Contents

CHAPTER 1

The Multivariate Normal Distribution

Let's say that you are gathering statistics on a species of snail. You have found 89 snails. Every snail has been weighted and had its shell measured. You have created a table like this:

Weight	Diameter
4.4769 grams	2.2692 cm
5.4755 grams	2.1973 cm
4.1183 grams	2.52928 cm
...	...
3.0522 grams	1.7822 cm

You have been told that for any particular species of snail, the weight and shell diameter are typically normally distributed. So, you compute the mean of each:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Your snails have a mean mass off 4.25 grams and a mean shell diameter of 2.35 centimeters. Now you know where the center of each normal distribution will be.

To know how wide each normal distribution is, you need to compute the variance:

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

(Reminder: The standard deviation is the square root of σ^2)

Plotting this data is shown in Figure 1.1.

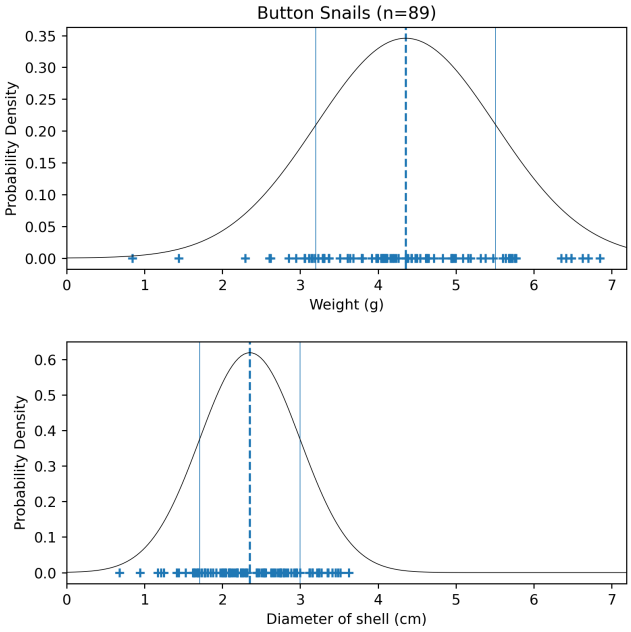


Figure 1.1: Probability density functions of weight and diameter.

Nice! However, then you think: ”Hmm. Maybe the mass and the diameter of the shell are related. It seems like a heavy snail might tend to have a larger shell.” So, you make a scatter plot:

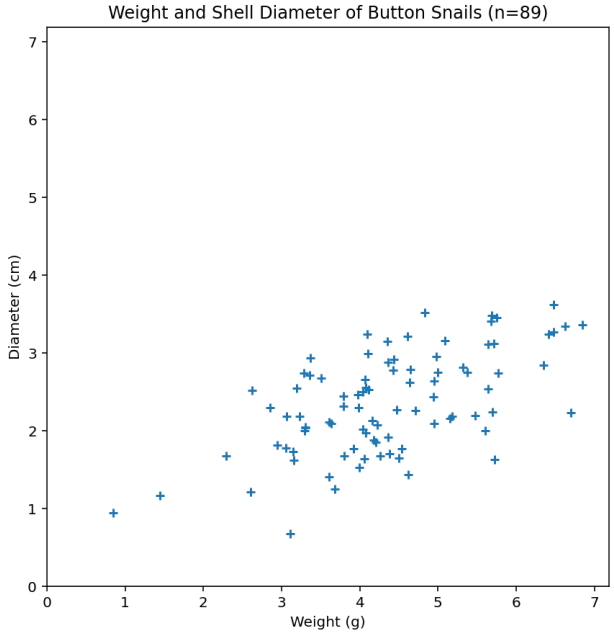


Figure 1.2: A scatter plot with weight of the snail horizontally and diamter of the shell on the y-axis.

Sure enough, there is some correlation between the mass of the snail and the diameter of its shell.

There is a form of the normal distribution that can deal with multiple variables and their correlations. This is known as the *multivariate normal* or the *Gaussian distribution*.

In a multivariate normal distribution, the mean (often denoted μ) is a vector containing the mean of every variable. For your snails, $\mu = [4.25, 2.35]$.

Just as in the single-variable normal distribution, measurements near the mean are what you are most likely to observe. In a single-variable normal distribution, the standard deviation tells us how fast that likelihood falls off as we move away from the mean. In multivariate normal distributions, we need something a little more expressive: a matrix.

1.1 The Covariance Matrix

We can think of the data as a list of vectors. For example, if we have n snails and d properties that we have measured, we would get a list like this:

$$\begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,d} \\ x_{2,1} & x_{2,2} & \dots & x_{2,d} \\ \dots & \dots & & \dots \\ x_{n,1} & x_{n,2} & \dots & x_{b,d} \end{bmatrix}$$

Each row represents one snail. Each column represents one property.

Note that to make μ (the mean vector), you just take the mean of each column of this data matrix. (For convenience, we will use μ_i to refer to the mean of column i.)

