Data Literacy

William John Holden

Contents

Chapter 1

Introduction

1.1 Parameters and statistics

Statistics are the foundation of most data mining, machine learning (ML), and artificial intelligence (AI) methods today. A *statistic* is an estimate of a *parameter*, which is a characteristic of an entire *population*. Statistics are calculated from taking *samples* (subsets) from the population.

For example, suppose we wanted to find the height of the tallest mountain in the world. We might sample $n = 100$ mountains at random from an almanac. Suppose the tallest mountain in our sample is Mount Fuji. Mount Fuji, the tallest mountain in Japan, is 3776 meters tall. We can conclude that the tallest mountain in the world is *at least* 3776 meters tall.

Our estimate is unfortunately quite low. Mount Everest in Nepal, the *highest* mountain in the world, stands 8849 meters above sea level. Mauna Kea in Hawai'i, the *tallest* mountain in the world, stands 4207 meters above sea level and another 6004 meters below. Our estimates of population parameters, *statistics*, generally improve with larger sample sizes, and many statistical methods provide a *margin of error* quantifying sampling error.

One might use statistics to create a *model* to explain a population, based upon sampling data. Models can be useful both for describing the population and also for forming predictions.

1.2 Levels of measurement

There are four distinct *levels of measurement* that a value may fit. *Nominal* data is simply names or categories, with no concept of order or distance. A movie might be animated or live-action: these are simple categories or order.

Another example might be the film's genre (children, comedy, action, romance, documentary, etc).

Ordinal data has ordering but not distance. Ordinal data might be represented as ordered categories or as numerals, though these numerals do not provide meaningful addition and subtraction. The ratings of a film (G, PG, PG-13, R, and so on) form a ranking, but addition is meaningless (does G + PG-13 = R?) and our concept of distance is weak at best. Another example of ordinal might be the rankings the films receive at an awards ceremony, where one film is the winner and another is the runner-up.

Interval data is numerical data with a concept of distance but not multiplication. The year when a film was produced is an example of interval data. If two films were produced in 2000 and 2010, then it makes sense to say one was made ten years later, but we would not say that the latter film is \$2010/2000 = 1.005\$ times the first.

Ratio data is numerical data with both distance and multiplication. The gross earnings of a film is an example of ratio data. If the 2000 film earned one million dollars and the second earned two million dollars, then it makes sense to say the second film earned double the first.

Name	Operations	Type
Nominal	$=, \neq$	Categories
Ordinal	$<, >$	Ordered categories
Interval	$+, -$	Numbers with distance
Ratio	$\times, \div$	Numbers with meaningful zero

Interval data might be initially confusing to distinguish from ratio data. One indication is the absence of a meaningful zero. Does zero degrees Celsius or Fahrenheit mean the absence of temperature? No, these measurements are simply points along a scale. Twenty degrees Celsius is not "twice" ten degrees Celsius; multiplication is not defined on interval data.

Grid coordinates might be another example of interval data. One can calculate the distance between two grid coordinates, but we would not say that coordinate 1111 is "half" of coordinate 2222.

Data might be represented in numerical formats when some operations do not make sense. Suppose a political scientist encoded voter's political party as "1", "2", "3", and "4". Is "2" an intermediate value between "1" and "3", or are these actually nominal data where the only arithmetic operations are = and $\neq$? AI methods sometimes make incorrect assumptions about data that domain experts can easily prevent.

1.3 Discretization

Measurements with arbitrarily many decimal digits of precision are *continuous*, whereas measurements with finite steps in between (including categories) are *discrete*. For example, when driving along a road, the house numbers (150 2nd Street, 152 2nd Street, 154 2nd Street…) are discrete; there is no intermediate value between 150 and 151. On the other hand, the grid coordinates associated with each address are continuous; one could (theoretically) specify grid coordinates to the nanometer.

It can be useful to combine continuous measurements into discrete categories. An example might be one's birth date and birth year. No one knows their birth *instant* with subsecond precision. Rather, the year, year and month, or year, month, and day are almost almost always enough information. We even combine years into groups when discussing generations and peer groups. Combining a range of birth years into generational categories is an example of *discretization*.

1.4 Missing values

In practice, sets of data (a *data set*) are often missing values. Different programming languages have substantially different syntax and semantics for representing and handling missing values.

As a small exercise, open Microsoft Excel and enter the values 1, 2, 3, and 5 into cells A1, A2, A3, and A5. Leave cell A4 blank. In cell A6, enter the formula `=PRODUCT(A1:A5)`. The result is $30 = 1 \cdot 2 \cdot 3 \cdot 5$. Excel did *not* treat the missing value as a zero.

Now change cell A4 to `=NA()`. `NA` means "value not available", an explicit indication that a value is not given. The product in cell A6 should update to `#N/A`, which explicitly tells us that there is a problem in the calculation.

Now change cell A4 to `=1/0`. Both cells A4 and A6 should both say `#DIV/0!`, a fault telling us that a division by zero has made further calculation impossible.

Error values propagate from source data through intermediate calculations to final results. If we enter a formula into A7 referencing A6, such as `=SQRT(A6)`, then we will find the same faults in A7 that we see in A6.

Structured Query Language (SQL) databases use the symbol `NULL` to denote missing values. One might build the database *schema* (the structure of the database) to explicitly forbid `NULL` values. For example, `CREATE TABLE Run (Name TEXT NOT NULL, Time INTEGER NOT NULL, Distance REAL NOT NULL)` defines a table *schema* where each of the three columns must be specified. Many programming languages (including C, Java, and JavaScript) also use the term `null` for variables that do not reference any specific value.

Many programming languages support a `NaN` ("not a number") value in error conditions. One might encounter `NaN` when dividing by zero, subtracting infini-

ties, and parsing non-numeric words as numbers. Comparisons with `NaN` can be confusing, such as `NaN == NaN` returning *false*.

Some programming languages will automatically *initialize* variables with some zero value. Other languages give some `Undefined` value to uninitialized variables. Still other languages raise an error if no explicit value is assigned to a variable.

