y$ is given by

$$\Pr(X = x, Y = y) = f_{\text{coin}}(X = x)f_{\text{die}}(Y = y), \quad (4.12)$$

where we use f_{coin} and f_{die} to distinguish the different distributions the RVs follow. We usually simplify such expressions using the RVs only to distinguish the densities: $\Pr(X = x, Y = y) = \Pr(X = x)\Pr(Y = y)$, with the idea being that the distributions must be different for these different arguments. (When unambiguous, it is also common to drop the values x and y , $\Pr(X, Y) = \Pr(X)\Pr(Y)$.) If there is ambiguity, notation should always make it clear which RV goes with which distribution.

i If two or more RVs follow the same distribution, they are *identically distributed*. If they are also independent, they are *independent and identically distributed* or **iid**.

Probability mass and probability density For a discrete random variable like the coin flip, the support of X is a finite or countably infinite set. In comparison, a continuous random variable is one with an uncountably infinite support. Typically these RVs are real-valued.


The distribution for a discrete random variable is referred to as a *probability mass function* (**pmf**). The pmf assigns probability to each supported value x such that $\sum_x \Pr(X = x) = 1$ (where the sum runs over all supported values). The probability that X takes on *some* value is 1 so the probabilities must sum to 1 over the supported values.

On the other hand, for a continuous random variable, the probability function is referred to as a *probability density function* (**pdf**), which assigns probability *densities* to values of x . The distinction here is that for a continuous random variable there will be an infinite number of possible values, so each value must be assigned only a infinitesimal probability in order to be normalized. In other words, the values of a pdf are *not probabilities*! Instead, they are probability *densities*. And there is zero *probability mass* at any precise value of the random variable ($\Pr(x) = 0$); the probability mass only exists when we consider an interval—for instance, the probability that x is between a

and b is $\Pr(a < x < b) = \int_a^b \Pr(x)dx$. Defined in this way, normalization is ensured through an integral over the supported values:

$$\int \Pr(x)dx = 1. \quad (4.13)$$

The area under the curve must be equal to 1 for the distribution to be properly normalized.

 A further consequence of properly normalized pdfs is that the probability densities do not necessarily remain below 1 as probabilities do. Suppose X is supported for $0 \leq x \leq 1/2$. The width of this interval is less than 1, so the height of any pdf defined on it must exceed 1 at some point for the area under the curve to equal 1.

Cumulative distributions For pdfs and pmfs defined on ordinal support, we can derive a *cumulative distribution function* (**CDF**)¹³ that also assigns probabilities to values of x . Instead of asking what is the probability or density of a particular value x , the CDF asks what is the *probability of having a value less than or equal to x* . The CDF $F_X(x)$ is defined as:

$$F_X(x) = \Pr(X \leq x) = \begin{cases} \sum_{x' \leq x} \Pr(x') & \text{for a pmf,} \\ \int_{-\infty}^x \Pr(x')dx' & \text{for a pdf.} \end{cases} \quad (4.14)$$

The CDF is a monotonically increasing function defined over the entire support of X and it connects the percentiles of X to each value of x (for instance, the 50th percentile or median of x is the x where $F_X(x) = 1/2$). In the context of network analysis, we also use the *other* cumulative distribution function, called the *complementary cumulative distribution function* (**CCDF**)¹⁴ and defined as:

$$\bar{F}_X(x) = \Pr(X > x) = 1 - F_X(x). \quad (4.15)$$

While conveying the same information as the CDF, the CCDF is particularly useful when we examine RVs that are strongly *skewed* such that values far larger than the mean and median have some nonvanishing probability of being observed.¹⁵ A logarithmic plot of the CCDF will visually stretch out the small probabilities assigned to the extreme values, whereas those values will be squashed visually as they accumulate near 1 when plotting the CDF.

Both the CDF and CCDF can be obtained from the functional form of the pmf or pdf using Eqs. (4.14) and (4.15), or estimated from the actual data points (creating what are called the “empirical” CDF and CCDF; see Exercises and Ch. 11).

4.2.5 Commonly encountered distributions

There is unlimited variety in probability distributions, both pmfs and pdfs, but some types of distributions are so fundamental and so frequently encountered that they are given specific names. Here we discuss a few.