We usually use Σ (the uppercase Greek sigma) to represent a covariance matrix. Σ is a $d \times d$ matrix. The entries on the diagonal of Σ are just the variance of each property. So, we can caluculate the entry at row j, column j like this:

$$\Sigma_{j,j} = \frac{\sum_{i=1}^{n}(x_{i,j} - \mu_j)^2}{n}$$

(Yes, we are using Σ to represent both summation and the covariance matrix here. It can be confusing, but you will be able to tell from context how it is being used.)

The other entries, $\Sigma_{j,k}$ where $j \neq k$, are the *covariance* between property j and property k. If the covariance is a positive number, property j and property k are correlated. For example, in your snails, weight and diameter have a positive covariance: If the snail is

heavier than average, it tends to have a diameter that is larger than average.

If the covariance is a negative number, when property j is greater than average, property k tends to be less than average. For example, the number of times a restaurant mops the floor in a week has a negative covariance with how much bacteria lives on the floor.

How do we compute the covariance?

$$\Sigma_{j,k} = \frac{\sum_{i=1}^{n}(x_{i,j} - \mu_j)(x_{i,k} - \mu_k)}{n}$$

Note that $\Sigma_{j,k} = \Sigma_{k,j}$, so the matrix is symmetric.

1.1.1 Computing Mean and Covariance in Python

If you have a numpy array where each row represents one sample and each column represents one property, it is really easy to compute μ and Σ:

```
# Read in the data matrix, each row represents one snail
snails = ...

# Compute mu
mean_vector = snails.mean(axis=0)
print(f"Mean = {mean_vector}")

# Compute Sigma
covariance_matrix = np.cov(snails, rowvar=False)
print(f"Covariance = {covariance_matrix}")
```

1.2 Multivariate Normal Probability Density

Now that we have good estimates of μ and Σ, how can we compute the probability density?

When you were working with just one variable, x, you computed the probability density using the mean μ and the variance σ^2 like this:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The probability density function of a d-dimensional multivariate normal distribution with a mean of μ and a covariance matrix of Σ is given by:

$$p(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x}-\boldsymbol{\mu})\right)$$

If we draw lines to show where $\boldsymbol{\mu}$ is and contours to show lines of equal probability density, we get a plot shown in Figure 1.3.

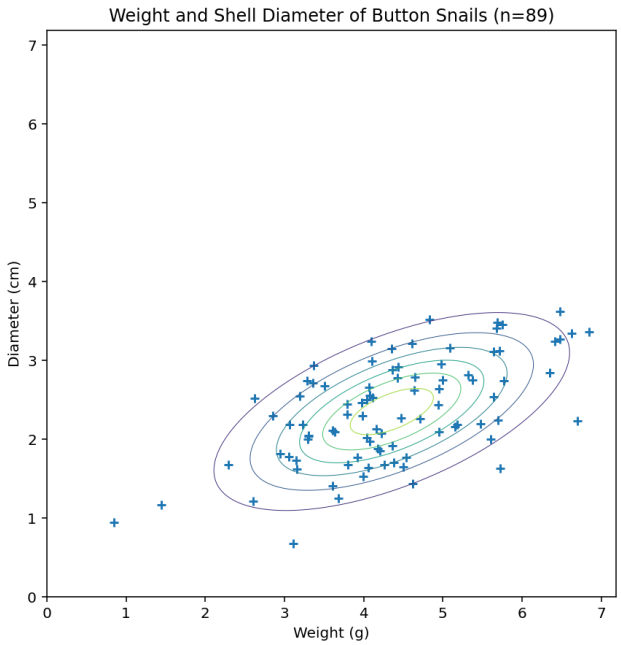


Figure 1.3: Contour plot of the snail data.

Or, we can do a 3D plot, as shown in Figure 1.4.

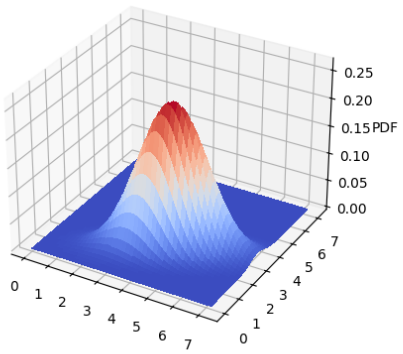


Figure 1.4: 3D plot of the probability density function.

A probability density always integrates to 1.0, so the volume under the surface must be 1.0.

1.2.1 Multivariate Normal Probability Density in Python

The scipy library has a class that represents the multivariate normal. Here is how you could compute the probability density for a particular snail:

```
import numpy as np
from scipy.stats import multivariate_normal

...Compute mean_vector and covariance_matrix...

# Get the probability density at [3 grams, 2 cm]
x = np.array([3.0, 2.0])
pd = multivariate_normal.pdf(x, mean=mean_vector, cov=covariance_matrix)
print(f"The probability density at {x} is {pd}")
```

Alternatively, maybe you would like to generate weights and diameters for a fictional population of 700 snails:

```
new_snails = multivariate_normal.rvs(mean=mean_vector, cov=covariance_matrix, size=700)
print(f"My fictional snails:\n{new_snails}")
```

Chapter 2

Numerical Double Integration

In an earlier chapter, we gave an example of the multivariate normal distribution: the mass and shell diameter of a population of snails.