1.5 Strong/weak and static/dynamic typing

Values come in many forms: categorical and numerical, ordered and unordered, discrete and continuous, defined and missing. *Types* can be used to constrain variables to allowable values and applicable operations.

For example, suppose a database indicates how many cars a person owns. It makes no sense to own a fractional or negative car, so we might find an existing type (in this case, whole numbers) or define some new type to model the domain.

Some programming languages offer *dynamic* types that implicitly change the type (*cast*) of values to operate correctly. Go to https://jsfiddle.net or press F12 to open the developer console in most modern browsers. Enter the following into the JavaScript console:

```
>> "5" * 5
<- 25
```

Characters inside quotation marks (`"5"`) are called *strings* and are ordinarily used for text, but JavaScript automatically parses `"5" * 5` as the product of two numerical values and returns `25`.

JavaScript is notoriously inconsistent.

```
>> "5" + 5
<- "55"
```

The resulting string, `"55"`, is the *concatenation* of two strings – perhaps not what one expects.

Many languages and environments seek to automatically parse values. Microsoft Excel and the Python programming language are also dynamic. Other languages, such as Java and Go, are more strict with values and do not automatically change values, especially when the conversion might be "lossy" (where information might be lost, such as approximating the exact value of π as 3.14, or rounding 3.14 to 3, or even changing 3.0 to 3). These languages have both *strong* and *static* typing: the programmer must specify the type of each variable, and lossy type conversions require an explicit cast.

Excel does provide some basic functionality to set number *formats*, but this feature might not stop one from confusing one type of data for another. Excel uses *weak* typing that does prevent one from using unexpected values. Data

analysts can benefit greatly by using the appropriate types for the values in their problem.

1.6 Tables, lists, and data frames

Tables of data are structured in *columns* and *rows*, where the rows represent the *individuals* or *observations* in the data set and the columns represent the *features*. For example, a table of employee names might have two columns (the given and surnames) and ten rows, where each row represents one of the ten employees.

In computer science, the terms *list* and *array* both refer to single-column tables, but with different internal memory representation. The distinction is usually unimportant to data analysts.

Scientific languages, such as Julia and R, often use the term *data frame* (or *dataframe*) as their method for representing tables of data. Data frames often provide rich syntax for row-wise and column-wise operations. By contrast, in an object-oriented language, such as Java and JavaScript, the idiomatic representation of a table is likely an array of objects.

1.7 Forms and input validation

1.8 Vectors and matrices

We now quickly mention the terms *vector* and *matrix* here to disambiguate them from other terms already defined.

Arrays, lists, and columns containing numeric data may sometimes be represented with *vectors*. Likewise, tables and data frames might be represented with *matrices*.

A vector is a quantity with both magnitude and direction, often consisting of two or more elements.

$$\mathbf{x} = (x_1, x_2, x_3, \cdots, x_n)$$

The above vector $\mathbf{x}$ has three components and length $\sqrt{x_1^2 + x_2^2 + x_3^2 + \cdots + x_n^2}$.

A matrix is a collection of vectors used for linear transformations. For example, the three-component *identity matrix*

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

has the property

$$\begin{aligned} I_3\mathbf{x} &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \\ &= \begin{pmatrix} 1 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 \\ 0 \cdot x_1 + 1 \cdot x_2 + 0 \cdot x_3 \\ 0 \cdot x_1 + 0 \cdot x_2 + 1 \cdot x_3 \end{pmatrix} \\ &= \mathbf{x}. \end{aligned}$$

Vectors and matrices form the foundations of *linear algebra*, a rich and powerful branch of mathematics that produces many of the results needed for modern statistics, ML, and AI methods.

Remember that the different in ratio and interval data was that *multiplication* is only defined for ratio data. Similarly, multiplication is well-defined for vectors and matrices, but not on tables of data. Depending on the problem domain, it may be inappropriate to use matrices and vectors to represent data where such operations are not necessary.

1.9 Data visualization with plots

Plots allow us to visualize data. Good plots help us to quickly intuit patterns in the data that might otherwise be difficult to understand.

(Note: the term *graph* has different definitions in lower and higher mathematics. We will explain the term "graph" in chapter 4. This text uses the term "plot" as the verb and noun for visualizing data with graphics.)

The *bar plot* helps us to compare the count each category in a discrete (or discretized) variable. The *box plot* helps us to see the center and variation of a numerical variable. The *histogram* also helps us to see the center and variation of a numerical variable, often producing the familiar *bell curve* shape, where the height of the curve indicates the count of observations within the range of each "bin." A histogram is essentially a set of bar plots over discretized numerical values.

A *scatter plot* (sometimes called an XY plot) uses x and y axes to show relationships between two variables. One can also color and shape the points to show third and fourth variables. Three-dimensional XYZ plots are sometimes useful, especially in video and interactive presentations.

As a small exercise to experiment with these four plots, go to https://webr.r-wasm.org/latest/ to use the R language in a web browser. R is a programming language for statistics and data visualization.

R includes several built-in data sets. In the *read-evaluate-print loop* (*REPL*), enter

```
> head(mtcars)
```

to view the column names and first six rows of the Motor Trend Cars (`mtcars`) data set. Now enter the following commands to quickly visualize a few columns in the data set.

```
> barplot(mtcars$cyl)
> boxplot(mtcars$mpg)
> hist(mtcars$mpg)
> plot(mtcars$wt, mtcars$mpg)
```

1.10 Linear and logarithmic scales

Scientists use the term *order of magnitude* to compare values only by the power of 10. One would say $a = 1.6 \times 10^3$ is three orders of magnitude smaller than $b = 8.3 \times 10^6$, which is to say $b/a \approx 1000$.

The *scale* of an axis, such as in bar plot, is the spacing between values. A *linear scale* might show marks at 10, 20, 30, 40, and so on. A *logarithmic scale* might show marks at 10, 100, 1000, 10 000, and so on.