¹³ The CDF is sometimes just called the *distribution function*.

¹⁴ The definitions for CDFs and CCDFs are sometimes exchanged.

¹⁵ The degree distribution of networks often exhibits such a skewed or “heavy-tailed” distribution, making this scenario especially important when studying networks.

Bernoulli distribution A Bernoulli random variable X represents a binary or two-outcome variable ($x = 0$ or $x = 1$) with a constant probability for outcomes:

$$\begin{aligned}\Pr(X = 1) &= p, \\ \Pr(X = 0) &= 1 - p,\end{aligned}\tag{4.16}$$

or, written more compactly,

$$\Pr(X = x; p) = p^x(1 - p)^{1-x} \text{ for } x \in \{0, 1\}.\tag{4.17}$$

The *Bernoulli distribution* is foundational since so many problems can be reduced to binary events, such as true/false or win/lose outcomes. In the network context, whenever we think about random processes such as creating edges at random in a null model, we think about Bernoulli trials.

Binomial distribution The *binomial distribution* is a companion to the Bernoulli. It describes the number of positive outcomes k from a collection of n independent, identically distributed (*iid*) Bernoulli variables, where each Bernoulli has a positive outcome with probability p :

$$\Pr(X = k; n, p) = \binom{n}{k} p^k (1 - p)^{n-k},\tag{4.18}$$

for $k = 0, 1, \dots, n$ and

$$\binom{n}{k} = \frac{n!}{k! (n - k)!}$$

is the *binomial coefficient*. Here n and p serve as parameters, $p^k(1 - p)^{n-k}$ is the probability of one particular way to achieve k successes and $n - k$ failures, and $\binom{n}{k}$ captures the number of arrangements of these k successes across n trials. We can write $X \sim \text{Binom}(n, p)$ to denote that X follows a binomial distribution with parameters n and p .

In the context of networks, the binomial can, for instance, describe the degree distribution of a random graph, which is created by performing Bernoulli trials for every possible pair of nodes—an edge exists between the two nodes with probability p or it does not with probability $1 - p$.

Poisson distribution The binomial distribution can be well approximated, when n is large and p is small, by a *Poisson distribution*:

$$\Pr(X = k; \lambda) = \lambda^k \frac{e^{-\lambda}}{k!},\tag{4.19}$$

where $\lambda = np$.¹⁶ A Poisson gives the distribution of the number of events in a fixed time (or space) interval when events occur independently from one another and the *rate* of events λ is known.

¹⁶ Actually, this is imprecise. The Poisson distribution derives in the limit from the binomial distribution by taking $n \rightarrow \infty$ and $p \rightarrow 0$ such that the expected value $np \rightarrow \lambda > 0$.

The correspondence between the binomial and the Poisson can be understood as follows. If we take a continuous time interval and divide it up into many small intervals, and assume at most one event can occur per segment,¹⁷ the number of segments containing an event can be represented by a binomial with n as the number of segments. If we divide into smaller and smaller segments and reduce the probability for a single event to occur such that the total rate of events occurring is constant (or, $n \rightarrow \infty$ and $p \rightarrow 0$ such that $np \rightarrow \lambda$) then we can show that in the limit the binomial distribution will converge to the Poisson distribution.¹⁸

Normal (Gaussian) distribution A *normal distribution* or *Gaussian distribution* is one of the classic bell-shaped probability distributions. You are likely already familiar with it. It is ubiquitous due to the *central limit theorem*—the normal distribution is the limiting form for the distribution of sums of *any* iid RVs with finite variances. Thanks to the central limit theorem, both binomial and Poisson distributions can be approximated by a normal distribution when the number of trials is large.

Its pdf is given by

$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2} \left(\frac{x - \mu}{\sigma}\right)^2\right), \quad (4.20)$$

for $x \in \mathbb{R}$, where μ and σ^2 are the mean and variance, respectively, of the random variable. We denote that an RV X is normally distributed with mean μ and variance σ^2 with $X \sim \mathcal{N}(\mu, \sigma^2)$. A *standard normal distribution* is one with $\mu = 0$ and $\sigma^2 = 1$.