Starting with a table of measurements, we estimated that the mean vector μ was $\begin{bmatrix}4.25\text{g} & 2.35\text{cm}\end{bmatrix}$.

We then computed the covariance matrix:

$$\Sigma = \begin{bmatrix}1.33 & 0.443 \\ 0.443 & 0.416\end{bmatrix}$$

Plotting the data points, the mean, and the equal-density contours looked like this:

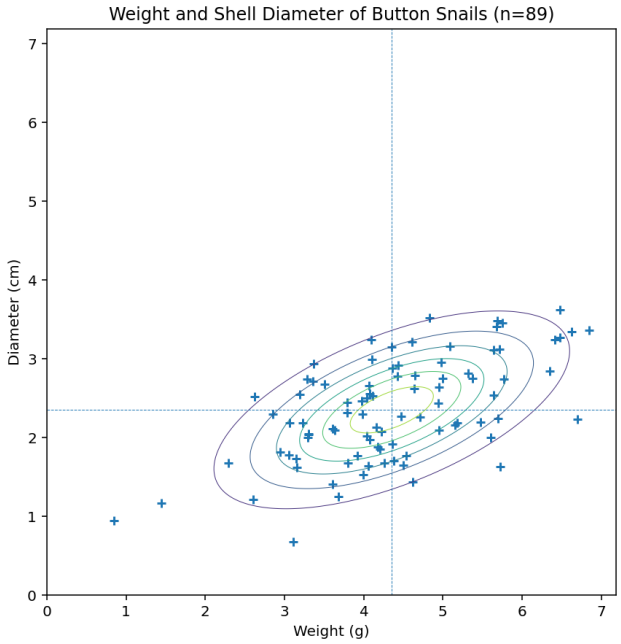


Figure 2.1: Equal density contours plotted with the snail data.

We have a great formula for computing the probability density for any mass/diameter combination:

$$p(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x}-\boldsymbol{\mu})\right)$$

Can you answer the following question? "If I pick a random snail off the floor of the ocean, what is the probability that its mass is between 3 and 4 grams and its diameter is between 1.5 and 2.0 centimeters?"

If we think of the probability density as a surface, this question is really "What is the volume under the surface in the rectangular patch $3 <= x_1 <= 4$ and $1.5 <= x_2 <= 2.0$?" Which you should recognize as a double integration problem:

$$P = \int_{1.5}^{2.0} \int_{3}^{4} \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x}-\boldsymbol{\mu})\right) dx_1 dx_2$$

Sadly, however, no one has ever been able to find the antiderivative of the multivariable probability density function, so no one can solve this problem.

Instead, we use Reimmann sums to find an approximate solution. This is known as *numerical integration*.

(After spending so much time learning the techniques for integration, it may be disappointing to hear this: For a lot of real-world problems, there is no way to find an antiderivative, so we end up doing numerical integration much more often than most people realize.)

2.1 Reimann Sums on 2-Dimensional Domains

When doing Reimann sums on function that takes a single real number, you summed the area of the rectangles under the function to approximate the integral:

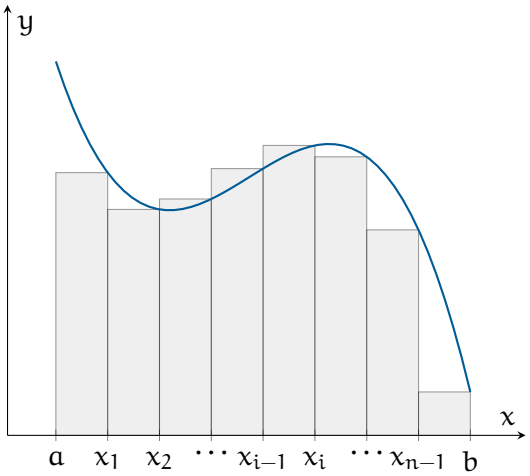


Figure 2.2: A representative function divided into n rectangles of equal width, with rectangle height determined by the right endpoint of the subinterval

Here you are finding the volume under a function that takes two variables (the probability density is based on the mass and diameter of the shell).