Logarithmic scales can be useful for comparing values that differ by more than one order of magnitude. For example, suppose feature of a data set contains categories a, b, c, and d, and the count of each category is

Category	Count
a	10 736
b	1711
c	398
d	319

Return to https://webr.r-wasm.org/latest/ and plot this data with linear and logarithmic scales:

```
> category_counts <- c(10736, 1711, 398, 319)
> category_counts
[1] 10736  1711   398   319
> barplot(category_counts)
> barplot(category_counts, log="y")
```

1.11 Sets, relations, functions, and algorithms

We now introduce a few terms from *discrete mathematics* that are fundamental to all analysis. A *set* is an unordered collection of *distinct* elements. Sets may

be finite or infinite in size. Sets are denoted with curly braces, and the empty set has the special symbol $\emptyset = \{\}$. An example of a set might be

$$W = \{\text{Sunday}, \text{Monday}, \text{Tuesday}, \text{Wednesday}, \text{Thursday}, \text{Friday}, \text{Saturday}\}$$

A *relation* is an association between members of sets. Relations can be used to model any relationship between members any two sets, or even members in the same set. An example might be the relation between integers and elements of W with that many letters, i.e. 6 has a relation on Sunday, Monday, and Friday, 7 has a relation on Tuesday, 8 has a relation on Thursday and Saturday, and 9 has a relation on Wednesday. The term "relation" is seldom used outside of discrete mathematics, but there is a *special case* of a relation that occurs in all mathematical disciplines: *functions*.

A *function* is a relation that uniquely relates members of one set (the *domain*) to another set (the *range*). An example of some functions might be:

$$\begin{aligned}
\text{Translate}\,(\text{Friday}, \text{English}, \text{German}) &= \text{Freitag} \\
\text{Length}\,(\text{Wednesday}) &= 9 \\
\text{Distance}\,(\text{Thursday}, \text{Tuesday}) &= -2 \\
\text{DaysOfLength}\,(6) &= \{\text{Sunday}, \text{Monday}, \text{Friday}\} \\
\text{Sunday} &= \text{Next}\,(\text{Saturday}) \\
&= \text{Previous}\,(\text{Monday}) \\
&= \text{Previous}\,(\text{Previous}\,(\text{Tuesday})) \\
&= \text{Previous}\,(\text{Next}\,(\text{Sunday}))
\end{aligned}$$

Each of these functions accepts one or more *parameters* as *arguments* and returns the unique corresponding value (if any) from its range. It may appear that the third function, DaysOfLength, has returned three values, but actually this function has returned a single set which contains three values.

Many programming languages use the term "function" as a synonym for *procedure*, *subroutine*, and *method*. Functions are "pure" if they have no side-effects, such as mutating a value outside of the function.

The mathematical definition of the term *algorithm* is the set of instructions necessary to solve a problem. Long division, a procedure for manually dividing numbers, is an example of an algorithm. The term "algorithm" has recently entered the popular lexicon in relation to AI systems. Here, the instructions of the algorithm are part of a model, which is created from data.

1.12 Discussion prompts

1. Who owns knowledge management?

2. What are good and bad uses for spreadsheets?
3. What is reproducibility? Why would this be important for scientific inquiry?
4. Like a barplot, a pie chart shows the relative sizes of categorical values. What are some disadvantages of using pie charts?

1.13 Practical exercises

1. Given a dataset, plot the data and explain why this plot technique is appropriate.
2. Given a noisy and poorly structured dataset, propose a method of restructuring the data.
3. Discretize the values of a dataset and explain the reasoning.
4. Be creative and construct intentionally misleading plots that deliberately distort information presented.

Chapter 2

Data Operations

2.1 Relational algebra

Codd's *relational algebra* is the framework theory describing all modern *database management systems* (DBMS). The relational algebra can be described with five primitives: *selection* (σ), *projection* (π), the *Cartesian product* ($\times$; also known as the *cross product*), set *union* ($\cup$), and set *difference* ($\setminus$).

Selection takes all or a subset of a table's rows. Projection takes all or a subset of a table's columns. In Structured Query Languages (SQL), selection is specified in the `WHERE` clause and projection is specified in the list of columns immediately after `SELECT`.

A Cartesian product is the multiplication of sets. If $A = \{i, j\}$ and $B = \{x, y, z\}$, then $A \times B = \{(i, x), (i, y), (i, z), (j, x), (j, y), (j, z)\}$. The Cartesian product produces the set of all possible pairwise combinations of elements in each set. These composite values are called *tuples*. Tuples may contain more than two values. If $C = \{c\}$, then

$$A \times B \times C = \{(i, x, c), (i, y, c), (i, z, c), (j, x, c), (j, y, c), (j, z, c)\}.$$

As an exercise, go to https://sqlime.org/#deta:mb9f8wq2mq0b to use a DBMS named SQLite. Enter the following commands to reproduce the above Cartesian product.

```
CREATE TABLE A (a text);
CREATE TABLE B (b text);
CREATE TABLE C (c text);

INSERT INTO A(a) VALUES ('i'), ('j');
INSERT INTO B(b) VALUES ('x'), ('y'), ('z');
```

```
INSERT INTO C(c) VALUES ('c');

SELECT * FROM A CROSS JOIN B CROSS JOIN C;
```

This text views tuples as unordered and "flattened" sets, and therefore Cartesian products are both *commutative* ($R \times S = S \times R$) and *associative* ($R \times (S \times T) = (R \times S) \times T$). Some mathematical texts use a stricter definition for the Cartesian product where the result is a set, which does not "flatten" and therefore provides neither commutivity nor associativity. This text uses the looser definition for compatibility with practical DBMSs, including SQLite. Mathematics is partly discovered and partly invented.