Log-normal distribution Sometimes, it is not the “raw” value that is normally distributed, but the *logarithm* of the value:

$$f(x; \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right). \quad (4.21)$$

Just as the normal distribution arises when summing multiple iid RVs (via the central limit theorem), the *log-normal distribution* is produced by a multiplicative process. In other words, when many independent (positive) RVs are multiplied together, the resulting random variable approaches the log-normal distribution (this is also called *Gibrat’s law*). For instance, in the distribution of wealth, income, or other financial applications (e.g., the *Black–Scholes model* for option pricing), a log-normal distribution is often a good approximation because the underlying process is *multiplicative*.

The log-normal and many other distributions are said to be *broadly distributed* or *heavy-tailed* in the sense that the distribution spans multiple orders of magnitude and a large value of x can be expected every so often, a value of x that we would never see under a normal distribution with the same mean.

¹⁷ Which becomes more valid of an assumption as segments become smaller.

¹⁸ In fact, in Ch. 22 we derive this not using a traditional calculation but to demonstrate a combinatorial tool called *generating functions* that are helpful for mathematically analyzing networks and other problems.

Power-law distribution Another common “broad” distribution is a *power-law distribution* $p(x)$ where the pdf (x continuous) or pmf (x discrete) follows a power-law functional form:

$$p(x) = Cx^{-\alpha}, \quad (4.22)$$

with exponent $\alpha > 0$ (usually between 2 and 4) and a suitable normalization constant C . Some choices are necessary for the support (since α is positive, we must exclude $x = 0$) to ensure normalization, which dictates the possible values of α ; both affect whether the mean, variance, or other moments are finite.

An extension of a *pure* power-law distribution is a power law with an exponential cutoff,

$$p(x) = Cx^{-\alpha}e^{-\lambda x}. \quad (4.23)$$

This introduces a second parameter λ , which governs the value of x when the power law is “overwhelmed” by the exponential. Note that Eq. (4.23) reduces to Eq. (4.22) when $\lambda = 0$.

A useful property of power-law distribution is that it shows up as a straight line in a log–log plot of x and $p(x)$. If we take the logarithm of both sides of the equation, we get

$$\log p(x) = \log C - \alpha \log x. \quad (4.24)$$

In log–log scale, this is a straight line with slope $-\alpha$ and intercept $\log C$. Furthermore, the CCDF of a power-law distribution is *also* a power-law distribution, with exponent $\alpha - 1$ instead of α ,

$$\int_x^\infty Cy^{-\alpha} dy = \frac{C}{1-\alpha} [y^{1-\alpha}]_x^\infty = C'x^{-(\alpha-1)}. \quad (4.25)$$

Because of this and the abundance of heavy-tailed degree distributions in real-world networks, examining CCDFs in log–log scale is a common exploratory analysis technique for network data. We will revisit power-law distributions again in Chs. 11 and 12.

Note that a power-law probability distribution is distinct from a *power-law scaling relation*. For instance, Newton’s law of gravitational attraction in classical mechanics $F = G \frac{m_1 m_2}{r^2} \propto r^{-2}$, an “inverse-square” law, where $A \propto B$ means that there exists some constant c such that $A = cB$, is mathematically a power-law functional form relating F and r but is not describing the distribution of a random variable.

Broadly distributed random variables appear often in network data, especially as degree distributions. The degree distribution of many networks can be explained either by log-normal distribution or the power-law distribution, as we will see later in this book. Because a log-normal distribution exhibits a heavy tail due to its logarithmic nature, it is often not easy to distinguish this distribution from the power-law distribution (or vice versa). In fact, identifying statistically broad distributions from finite data samples is often fiendishly difficult (Ch. 11 and Sec. 22.6)

While not exhaustive, the above distributions are the most commonly encountered by scientists working with (network) data and when analyzing and implementing data analysis methods.

4.3 Statistics

As a subject, statistics is often lumped in with probability, but probability is a fully axiomatic branch of mathematics whereas statistics is a mathematical science that leverages probability (in particular) to understand better the properties of data. Statistical questions often ask about underlying phenomena that give rise to empirical data by comparing that data with probabilistic models. Answering such questions requires care, and often we must thoughtfully consider *uncertainty* when data are noisy or incomplete.