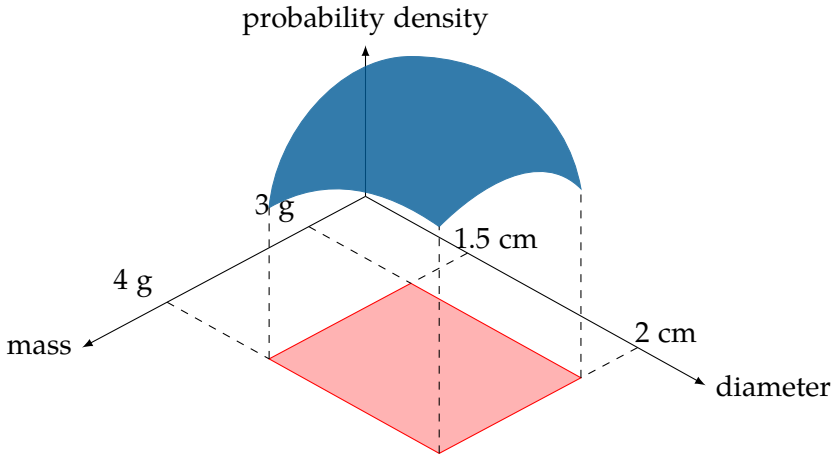


Figure 2.3: The probability is the volume under a surface for a region

To do the Reimann sum, we can break the range of the mass into n_1 equally sized intervals and the range of the diameter into n_2 equally sized intervals. For the diagram, we will just break each range into three, but you will get more accurate estimate as the intervals get smaller.

Now you will be calculating the volume of rectangular solids and summing up those volumes. Here are a few of the rectangular volumes:

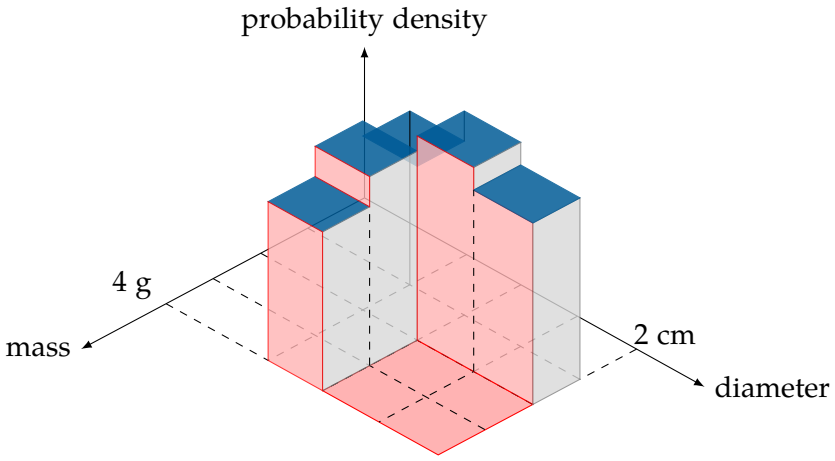


Figure 2.4: Reimann Sum

2.2 Numerical Integration in Python

Here's the code:

```
import numpy as np
from scipy.stats import multivariate_normal

# pip3 install scipy
weight_lower_limit = 3.0
weight_upper_limit = 4.0
weight_slices = 100

diameter_lower_limit = 1.5
diameter_upper_limit = 2.0
diameter_slices = 100

# What's the average weight and diameter
mean_vector = np.array([4.35559489,2.3526593 ])
print(f"Mean [weight, diameter] = {mean_vector}")

# Do heavier snails tend to have bigger shells?
covariance_matrix = np.array([[1.33099714,0.44309754],
 [0.44309754,0.41603925]])
print(f"Covariance = \n{covariance_matrix}")

# Create a multivariate normal distribution
rv = multivariate_normal(mean_vector, covariance_matrix)

delta_weight = (weight_upper_limit - weight_lower_limit) / weight_slices
delta_diameter = (diameter_upper_limit - diameter_lower_limit) / diameter_slices

sum = 0.0
```

```
# Step though each different weight
for i in range(weight_slices):
    # What is the weight in the middle of this slice?
    current_weight = weight_lower_limit + (i + 0.5) * delta_weight

    for j in range(diameter_slices):
        # What is the diameter in the middle of this slice?
        current_diameter = diameter_lower_limit + (j + 0.5) * delta_diameter

        # What is the probability density there?
        prob_density = rv.pdf((current_weight, current_diameter))

        # What is the volume under that for this tiny square
        sum += prob_density * delta_weight * delta_diameter

print(
    f"\nThe probability that the weight is between {weight_lower_limit} and {weight_upper_limit}"
)
print(
    f"and that the diameter is between {diameter_lower_limit} and {diameter_upper_limit}"
)
print(f"is about {sum * 100.0:.4f}%")
```

This should get you the following output:

```
> python3 num_integration.py
Mean [weight, diameter] = [4.35559489 2.3526593 ]
Covariance =
[[1.33099714 0.44309754]
 [0.44309754 0.41603925]]

The probability that the weight is between 3.0 and 4.0
and that the diameter is between 1.5 and 2.0
is about 7.8316%
```

Chapter 3

Sets and Logic

The use of math usually falls into two categories:

- *Developing mathematical tools that let us make better predictions.* This is how engineers and scientists use math. It is usually referred to as *applied math*.
- *Creating interesting statements and proving them to be true or false.* This is known as *pure math*.