Set union, $\cup$, combines two sets. Sets definitionally contain only distinct elements. If $A = \{i, j, k\}$ and $B = \{k, l, m\}$, then

$$A \cup B = \{i, j, k, l, m\}\,.$$

Set difference, $\setminus$, retains the elements of the left set that are not present in the right set.

$$A \setminus B = \{i, j, k\} \setminus \{k, l, m\} = \{i, j\}\,.$$

2.2 Join

The *join* ($\bowtie$) is a combination of the Cartesian product and selection. For example, suppose we have a tables named `Swim`, `Bike`, and `Run`. Each table has a column that uniquely identifies an athlete. To get a triathletes (the athletes who participate in swimming, cycling, and running), we use an *equijoin* to find the product where the names are equal. Return to https://sqlime.org/#deta:36fadcq9apak to demonstrate experiment with the `JOIN` operator.

```
CREATE TABLE IF NOT EXISTS Swim (sn TEXT UNIQUE);
CREATE TABLE IF NOT EXISTS Bike (bn TEXT UNIQUE);
CREATE TABLE IF NOT EXISTS Run (rn TEXT UNIQUE);

INSERT OR IGNORE INTO Swim (sn) VALUES
    ('John'), ('Jane'), ('Luke'), ('Phil');
INSERT OR IGNORE INTO Bike (bn) VALUES
    ('Mary'), ('Alex'), ('Jane'), ('Levi');
INSERT OR IGNORE INTO Run (rn) VALUES
    ('Mike'), ('John'), ('Jane'), ('Sven');

SELECT * FROM Swim, Bike, Run WHERE sn = bn AND sn = rn;
```

There are other syntaxes which achieve the same result using the `ON` and `USING` clauses. As an exercise, try to predict how many rows will return from `SELECT * FROM Swim, Bike, Run` without a `WHERE` clause.

2.3 Grouping and aggregation

DBMSs provide robust *grouping* functions for operating on related rows. Return to https://sqlime.org/#deta:32lpfoo57r8g and create a small table of hypothetical marathon times.

```
CREATE TABLE IF NOT EXISTS Marathon (rn TEXT UNIQUE,
  time INTEGER,
  gender TEXT CHECK( gender IN ('M', 'F') ));

INSERT OR IGNORE INTO Marathon (rn, time, gender) VALUES
  ('Kyle', 2*60*60 + 14*60 + 22, 'M'),
  ('Hank', 2*60*60 + 10*60 + 45, 'M'),
  ('Lily', 2*60*60 + 24*60 + 47, 'F'),
  ('Emma', 2*60*60 + 22*60 + 37, 'F'),
  ('Elle', 2*60*60 + 25*60 + 16, 'F'),
  ('Fred', 2*60*60 + 6*60 + 17, 'M');

SELECT MIN(time) FROM Marathon GROUP BY (gender);
```

`MIN` is one of the *aggregate functions* in SQLite. The `GROUP BY` clause tells the DBMS to split the rows into groups on the `gender` column.

One might be tempted to find the names of our male and female champions with `SELECT rn, MIN(time) FROM Marathon GROUP BY (gender)`. This may work in some DBMSs but there is a subtle bug. It might be obvious that we want the `rn` associated with the `MIN(time)` value, but suppose we change the query to also include `MAX(time)`:

```
SELECT rn, MIN(time), MAX(time) FROM Marathon GROUP BY (gender);
```

Now it is no longer clear which `rn` the query should return. Should the DBMS return the `rn` associated with the `MIN(time)`, the `MAX(time)`, or some other `rn` from the group?

The solution in this particular case is to nest our `MIN(time)` aggregation as a *subquery*.

```
SELECT * FROM Marathon
  WHERE time IN (
    SELECT MIN(time) FROM Marathon GROUP BY (gender));
```

2.4 Filter, map, and reduce

SQL syntax makes it easy to write select, project, and join (SPJ) queries. SQL's grouping and aggregate functions make it possible to perform row-wise and column-wise operations. One can find comparable semantics (with different syntax) in many programming language's *filter*, *map*, and *reduce* functions.

Filter works much like the `WHERE` clause: it takes a subset of the rows, based off of a condition. In JavaScript, we might filter an array with:

```
>> ['cat', 'dog', 'fish', 'bird'].filter(v => v.includes('i'))
<- ['fish', 'bird']
```

Map performs the same function over each element of an input set, creating "mappings" to elements of an output set.

```
>> ['fish', 'bird'].map(v => v.toUpperCase())
<- ['FISH', 'BIRD']
```

Reduce, also known as *fold*, performs some operation on each element of an input set and returns an *accumulator*, which is passed again to the reduce function with the next input value. To take an array's sum, we use an initial accumulator value of 0.

```
>> 15 + 25 + 35
<- 75
>> [15,25,35].reduce((a, v) => a + v, 0)
<- 75
```

For the array's product, we use 1 for the initial accumulator value.

```
>> 15 * 25 * 35
<- 13125
>> [15,25,35].reduce((a, v) => a * v, 1)
<- 13125
```

Both filter and map can be implemented in terms of reduce.

```
>> ['cat', 'dog', 'fish', 'bird'].reduce((a,v) => {
        if (v.includes('i')) {
            a.push(v);
        }
        return a;
    }, [])
<- ['fish', 'bird']
>> ['fish', 'bird'].reduce((a,v) => {
        a.push(v.toUpperCase());
        return a;
    }, [])
<- ['FISH', 'BIRD']
```

Here, we use an empty array (`[]`) instead of a numeric identity as our initial accumulator value.

2.5 Vectorized functions and concurrency

A *vectorized function* automatically iterates over array inputs. This design is less common in traditional languages (C, Java, JavaScript) and more common in scientific programming (R, Matlab, Julia). Some examples in the R language, which one can reproduce at https://webr.r-wasm.org/latest/, are:

```
> c(1, 2, 3) + 4
[1] 5 6 7
> c(1, 2, 3) + c(4, 5, 6)
[1] 5 7 9
> sqrt(c(1, 4, 9))
[1] 1 2 3
```

Observe that the pairwise sums in `c(1, 2, 3) + c(4, 5, 6)` are independent. No sum depends on another, and therefore the computing machine can safely perform each operation in *parallel*.

Concurrency is the ability for a computing machine to perform simulataneous operations. Concurrent programming can be challenging because one *process* or *thread* (sometimes called *task* or *routine*) might interfere with another, but performance benefits often justify the additional complexity.

Some problems can be partitioned into *subproblems* which can be solved in parallel. Other problems cannot. Some encryption algorithms intentionally *chain* the output from one block into the next. One cannot calculate block n without first calculating block $n-1$, and $n-2$, and so on. The reduce operation applies to this algorithm design.