4.3.1 Summary statistics

We often turn to summary or descriptive statistics to quantify our data. Measures of *central tendency* can tell us what are “typical” values, and measures of *dispersion* tell us how tightly packed values tend to be around their centers. The form of our data dictates what mathematical operations, and therefore what statistics, we can use. Categorical variables cannot be added or multiplied, for instance, but we can tell whether two values are equal, so we can only use the mode for a categorical’s measure of central tendency. For a numeric variable, however, those operations are permitted, giving us medians or means to measure central tendencies.

When computing (summary) statistics, such as means and variances, throughout this book, we’ll use the following notation.

The *mean*, *expectation*, or *expected value* of a random variable X is denoted $\langle x \rangle$ (sometimes $E[X]$) and given by

$$\langle x \rangle = \sum_x xP(x) \quad (\text{pmf}) \quad \langle x \rangle = \int_{-\infty}^{\infty} xP(x) dx \quad (\text{pdf}), \quad (4.26)$$

where the summation or integration is over X ’s support. The expectation of a function $f(x)$ of an RV is

$$\langle f(x) \rangle = \sum_x f(x)P(x) \quad (\text{pmf}) \quad \langle f(x) \rangle = \int_{-\infty}^{\infty} f(x)P(x) dx \quad (\text{pdf}). \quad (4.27)$$

Using this, the n th *moment* (about zero) of a distribution is given by $\langle x^n \rangle$. The mean is the first moment. The *variance* of X compares the first and second moments,

$$\text{Var}(X) = \langle (X - \langle X \rangle)^2 \rangle = \langle X^2 \rangle - \langle X \rangle^2. \quad (4.28)$$

(The variance of X is also commonly denoted by σ_X^2 or just σ^2 when the context is clear.) The *standard deviation* is the square root of the variance. The variance is a special case of the *covariance* $\text{Cov}(X, Y)$ for two RVs ($\text{Var}(X) = \text{Cov}(X, X)$):

$$\text{Cov}(X, Y) = \langle (X - \langle X \rangle)(Y - \langle Y \rangle) \rangle = \langle XY \rangle - \langle X \rangle \langle Y \rangle. \quad (4.29)$$

The covariance tells us how strongly related two variables are, but it’s also common to use the (Pearson) *correlation coefficient*,

$$\rho_{XY} = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y} = \frac{\langle XY \rangle - \langle X \rangle \langle Y \rangle}{\sqrt{\langle X^2 \rangle - \langle X \rangle^2} \sqrt{\langle Y^2 \rangle - \langle Y \rangle^2}}, \quad (4.30)$$

instead of the covariance to measure how strongly correlated or linearly related the variables are to each other. Rescaling the covariance in this way ensures $-1 \leq \rho \leq 1$, making for a more interpretable statistic and more easily compared across different datasets.

Useful properties Expectation is linear:

$$\langle c_1 X + c_2 Y \rangle = c_1 \langle X \rangle + c_2 \langle Y \rangle \quad c_1, c_2 \text{ constants}, \quad (4.31)$$

$$\left\langle \sum_{i=1}^n c_i X_i \right\rangle = \sum_{i=1}^n c_i \langle X_i \rangle \quad \text{in general.} \quad (4.32)$$

The variance obeys the following (c, c_i constants):

$$\text{Var}(X) \geq 0, \quad (4.33)$$

$$\text{Var}(X + c) = \text{Var}(X), \quad (4.34)$$

$$\text{Var}(cX) = c^2 \text{Var}(X) \quad (\text{that } c \text{ is squared is a common "gotcha"}), \quad (4.35)$$

$$\text{Var}\left(\sum_{i=1}^n c_i X_i\right) = \sum_{i=1}^n \sum_{j=1}^n c_i c_j \text{Cov}(X_i, X_j) \quad (4.36)$$

$$= \sum_{i=1}^n c_i^2 \text{Var}(X_i) + \sum_{i \neq j} c_i c_j \text{Cov}(X_i, X_j), \quad (4.37)$$

$$\text{Var}\left(\sum_{i=1}^n c_i X_i\right) = \sum_{i=1}^n c_i^2 \text{Var}(X_i) \quad (\text{if the } X_i \text{ are uncorrelated}). \quad (4.38)$$