Many mathematical ideas start out as pure math, and eventually become useful. For example, the field of number theory is devoted to proving things about prime numbers. The mathematicians who created number theory were certain that it could never be used for any practical purpose. After a century or two, number theory was used as the basis of most cryptography systems.

Conversely, some ideas start out as a "rule of thumb" that engineers use, and are eventually rigorously defined and proven.

This course tends to emphasize applied math, but you should know something about the tools of pure math.

You can think of all the mathematical proofs as a tree. Each proof proves some statement true. To do this, the proof uses logic and statements that were proven true by other truths. So, the tree is built from the bottom up. However, the tree has to have a bottom. At the very bottom of the tree are some statements that we just accept as true without proof. These are known as *axioms*.

All of modern mathematics can be built from:

- A short list of axioms.
- A few rules of logic.

There have been several efforts to codify a small but complete axiomatic system. The most popular one is known as *ZFC*. "Z" is for the Ernst Zermelo, who did most of the work. "F" is for Abraham Fraenkel, who tidied up a couple of things. "C" is for The Axiom of Choice. As a community, mathematicians debate whether the Axiom of Choice should be an axiom; we get a couple of strange results if we include it in the system. If we do not, there are a few obviously useful ideas that we can't prove true.

ZFC has 10 axioms. We simply accept these 10 statements as true, and all the proofs of modern mathematics can be extrapolated from them. The 10 axioms are all stated in terms of sets.

3.1 Sets

A *set* is a collection. For example, you might talk about the set of odd numbers greater than 5, or the set of all protons in the universe.

We have a notation for sets. For example, here is how we define S to be the set containing 1, 2, and 3:

$$S = \{1, 2, 3\}$$

We say that 1, 2, and 3 are *elements* of the set S. (Sometimes we will also use the word "member")

If you want to say "2 is an element of the set S" in mathematical notation, it is done like this:

$$2 \in S$$

If you want to say "5 is *not* an element of the set S", it looks like this:

$$5 \notin S$$

We have notation for a few sets that we use all the time:

Set	Symbol
The empty set	$\emptyset$
Natural numbers	$\mathbb{N}$
Integers	$\mathbb{Z}$
Rational numbers	$\mathbb{Q}$
Real numbers	$\mathbb{R}$
Complex numbers	$\mathbb{C}$

The empty set is the set that contains nothing. It is also sometimes called *the null set*.

Often, when we define a set, we start with one of these big sets and say "The set I'm talking about is the members of the big set, but only the one for which this statement is

true". For example, if you wanted to talk about all the integers greater than or equal to -5, you could do it like this:

$$A = \{x \in \mathbb{Z} \mid x \geq -5\}$$

When you read this aloud, you say "A is the set of integers x where x is greater than or equal to negative 5."

3.1.1 And and Or

Sometimes you need the members to satisfy two conditions; for this, we use "and":

$$A = \{x \in \mathbb{Z} \mid x > -5 \text{ and } x < 100\}$$

This is the set of integers that are greater than -5 *and* less than 100. In this book, we usually just write "and", but if you do a large amount of set and logic work, you will use the symbol $\wedge$:

$$A = \{x \in \mathbb{Z} \mid (x > -5) \wedge (x < 100)\}$$

Sometimes, you want a set that satisfies at least one of two conditions. For this, you use "or":

$$A = \{x \in \mathbb{Z} \mid x < -5 \text{ or } x > 100\}$$

These are the number that are less than -5 or greater than 100. Once again, there is a symbol for this:

$$A = \{x \in \mathbb{Z} \mid (x < -5) \vee (x > 100)\}$$

3.1.2 How simple are sets?

Sets are so simple that some questions just don't make any sense:

- "What is the first item in the set?" makes no sense to a mathematician. Sets have no

order.

- "How many times does the number 6 appear in the set?" makes no sense. 6 is a member, or it is not.

3.1.3 Subsets

If every member of set A is also in set B, we say that "A is a subset of B."

For example, if $A = \{1, 4, 5\}$ and $B = \{1, 2, 3, 4, 5, 6\}$, then A is a subset of B. There is a symbol for this:

$$A \subseteq B$$

Remember the table of commonly used sets? We can arrange them as subsets of each other:

$$\emptyset \subseteq \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$$

Note that that subsets have the transitive property: $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q}$ thus $\mathbb{N} \subseteq \mathbb{Q}$

Note that if A and B have the same elements, $A \subseteq B$ *and* $B \subseteq A$. In this case, we say that the two sets are equal.

We also have a symbol for "is not a subset of": $A \not\subset B$

3.1.4 Union and Intersection of Sets

If you have two sets A and B, you might want to say "Let C be the set containing element that are in *either* A or B." We say that C is the *union* of A and B. There is notation for this too:

$$C = A \cup B$$

For example, if $A = \{1, 3, 4, 9\}$ and $B = \{3, 4, 5, 6, 7, 8\}$, then $A \cup B = \{1, 3, 4, 5, 6, 7, 8, 9\}$.