Other problems can be effortlessly partitioned into subproblems and solved quickly with a *divide-and-conquer* approach. A trivial example might be finding the minimum value in a large dataset. One can partition the dataset, find the minimum value in each partition, and then find the minimum value among those results. This process can be repeated.

Go to https://go.dev/play/p/IOwH08R_z7Z to experiment with a divide-and-conquer `minimum` function in the Go language.

```
package main

import "fmt"

func min(x, y int) int {
    if x <= y {
        return x
```

```
    }
    return y
}

func minimum(x []int) int {
    fmt.Println(x)
    n := len(x)
    switch n {
    case 1:
        return x[0]
    case 2:
        return min(x[0], x[1])
    default:
        middle := n / 2
        lower := minimum(x[:middle])
        upper := minimum(x[middle:])
        return min(lower, upper)
    }
}

func main() {
    fmt.Println(minimum([]int{610, 144, 34, 21, 2584, 55, 55}))
}
```

Click the "Run" button several times and observe that the output is completely *deterministic*. Now go to https://go.dev/play/p/Vbe7BWrwlku for a slightly modified version of the same program.

```
    default:
        middle := n / 2
        lower := make(chan int)
        upper := make(chan int)
        go func() { lower <- minimum(x[:middle]) }()
        go func() { upper <- minimum(x[middle:]) }()
        return min(<-lower, <-upper)
    }
```

This version constructs two *channels* for communication among concurrent tasks. We use the `go` keyword to create two *Goroutines* (threads in the Go language), which concurrently solve the `minimum` function over subproblems. Finally, we read the results from each channel with `<-lower` and `<-upper` and return. Click the "Run" button several times and observe that the final result is consistent, but the order of operations is not.

The computer industry has recently turned to *Graphical Processing Units* (GPU) as a fast, inexpensive, and energy-efficient method for solving highly parallelizable problems. GPUs were originally designed to draw computer graphics, which extensively use matrix and vector multiplication. These linear transformations

can be performed in parallel and GPU makers designed their products to perform many simple calculations in parallel.

2.6 Consistency, availability, and partition-tolerance (CAP) theorem

Brewer's *CAP theorem* states that a *distributed system* has at most two qualities of *consistency*, *availability*, and *partition-tolerance*. Consider a system of databases with many replicas. The replicas are consistent if they contain perfect copies of the database, and they are available only they are writable. The distributed system is partition-tolerant if all replicas remain identical, but this is impossible if one allows writes that cannot propagate into the other partition.

The CAP theorem has many practical implications on data integrity and should be considered in design methodology. One must anticipate server and network outages that would create a partition in the distributed in the system and then choose the desired behavior. Can we accept lost database writes when we reconcile after a partition is restored? Or should be accept service outages in order to protect the integrity of the database during an interruption?

A partial solution is to weaken our definition of each quality. Perhaps a system reserves certain rows or columns that are only writable by a specific database, guaranteeing that there will be no conflict if this database continues to write to those changes during a partition. A system might establish some form of confidence intervals in certain data, such as the position of a tracked aircraft with error margins, in recognition that imperfect information might still be useful. Finally, a system might use a quorum model (i.e., 3 of 5 available nodes) to preserve partial availability in the majority partition.

2.7 Discussion prompts

1. How does the CAP theorem impact intelligence and fires in relation to the command and control (C2) warfighting function (WfF)?
2. Where should unclassified data be stored and processed?
3. What are some methods to prevent conflicts among concurrent writes in a shared database?
4. What could possibly go wrong when altering database schema?

2.8 Practical exercises

1. Create a custom list in SharePoint that provides multiple views showing grouped and aggregated values.

2. Given a noisy dataset, identify problems in each column that could influence inclusion and exclusion criteria.
3. Implement filter and map in terms of reduce using a programming language which provides reduce.
4. Define an "embarrassingly parallel" problem and provide both examples and counterexamples.

Chapter 3

Measures of Central Tendency

3.1 Least squares method

The canonical definition of the *arithmetic mean* for a set of n numbers x is

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{x_1 + x_2 + x_3 + \cdots + x_4}{n}.$$

Where does this definition come from? Let's open a new workbook in Microsoft Excel and find out.

Leave `A1` blank. This will become our *estimate of the mean.* In fields `B1` through `B9`, enter integers 1 through 9. In fields `C1` through `C9`, enter formulas `=B1-$A$1`, `=B2-$A$1`, `=B3-$A$1`, and so on (keeping `$A$1` fixed). In fields `D1` through `D9`, enter formulas `=POWER(C1,2)`, `=POWER(C2,2)`, and so on. In field `E1`, enter formula `=SUM(D1:D9)`. Finally in field `F1`, enter the formula `=AVERAGE(B1:B9)`.

Now go to Data, What-If Analysis, Goal Seek. In the Goal Seek dialog, enter `Set cell:` to `E1`, `To value:` to `0`, and `By changing cell:` to `A1`. Click OK. The Goal Seek function runs and should produce a value in `A1` near to that in `F1`. (Goal Seek is not foolproof. Enter `10` into `A1` to nudge Excel with a hint if you get a ridiculous answer.)

We have used Goal Seek to minimize the *sum of the squared differences* between our values, x_i, and our estimate of the mean, $\bar{x}$. This *least squares method* dates back to Carl Friedrich Gauss and Adrien-Marie Legendre in the 1800s.

Squaring the errors makes the values positive, which prevents underestimates from negating overestimates. One might also consider the *absolute value* (`ABS`

in Excel) as an alternative, but there is a second reason for squaring the errors. Squaring the errors penalizes large errors more than small errors. Accepting small errors but avoiding large errors is the bias that gives the least squares method its strength.

3.2 Expected values

The term *average* can refer to *mean*, *median*, and *mode*. Mean only applies to interval and ratio data. Median is simply the middle value among ordinal, interval, and ratio data. Mode is the "commonest" (most frequent) value among nominal, ordinal, interval, and ratio data.