Sample statistics Lastly, it can be important to distinguish between the statistics of a random variable defined above and the statistics computed from a data sample. For a collection of n data points $\{x_i\}$ and $\{y_i\}$, define

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \quad (\text{sample mean}), \quad (4.39)$$

$$s_x^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2 \quad (\text{sample variance}), \quad (4.40)$$

$$s_{xy}^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y}) \quad (\text{sample covariance}), \quad (4.41)$$

$$r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}} \quad (\text{sample correlation}). \quad (4.42)$$

The sample covariance is unbiased when using $n-1$ in the denominator (this is called Bessel's correction).

While the statistics of a random variable are perfectly defined, they are facts of the distribution, the sample statistics themselves vary depending on the data sample being

used. A different sample would lead to a different sample statistic, with this difference decreasing with increasing sample size.

Standardizing A standard score z is useful when we wish to “correct” for the mean and variance of x :

$$z = \frac{x - \langle x \rangle}{\sigma_x}. \quad (4.43)$$

A standard score of z means that x is z standard deviations above (or below, if negative) the mean of x . This z -score lets us compare different data on a common scale and lets us perform z -tests using the properties of the standard normal. Note that if the variable we are standardizing is itself the sample mean of a randomly sampled underlying variable, z can be expressed with the underlying variable’s standard deviation with a correction in the denominator:

$$z = \frac{\bar{x} - \langle \bar{x} \rangle}{\sigma_x / \sqrt{n}}. \quad (4.44)$$

Intuitively, this makes sense, as the variability in \bar{x} should decrease with increasing n .

4.3.2 Inference

Statistical inference is a process of inferring models (assumptions) that explain the data well. The common approaches to inference are (null) hypothesis testing and Bayesian inference.

Hypothesis testing proceeds by defining a null hypothesis related to a question of interest. For example, if we want to understand how strongly correlated x and y are, we could take as a null hypothesis that “ x and y are uncorrelated.” We then consider a *test statistic*, something we can measure in our real data and can understand, traditionally mathematically, how it behaves assuming the null is true. We then measure the test statistic in our real data, and ask how probable it is to see a value that large or larger under the null hypothesis. This “tail probability” is given by the CDF of the null distribution of the test statistic. If that probability, called a p -value, is sufficiently small, we can argue that it is unlikely for the null hypothesis to be true. Hypothesis testing is a richly developed set of tools, with many tests for many situations and careful ways to report results. It brings a lot of baggage, however, and is (often, rightfully) criticized. For instance, we are actually only testing the null, not any actual hypothesis. Likewise, we often rely on arbitrary thresholds to determine whether a test is “significant”; commonly, if $p < 0.05$ the test is significant, otherwise it is not. And, since a smaller p -value is a stronger result, researchers often, intentionally or not, find themselves optimizing for p , a misleading practice called *p-hacking* (Ch. 3). Used properly, hypothesis testing has its place, and it can be simple and effective, but these concerns should be front of mind.

Bayesian inference is built on Bayes’ theorem. We’ve seen Bayes’ theorem before

(Eq. (4.11)) but let's write it again using two symbols D and θ , and label it:

$$\underbrace{P(\theta | D)}_{\text{"posterior"}} = \frac{\overbrace{P(D | \theta)}^{\text{"likelihood"}} \overbrace{P(\theta)}^{\text{"prior"}}}{\underbrace{P(D)}_{\text{"evidence"}}}. \quad (4.45)$$

Here D refers to *data* and θ refers to *parameters*. $P(\theta)$ is called the *prior*, or the prior probability distribution of the parameters. $P(D | \theta)$ is called the *likelihood*, or the likelihood that our model with parameters θ generates the data D . $P(D)$ is usually called the *evidence* or *marginal likelihood*.¹⁹ Finally, $P(\theta | D)$ is called the *posterior*, or the posterior probability distribution of θ given the data.