You also want to say "Let C be the set containing elements that are in *both* A and B." We say that C is the *intersection* of A and B. There is notation for this too:

$$C = A \cap B$$

For example, if $A = \{1, 3, 4, 9\}$ and $B = \{3, 4, 5, 6, 7, 8\}$, then $A \cap B = \{3, 4\}$.

3.1.5 Venn Diagrams

When discussing sets, it is often helpful to have a Venn diagram to look at. Venn diagrams represent sets as circles. For example, the sets A and B above could look like this:

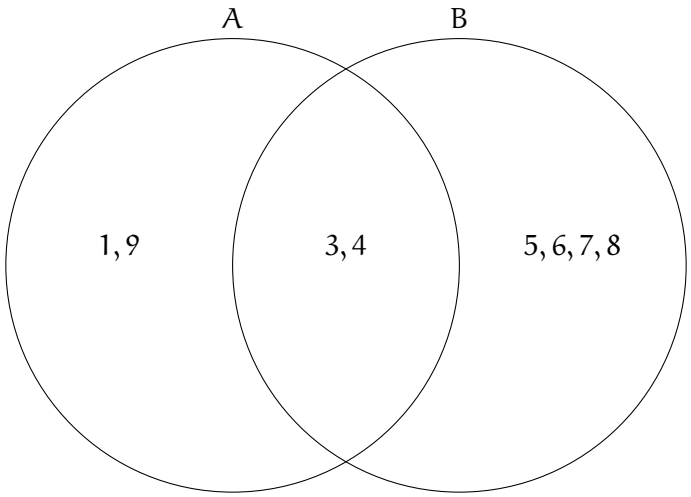


Figure 3.1: A Venn Diagram of the set of numbers $\{1, 9, 3, 4, 5, 6, 7, 8\}$

It makes it easy to see that A and B have a non-empty intersection, but they are not subsets of each other.

Often we won't even show the individual elements. For example, in the universe of all polygons, some rectangles are squares. Here's the Venn diagram:

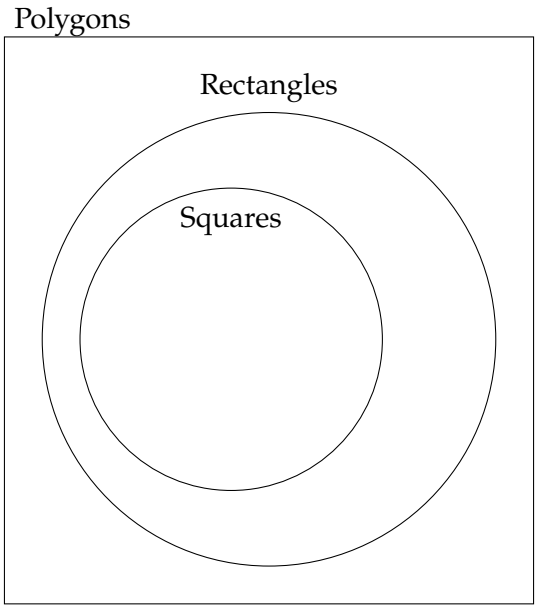


Figure 3.2: A Venn Diagram of polygons, rectangles, and squares.

As the combinations get more complex, we sometimes use shading to indicate what part we are talking about. For example, imagine we wanted all the rectangles with area greater than 5.0 that are not squares. The diagram might look like this:

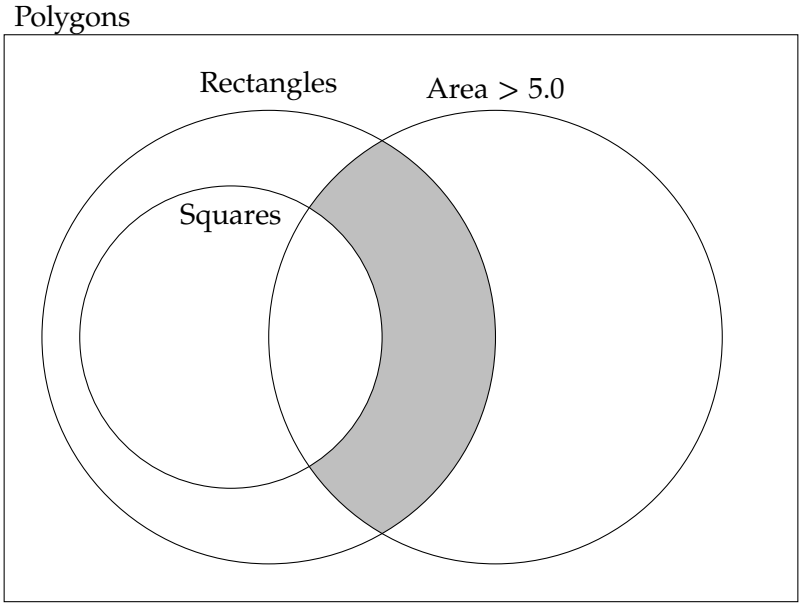


Figure 3.3: A Venn Diagram of the polygons, rectangle, and square number, and rectangles that have an area greater than 5.0 that are not squares.