Average	Levels of measurement	Symbols
Mean	Interval, ratio	μ, $\bar{x}$
Median	Ordinal, interval, ratio	(None)
Mode	Nominal, ordinal, interval, ratio	(None)

All three of these *measures of central tendency* enable us to find the *expected value* in a data set, $E(x)$. Population means are assigned the symbol μ. Estimates of the population mean (the sample mean) usually use the name of the sample with a bar, such as $\bar{x}$.

Median and mode can be useful even when analyzing interval and ratio data. Consider a classroom of 10 students who are 6 years old and 1 teacher who is 50 years old. If one selects a random person in the room, what is the expected value for their age? In this case, the modal value (6) is likely a better estimate than the mean value (10).

3.3 The Four Moments

The *four moments* describe the *distribution* of values in a data set. The first moment is the mean. The second moment is *variance*, the expected squared difference of values to the mean. The third moment is *skewness*, the expected cubed difference of values to the mean. The fourth moment is *kurtosis*, the expected difference of values to the mean raised to the fourth power.

Moment	Name	Definition
1	Mean	$E(x) = \mu$
2	Variance	$E(x - \mu)^2$
3	Skewness	$E(x - \mu)^3$
4	Kurtosis	$E(x - \mu)^4$

Variance (σ^2) is calculated from the sum of the squared differences in the random variable (x) and the mean (μ).

$$\begin{aligned}\sigma^2 &= E(x-\mu)^2 \\ &= E(x^2 - 2x\mu + \mu^2) \\ &= E(x^2) - 2E(x)\mu + \mu^2 \\ &= E(x^2) - 2(\mu)\mu + \mu^2 \\ &= E(x^2) - 2\mu^2 + \mu^2 \\ &= E(x^2) - \mu^2.\end{aligned}$$

To calculate the sample variance (s^2) in practice, we use the formula

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2.$$

where n is the number of elements in x. Observe that the calculation for this statistic is similar to the least squares method in section 3.1. s^2 is the average squared distance from the mean within the data set.

The *standard deviation* is the square root of variance,

$$s = \sqrt{s^2}.$$

An *outlier* is a value that is very different from other values in the data set. Let $x = \{15, 75, 79, 10, 7, 54, 4600, 91, 45, 86\}$. One can immediately intuit that the value 4600 is substantially different from all of the other values. Outliers can be defined in terms of the mean and standard deviation, where outliers are any values greater than $\bar{x} + 2s$ or less than $\bar{x} - 2s$. We can use the `mean` and `sd` functions with `subset` in the R language at https://webr.r-wasm.org/latest/ to confirm that 4600 is an outlier.

```
> x = c(15, 75, 79, 10, 7, 54, 4600, 91, 45, 86)
> subset(x, x <= mean(x) - sd(x) | x >= mean(x) + sd(x))
[1] 4600
```

We can use skewness to detect whether the data is imbalanced (skews) above or below the mean. If skewness is negative then the left tail is longer, if skewness is positive then the right tail is longer, and if skewness is zero then the distribution is equally balanced over the mean. The Excel function for skewness is `SKEW`.

We can use kurtosis to detect if a data set contains outliers. Most mathematical texts and software use kurtosis of 3 as an "normal" reference value, but Excel subtracts 3 and thus a "normal" kurtosis is 0. The Excel function for kurtosis is `KURT`.

3.4 Exponential moving averages

An *exponential moving average* (EMA) is a weighted average that creates a bias favoring recent observations. EMAs are used in the financial sector as an implicit means of modeling stock prices with time.

EMA is defined *recursively* and parameterized with a weighting multiplier $0 < p < 1$.

$$\text{EMA}\left(x, i\right) = \begin{cases} px_i + (1-p)\text{EMA}\left(x, i-1\right), & \text{if } i > 1. \\ x_1, & \text{otherwise.} \end{cases}$$

EMA can be easily implemented in terms of reduce, as shown in JavaScript.

```
>> x = [7, 8, 9, 10, 9, 8, 7, 9, 11, 13, 15, 17]
>> p = .25
>> x.reduce((ema,v) => p * v + (1-p) * ema, x[0])
<- 12.661770105361938
```

The *base case* is at x_1 in mathematical notation but `x[0]` in JavaScript. This is a matter of convention; the first element of a list is position 1 in math, but 0 in many programming languages.

3.5 Discussion prompts

1. Is four a lot?
2. First battalion has an average ACFT score of 482 while second battalion has an average ACFT score of 491. Which is better?
3. What do we do when statistics show us something that contradicts our values? For example, suppose we discover that Soldiers of a specific demographic have much lower promotion rates than their peers.
4. Is it more important for an organization to think about variance or the 99th percentile?
5. Given a sample set [Equation], what is the estimate of the mean ([Equation]), and what is the sample variance?

3.6 Practical exercises

1. Calculate the influence that outliers have on different-sized datasets that contain outliers.
2. Calculate the exponential moving average in a small dataset.
3. Given a dataset and experimental result, identify problems caused by analyzing categorical data represented in a numeric form.

4. Given multiple datasets with identical mean and standard deviation, use kurtosis to identify the dataset with more outliers.
5. Design or implement an algorithm to incrementally calculate standard deviation, where the estimate of the sample standard deviation is updated with each additional value.

Chapter 4

Graph Theory

4.1 Vertices, edges, and paths

A *graph* (G) is a collection of *vertices* (V; also known as *nodes* or *points*) and the *edges* (E; also known as *relations* or *lines*; see section 1.11) connecting them.

$$G = \{V, E\}$$

Edges are conventionally named for the vertices they connect. For example, let $V = \{u, v\}$ and $E = \{(u, v)\}$, then $G = \{V, E\}$ is a graph with two vertices, u and v, and one edge, (u, v).

Graphs can be used to model any form of *network* where the elements of sets bear relations. Graphs can be directed, where the source and destination vertices in an edge are significant ($(u, v) \neq (v, u)$), or undirected ($(u, v) = (v, u)$).