Once we write the theorem in this way, we can interpret it as following: we can learn about the conditional probability of parameters given data ("posterior") by knowing the conditional probability of data given parameters ("likelihood"), the probability of parameters ("prior"), and the marginal likelihood of data ("evidence"). In effect, the left-hand side of Eq. (4.45) tells us what statistical models (what parameters) are probable given our data and what models are improbable, but this probability is not easy to compute. The terms on the right-hand side are computable so, thanks to Bayes' theorem, we have a way to address the posterior, the probability we wish to assess.

If you are not familiar with Bayesian inference, you may be wondering how can we even think about the "marginal likelihood of data" or the "prior probability distribution of parameters." Those are great questions! In Bayesian statistics, probability is conceptualized, not as a fixed number that we can estimate by performing many trials, but as the degree of belief that an outcome will occur. That is why we can think of the *prior* probability distribution of parameters. We explicitly state what we believe (or don't believe) about our parameters prior to seeing the data. Data then lets us adjust our initial belief (prior) to a more *informed* belief (posterior).

Let's walk through a toy example. Imagine yourself catching some fish on a fishing boat floating on a lake. Your goal is to create a good model that explains how many fish you can catch (per hour) from the lake. Your *data* are the number of fish you caught in the past several hours, say:

$$D = [4, 1, 2, 10]. \quad (4.46)$$

Then we can come up with some probabilistic models to explain the data. Suppose one model (M_1) assumes a uniform distribution of the number of fish caught with a single parameter N that determines the maximum number of fishes, or

$$\Pr(n; M_1, N) = \begin{cases} \frac{1}{N+1} & n \in \{0, \dots, N\}, \\ 0 & \text{otherwise.} \end{cases} \quad (4.47)$$

For a second model (M_2), let's assume a Poisson distribution, which describes processes that happen with a constant rate:

$$P(n; M_2, \lambda) = \frac{\lambda^n e^{-\lambda}}{n!}. \quad (4.48)$$

¹⁹ This name stems from writing the denominator as $P(D) = \sum_{\theta'} P(D | \theta') P(\theta')$.

Once we have our models, we can think about the likelihood, which captures how *likely* it is to have the data given a model. Unlike the probability of a model given the data, the likelihood is straightforward to calculate. For instance, if we assume our datapoints are iid, for the first model with $N = 10$, the likelihood is:

$$\Pr(D \mid M_1; N = 10) = \prod_i \Pr(d_i \mid M_1; N = 10) \quad (4.49)$$

$$= \frac{1}{11} \cdot \frac{1}{11} \cdot \frac{1}{11} \cdot \frac{1}{11}. \quad (4.50)$$

Likewise, we can compute the likelihood of M_2 in the same way (this time without plugging in a parameter value):

$$\Pr(D \mid M_2; \lambda) = \prod_i \Pr(d_i \mid M_2; \lambda) \quad (4.51)$$

$$= \frac{\lambda^4 e^{-\lambda}}{4!} \cdot \frac{\lambda^1 e^{-\lambda}}{1!} \cdot \frac{\lambda^2 e^{-\lambda}}{2!} \cdot \frac{\lambda^{10} e^{-\lambda}}{10!}. \quad (4.52)$$

The ability to calculate the likelihoods lets us compare models and parameters. Let's focus on the first model and think about the ratio of two posterior probabilities given two different parameters ($N = 10$ and $N = 20$):

$$\frac{\Pr(N = 10 \mid D)}{\Pr(N = 20 \mid D)} = \frac{\Pr(D \mid N = 10) \Pr(N = 10) \Pr(D)}{\Pr(D \mid N = 20) \Pr(N = 20) \Pr(D)} \quad (4.53)$$

$$= \frac{\Pr(D \mid N = 10) \Pr(N = 10)}{\Pr(D \mid N = 20) \Pr(N = 20)}. \quad (4.54)$$

If this ratio is larger than 1, that means, given the data, $N = 10$ is more likely than $N = 20$. Starting from here, there are multiple ways to approach the inference—identifying good models and their parameters. In general, the process of comparing and then choosing between models using data is called *model selection*.