3.2 Logic

We use a lot of logic in set theory. For example, the shaded region above represents all the polygons for which all the following are true:

- It is a rectangle.
- It is *not* a square.
- It has an area greater than 5.0.

3.3 Implies

In logic, we will often say "a implies b". That means "If the statement a is true, the statement b is also true." For example: "p is a square" implies "p is a rectangle".

There is notation for this: an arrow in the direction of the implication.

$$p \text{ is a square} \implies p \text{ is a rectangle}$$

Notice that implication has a direction: "p is a rectangle" does *not* imply "p is a square".

Implications can be chained together: If $A \implies B$ and $B \implies C$, then $A \implies C$.

3.4 If and Only If

If the implication goes both ways, we use "if and only if". This means the two conditions are equivalent. For example: "n is even if and only if there exists an integer m such that $2m = n$".

There is a notation for this too:

$$p \text{ is even} \iff \text{there exists an integer } m \text{ such that } 2m = n$$

There is even notation for "there exists". It is a backwards capital E:

$$p \text{ is even} \iff \exists\, m \in \mathbb{Z} \text{ such that } 2m = n$$

3.5 Not

The not operation flips the truth of an expression:

- If a is true, not(a) is false.
- If a is false, not(a) is true.

We sometimes talk about "notting" or "negating" a value. We won't use it much, but there is a symbol for this: $\neg$.

We might create a *logic table* for negation that shows all the possible values and their negation:

A	$\neg$A
F	T
T	F

This table says "If A is false, $\neg$A is true. If A is true, $\neg$A is false."

Most logic tables are for operations that take more than one input. For example, this logic table shows the values for and-ing and or-ing:

A	B	A and B	A or B
F	F	F	F
F	T	F	T
T	F	F	T
T	T	T	T

Notice that we have to enumerate all possible combinations of the inputs of A and B.

When a variable like A can only take two possible values, we say it is a *boolean* variable. (George Bool did important work in this area.)

Exercise 1 Logic Table

Working Space

Make a logic table that enumerates all possible combinations of boolean variables A and B and shows the value of the two following expressions:

- $\neg$ (A or B)
- ($\neg$A) and ($\neg$B)

Answer on Page 61

3.6 Cardinality

Informally, the *cardinality* of a set is the number of elements it contains. So, $\{1, 3, 5\}$ has a cardinality of 3. The null set has a cardinality of zero.

Things get a little trickier if a set is infinite. We say two infinite sets A and B have the same cardinality if there is some mapping that pairs every member of A with a member of B and mapping that pairs every member of B with a member of A.

For example, the set of all natural numbers is $\mathbb{N} = \{1, 2, 3, 4, \ldots\}$. The set of all even numbers is $\{2, 4, 6, 8, \ldots\}$. These have the same cardinality because we can pair each natural number n with an even number $2n$.

3.7 Complement of a Set

Most sets exist in a particular universe, for example you might talk about the even numbers as a set in the integers. You can then talk about the set's *complement*: the set of everthing else. For example, the complement of the even numbers (inside the integers) is the odd numbers.

If you have a set A, its complement is usually denoted by A'.

See the Venn Digram shown in Figure 3.4.

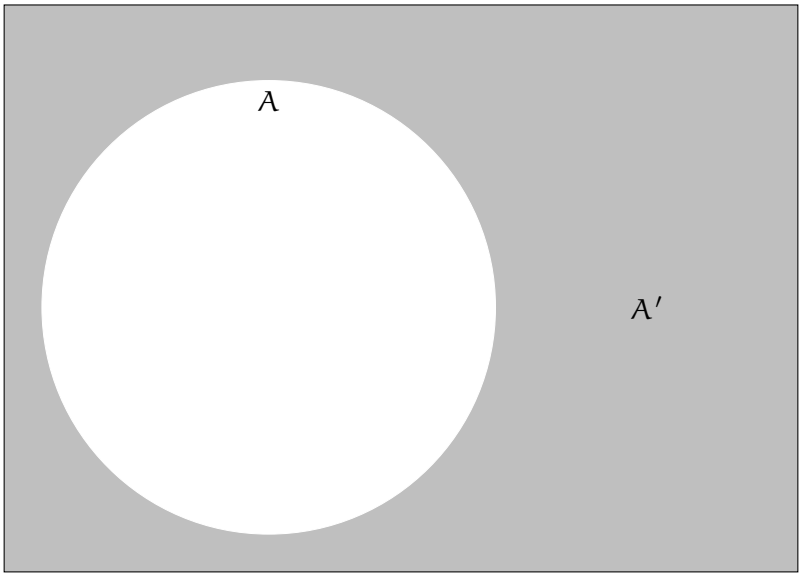


Figure 3.4: The complement of A is A'.