The edges of a graph may have a *distance function* (δ; also known as *weight* and *cost*), which relates each edge with some real number. Such graphs are *weighted graphs*. For example, suppose a high-speed railroad has train stations in Paris, Brussels, and the Hague. The distance from Paris to Brussels is about 350 km and the distance from Brussels to Hague is another 180 km. The total length of the *path* from Paris to Hague via Brussels is therefore 350 km + 180 km = 530 km.

$$\begin{aligned}\delta(\text{Paris}, \text{Hague}) &= \delta(\text{Paris}, \text{Brussels}) + \delta(\text{Brussels}, \text{Hague}) \\ &= 350\,\text{km} + 180\,\text{km} \\ &= 530\,\text{km}.\end{aligned}$$

Let us quickly reproduce this result using a *graph database* named Neo4j. Go to https://console.neo4j.org and click the "Clear DB" button. Enter the below *Cypher query* into the input field and press the triangle-shaped execute button.

```
CREATE
    (paris:CITY {name:"Paris"}),
    (brussels:CITY {name:"Brussels"}),
    (hague:CITY {name:"Hague"}),
    (paris)-[:ROUTE {dist:350}]->(brussels),
    (brussels)-[:ROUTE {dist:180}]->(hague);
```

This command created three vertices and two edges. Verify and visualize this graph with the below query. Neo4j's Cypher language uses `MATCH` as its selection (σ) operator.

```
MATCH (x:CITY)
RETURN x;
```

Finally, query the database for a path of any length (`*`) connecting Paris to Hague, returning the path (`p`) and its cumulative distance. Refer to section 2.4 for a description of the reduce operation.

```
MATCH (:CITY {name:'Paris'})-[p:ROUTE*]->(:CITY {name:'Hague'})
RETURN p, REDUCE(length=0, e IN p | length + e.dist) AS distance;
```

4.2 Special cases of graphs

A *directed acyclic graph* (DAG) is a special case of a directed graph. A *cycle* (also known as a *loop*) occurs in a directed graph when there is any path from some vertex to itself. For example, let G be a directed graph where

$$G = \{\{a, b, c, d\}, \{(a, b), (b, c), (c, d), (c, a)\}\}.$$

The edges (a, b), (b, c), and (c, a) form a cycle. Vertices a, b, and c together form a *connected component*. A directed graph with one or more connected components is not a DAG, but the graph of components (where vertices of the component subgraph are merged into a "super vertex") is itself a DAG.

A *tree* is another special case of an acyclic graph, typically in undirected graphs. One's ancestoral family tree is an instance of a tree; without a time machine, it is impossible to one to form a loop with an ancestor.

4.3 Representation

Graphs are modeled using either an *adjacency list* or an *adjacency matrix*. An adjacency list might look like

$$\begin{aligned} \text{adj}(\text{Paris}) &= \{\text{Brussels}\} \\ \text{adj}(\text{Brussels}) &= \{\text{Paris}, \text{Hague}\} \\ \text{adj}(\text{Hague}) &= \{\text{Brussels}\}. \end{aligned}$$

A separate data structure would be necessary to represent edge weights. *Dictionary* types, such as `dict` in Python and `map` in Go, can be convenient implementations for both adjacency lists and edge weight functions.

An adjacency matrix represents edges with weights in a single data structure, such as

$$E = \begin{array}{c} \\ \text{Paris} \\ \text{Brussels} \\ \text{Hague} \end{array} \begin{array}{c} \begin{array}{ccc} \text{Paris} & \text{Brussels} & \text{Hague} \end{array} \\ \begin{pmatrix} 0 & 350 & \infty \\ 350 & 0 & 180 \\ \infty & 180 & 0 \end{pmatrix} \end{array}.$$

In this example, the distance of a vertex to itself is defined as zero,

$$\delta(u, u) = 0$$

and the distance between vertices is considered infinite if those vertices are not directly connected by an edge.

$$(u, v) \notin E \implies \delta(u, v) = \infty$$

4.4 Search algorithms

Imagine a video game where the player searches the kingdom for treasure. The player has no knowledge of where the treasure might be, so from a starting point they fully explore the forest, mountains, and sea. Both the starting point and the sea connect to opposite sides of the city. Upon entering the city from the sea-side, the player explores the city and discovers the treasure. This is an example of a *depth-first search* (DFS).

Go to https://go.dev/play/p/AuH2qOgSG-c to run the following DFS implementation, written in Go. This implementation uses a *recursive* definition of the DFS function (the DFS function invokes itself as it explores the graph). The function uses an external data structure (`quest`) to identify which vertices have already been discovered. Upon successful search, the function prints its position in the graph as the recursive calls "unwind."

```
package main

import "fmt"

var g = map[string][]string{
    "start":     []string{"forest", "mountains", "sea", "city"},
    "forest":    []string{"start", "mountains", "desert", "cave"},
    "mountains": []string{"start", "forest", "glacier"},
```

```
    "desert":    []string{"forest"},
    "cave":      []string{"forest", "inferno"},
    "inferno":   []string{"cave"},
    "glacier":   []string{"mountains"},
    "sea":       []string{"start", "beach"},
    "beach":     []string{"sea", "city"},
    "city":      []string{"beach", "start", "castle"},
    "castle":    []string{"city", "treasure"},
    "treasure":  []string{"castle"}}

var quest map[string]string = make(map[string]string)

func dfs(src, dst string) bool {
    quest[src] = "discovered"

    if src == dst {
        fmt.Printf("Discovered %s:", dst)
        return true
    }

    for _, neighbor := range g[src] {
        if quest[neighbor] != "discovered" {
            if dfs(neighbor, dst) == true {
                fmt.Printf(" %s", src)
                return true
            }
        }
    }

    return false
}

func main() {
    dfs("start", "treasure")
    fmt.Println()
}
```

The program should output `Discovered treasure: castle city beach sea start`. DFS successfully discovers the treasure, but we have no guarantee that this algorithm will find the *optimal* (shortest) path.

Imagine the protagonist of our hypothetical adventure game was not a lone wanderer, but rather a field marshall commanding a large army. This army explores one region at a time, holding each area as adjacent units proceed into their respective area. The army incrementally expands the radius of the search *frontier*. Once one unit discovers the treasure, we are certain that no shorter path was possible thanks to an *invariant* in our search algorithm.