4.3.3 Maximum likelihood estimation

We can see from Eq. (4.54) that the ratio of two posterior probabilities is solely determined by the likelihood and the prior. Let's first assume that we do not have any knowledge or prior belief about what the parameters should be. Then we can assume an equal prior for all possible parameters ($\Pr(N_1) = \Pr(N_2)$ for any N_1 and N_2). In this case, the ratio of posteriors is entirely determined by the *likelihood*—the higher the likelihood of a given parameter, the higher the posterior probability of that parameter, which makes sense! In other words, under these conditions,

$$\Pr(\theta \mid D) \propto \Pr(D \mid \theta). \quad (4.55)$$

Then, the best parameter given data can be identified by finding the parameter that maximizes the likelihood:

$$\theta_{\text{MLE}} = \arg \max_{\theta} P(\theta \mid D) = \arg \max_{\theta} P(D \mid \theta). \quad (4.56)$$

(The “arg max” of a function $f(x)$ is the value(s) of x for which $f(x)$ is maximized. A similar definition holds for “arg min.”) This is the basic idea of *maximum likelihood estimation* (MLE). It is built on the assumption that, when we don’t have any information or prior belief about parameter distribution, it is reasonable to assume that the parameter that maximizes the likelihood is the best parameter. Note that MLE is a *point estimate*—it identifies a single parameter value that maximizes the likelihood but it cannot tell us what is the posterior probability *distribution*.

4.3.4 Maximum a posteriori

What if we have some information or belief about the prior distribution of the parameters? For instance, maybe we have reasons to believe that the fish yield of the lake should be around 5. Maybe lakes of this size and characteristics tend to have similar numbers of fish.

Bayes’ theorem provides a straightforward way to incorporate this information through the prior distribution of the parameter $\Pr(\theta)$. We just need to keep the prior:

$$\theta_{\text{MAP}} = \arg \max_{\theta} \Pr(\theta|D) = \arg \max_{\theta} \Pr(D|\theta) \Pr(\theta). \quad (4.57)$$

This is called *maximum a posteriori* (MAP) inference. For many common situations, MAP point estimates tend to be a simple interpolation between the MLE point estimate and the mean of the prior distribution. For more information, read about “conjugate priors” and the exponential family.

4.3.5 Bayesian inference

MLE and MAP obtain *point estimates*—a single point in the parameter space that maximizes the likelihood or posterior. Although a point estimate can be the “solution,” it is less useful than inferring the full distribution and sometimes can be misleading. For instance, imagine a hypothetical case where the likelihood function looks like Fig. 4.1.

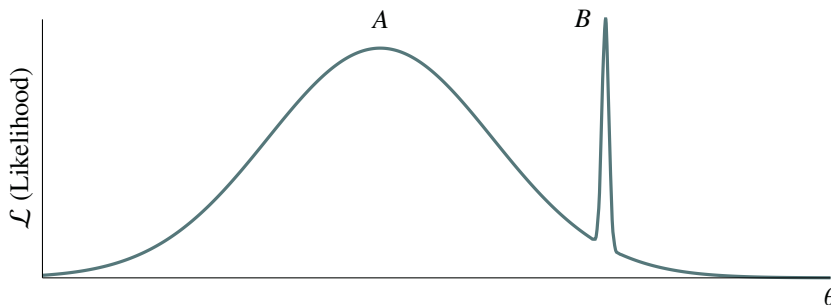


Figure 4.1 A hypothetical likelihood function where most of the probability mass exists (A) far away from the maximum likelihood point (B).

As you can see, MLE will (or may) point us to B , where the sharp peak is located. However, the peak is extremely sharp and the probability mass around the MLE is

negligible compared with the broad peak around A . In this case, can we justify the result of MLE, particularly given that most parameter values will be around A if we sample from this distribution?

As shown in this case, a point estimate can be misleading and miss critical information about the distribution. With a single estimated parameter value, it is impossible to know the uncertainty around the estimate and it is also impossible to know whether there exist other parameters that are almost equally likely. When we say “(full) Bayesian inference,” we refer to the practice of considering the posterior probability *distribution* rather than the point estimate like MLE or MAP.

Sometimes a full Bayesian inference can be accomplished mathematically, by writing down an expression for the posterior probability. Unfortunately, this tends to be possible only for simple models with tractable likelihoods. In practice, we often resort to computational approaches called *Markov Chain Monte Carlo* (MCMC) that avoid computing the posterior and instead generate large numbers of samples (parameter values) that will be distributed following the posterior.²⁰ MCMC can be expensive to compute, and may need careful guidance to ensure the posterior is being sampled correctly. This hampered Bayesian inference in the past, but computers are quite fast nowadays, and many new MCMC algorithms make inference both faster and more reliable.

4.4 Summary

Network science is a broadly interdisciplinary field, pulling from computer science, mathematics, statistics, and more. The data scientist working with networks thus needs a broad base of knowledge. Although helpful, we do not expect the reader to know all these areas well. Instead, the reader should be prepared enough that they can learn this material without also needing to learn much of the background that the primer itself depends on. This chapter serves as a primer for the reader on background information that we will use through the rest of the book. When necessary, we will refer back to these materials as we proceed.

Bibliographic remarks

Many resources abound for learning programming in general and for data science in particular. These days, the two major programming languages for data scientists are Python and R. Readers wanting to learn more on practical programming in Python may consider *Python for Scientists* by Stewart [444] or *An Introduction to Python Programming for Scientists and Engineers* by Lin et al. [276], while *R for Data Science* by Wickham and Golemund [491] is an excellent entry point for those interested in R. *Introduction to Algorithms* by Cormen et al. [117] is one of the classic starting points for studying algorithms and data structures.

Readers seeking to learn more linear algebra are encouraged to begin with the classic *Introduction to Linear Algebra* (Strang [447]) or the more recent *Linear Algebra and*

²⁰ An alternative to MCMC is *variational inference*, which approximates or lower bounds the intractable denominator in Eq. (4.45).

Learning from Data (Strang [446]), which gives a broad introduction to linear algebra using statistics and machine learning as the stage.

For readers interested in brushing up on probability and statistics, we recommend the excellent *All of Statistics* by Wasserman [484], the perennial classic *Probability & Statistics for Engineers & Scientists* by Walpole et al. [481], and *Computer Age Statistical Inference* by Efron and Hastie [142], the latter giving an exciting blend of statistics and computing. For Bayesian statistics, *Doing Bayesian Data Analysis* by Kruschke [256] is a practically focused introduction, while *Bayesian Data Analysis* by Gelman et al. [179] is suitable for those with more background.

Exercises

- 4.1 A *multiset* acts like a set except duplicates are allowed, so multiset elements need not be unique. Suppose you are working with a programming language that gives you `set`, `list`, and `dict` (dictionary) data structures. Describe how to implement `multiset` using these built-in data structures.
- 4.2 We have at the ready an infinite number of true/false questions. Assume questions are unrelated and each question has probability $p = 1/20$ to have an answer of T, otherwise the answer is F. We continue to ask questions until the first T answer. How many questions should we *expect* to ask?
- 4.3 Show that $\text{Var}(X) = 0$ if and only if $\Pr(X = c) = 1$ for some constant c .
- 4.4 The cumulative distribution function (CDF) $F(x)$ (Eq. (4.14)) can be estimated from data using the empirical CDF (ECDF):

$$F_n(x) = \frac{1}{n} \sum_{i=1}^n I(X_i < x), \quad (4.58)$$

where

$$I(X_i < x) = \begin{cases} 1 & \text{if } X_i \leq x, \\ 0 & \text{otherwise.} \end{cases} \quad (4.59)$$

An advantage of the ECDF over a typical *histogram* for estimating a distribution is that the ECDF is defined for all n data points, while the histogram is defined at fewer points due to the need to bin the data.

Describe with pseudocode a simple function that computes the ECDF efficiently using a `sort` function.

- 4.5 We wish to study how a network with N nodes and M edges changes when an edge is removed. We use a method $f(i, j)$ that compares a pair of nodes i, j . We study what removing an edge does by computing f before and after the edge is removed. Computing f has complexity $O(N + M)$ for one pair. For one edge removal, what is the complexity of checking the change in f over every pair of nodes? What is the complexity of checking every edge?