Maintaining an invariant is essential for *mathematical induction*, where we establish that some *predicate* P is true for the *base case* $P(0)$ and that $P(k)$ implies $P(k+1)$ and therefore $P(n)$ is true for all $n > 0$. The proof for the correctness of a BFS follows:

1. Along the frontier of radius $r = 0$, the BFS algorithm on graph G has not discovered a path from u to v. $\delta(u, v)$ is therefore at *closest* $r = 1$.
2. At $r = 1$, BFS has not found v and therefore $2 \leq \delta(u, v)$.
3. At $r = 2$, BFS has not found v and therefore $3 \leq \delta(u, v)$.
4. ⋮
5. At $r = k$, BFS has not found v and therefore $k + 1 \leq \delta(u, v)$.
6. ⋮
7. At $r = n$, BFS has located v and therefore $n = \delta(u, v)$. □

Gp to https://go.dev/play/p/yMIcmcsK_V9 and run the following BFS implementation, written in Go. This implementation uses an *iterative* BFS function. The BFS function does not invoke itself. Instead, the procedure adds unexplored vertices to a queue and records the "parent" of each vertex. We "unwind" the resulting tree from child to parent nodes to construct the shortest path.

```
func bfs(src, dst string) map[string]string {
    parent := map[string]string{src: src}
    queue := []string{src}
    for len(queue) > 0 {
        position := queue[0]
        queue = queue[1:]
        if position == dst {
            break
        }
        for _, neighbor := range g[position] {
            if _, ok := parent[neighbor]; !ok {
                parent[neighbor] = position
                queue = append(queue, neighbor)
            }
        }
    }
    return parent
}

func main() {
    tree := bfs("start", "treasure")
    fmt.Printf("Discovered treasure:")
    position := "treasure"
    for position != tree[position] {
        position = tree[position]
        fmt.Printf(" %s", position)
    }
```

```
    fmt.Println()
}
```

This program should output `Discovered treasure: castle city start`. BFS finds the shortest path between two vertices by *hop count*, but it does not consider edge weights. *Dijkstra's algorithm* performs a breadth-first traversal ordered by cumulative path cost.

Dijkstra's algorithm uses a *priority queue* to visit nodes from shortest to longest path. For this reason, Dijkstra's algorithm is also known as the *shortest-path first* (SPF). Like BFS, Dikjstra's algorithm is a *greedy algorithm* that discovered a globally optimal solution by repeatedly making locally optimal decisions. Let us turn to the Python language to demonstrate Dijkstra's algorithm on our same treasure-hunting graph, this time with edge weights. Run this program at https://www.python.org/shell/. (Note: the Python language is very picky about the use of tabs.

```
from heapq import *

g = dict(
    [
        ("start", ["forest", "mountains", "sea", "city"]),
        ("forest", ["start", "mountains", "desert", "cave"]),
        ("mountains", ["start", "forest", "glacier"]),
        ("desert", ["forest"]),
        ("cave", ["forest", "inferno"]),
        ("inferno", ["cave"]),
        ("glacier", ["mountains"]),
        ("sea", ["start", "beach"]),
        ("beach", ["sea", "city"]),
        ("city", ["beach", "start", "castle"]),
        ("castle", ["city", "treasure"]),
        ("treasure", ["castle"]),
    ]
)

w = dict(
    [
        (("start", "forest"), 70),
        (("start", "mountains"), 60),
        (("start", "sea"), 54),
        (("start", "city"), 81),
        (("forest", "start"), 42),
        (("forest", "mountains"), 51),
        (("forest", "desert"), 56),
        (("forest", "cave"), 63),
        (("mountains", "start"), 71),
```

```
        (("mountains", "forest"), 38),
        (("mountains", "glacier"), 72),
        (("desert", "forest"), 93),
        (("cave", "forest"), 19),
        (("cave", "inferno"), 17),
        (("inferno", "cave"), 71),
        (("glacier", "mountains"), 25),
        (("sea", "start"), 49),
        (("sea", "beach"), 88),
        (("beach", "sea"), 79),
        (("beach", "city"), 29),
        (("city", "beach"), 30),
        (("city", "start"), 33),
        (("city", "castle"), 36),
        (("castle", "city"), 39),
        (("castle", "treasure"), 76),
        (("treasure", "castle"), 76),
    ]
)


def dijkstra(src, dst):
    explored = set()
    distance = dict()
    distance[src] = 0
    queue = []
    heappush(queue, (0, src))
    while queue:
        _, current = heappop(queue)
        if current == dst:
            print("Path found": distance[dst])
            return distance[dst]
        if current in explored:
            continue
        explored.add(current)
        for neighbor in g[current]:
            if neighbor not in explored:
                d = distance[current] + w[(current, neighbor)]
                distance[neighbor] = d
                heappush(queue, (d, neighbor))
    print("No path found")

dijkstra("start", "treasure")
```

4.5 Dijkstra's algorithm

4.6 A*

4.7 Computational complexity and Big-[Equation]

4.8 Power Law Distribution

4.9 Discussion prompts

A graph can be represented with an adjacency list or a matrix. What are the advantages and disadvantages of each approach?

What algorithm can be used to solve the "seven ways to Kevin Bacon" problem?

Is a Gantt chart a graph? How can one find the critical path of a project if represented as a graph?

Which is bigger, [Equation] or [Equation]?

4.10 Practical exercises

Compare two different heuristic functions in a provided A* informed search implementation on the 8-piece puzzle problem.

Convert currency exchange rates from multiplication to addition using a logarithm, then prove that infinite arbitration is impossible given a set of exchange rates and Bellman-Ford implementation.

Define a topological sorting and relate it to a workplace problem.

Define the Traveling Salesman Problem (TSP) and explain the computational difficulty of this problem.

Determine the minimum paving needed to fully connect a tent complex using a list of coordinates and a Prim or Kruskal implementation.

Simulate an infection model in a dense social graph where edge weights represent probability of infection.