Outside of logic, the concept of a set

3.8 Subtracting Sets

If you have sets $A = \{1, 2, 3, 4\}$ and $B = \{1, 4\}$, it makes sense to subtract B from A by removing 1 and 4 from A.

If A and B are sets, we define $A - B$ to be $A \cap B'$. Take a second to look at this diagram and convince yourself that the white region represents $A - B$ and that it is the same as $A \cap B'$.

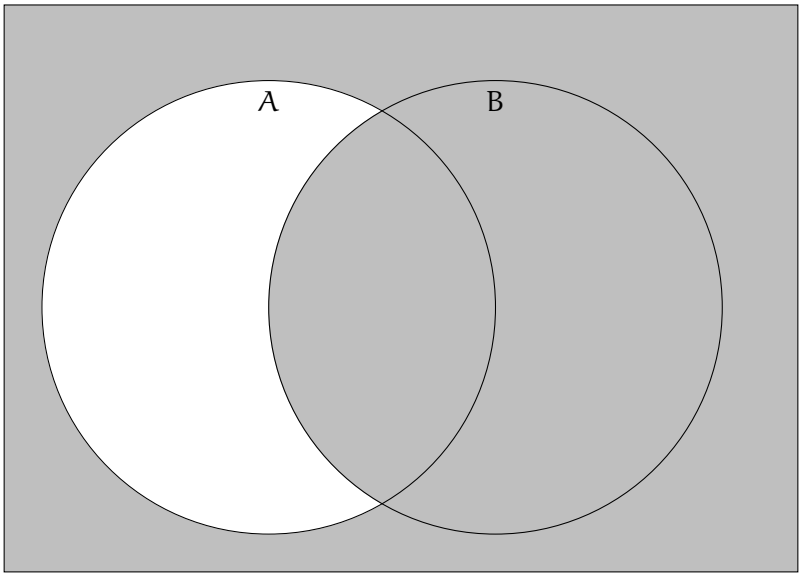


Figure 3.5: The subtraction of sets A and B.

3.9 Power Sets

It is not uncommon to have a set whose elements are also sets. For example, you might have the set that contains the following two sets: $\{1, 2, 3\}$ and $\{2, 3, 4\}$. You might write it like this: $\{\{1, 2, 3\}, \{2, 3, 4\}\}$. (Note that this set has a cardinality of 2 — it has two members that are sets.)

Given any set A, you can construct its *power set*, which is the set of all subsets of A. For example, if you have a set $\{1, 2, 3\}$, its power set is $\{\{1, 2, 3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1\}, \{2\}, \{3\}, \emptyset\}$.

If a set has n elements, its power set has 2^n elements. Note that the last element must be the empty set, because there are no subsets of an empty set. Recall that order does not matter in sets, so $\{1, 2\}$ and $\{2, 1\}$ are considered the same.

3.10 Booleans in Python

In Python, we can have variables hold boolean values: `True` and `False`. We also have operators: `not`, `and`, and `or`.

For example, you could find out what the expression "`a and not b`" is if both variables are false like this:

```
a = False
b = False
```

```
result = a or not b
print(f"a={a}, b={b}, a or not b = {result}")
```

This would print out:

```
a=False, b=False, a or not b = True
```

What if you wanted to try all possible values for `a` and `b`? You could use `itertools`.

```
import itertools

all_combos = itertools.product([False, True], repeat=2)
for (a, b) in all_combos:
    result = a or not b
    print(f"a={a}, b={b}: a or not b = {result}")
```

Type it in and run it. You should get the whole logic table:

```
a=False, b=False: a or not b = True
a=False, b=True: a or not b = False
a=True, b=False: a or not b = True
a=True, b=True: a or not b = True
```

If you had three inputs into the expression, your truth table would have eight entries. For example, if you wanted to know the truth table for $a and not (b and c)$, here is the code:

```
all_combos = itertools.product([False, True], repeat=3)
for (a, b, c) in all_combos:
    result = a and not(b and c)
    print(f"a={a}, b={b}, c={c}: a and not (b and c) = {result}")
```

Type it in and run it. You should get:

```
a=False, b=False, c=False: a and not (b and c) = False
a=False, b=False, c=True: a and not (b and c) = False
a=False, b=True, c=False: a and not (b and c) = False
a=False, b=True, c=True: a and not (b and c) = False
a=True, b=False, c=False: a and not (b and c) = True
a=True, b=False, c=True: a and not (b and c) = True
a=True, b=True, c=False: a and not (b and c) = True
a=True, b=True, c=True: a and not (b and c) = False
```

3.11 The Contrapositive

Here is a statement with an implication: "If it has rained in the last hour, the grass is wet."

This is *not* equivalent to "If the grass is wet, it has rained in the last hour." (After all, the sprinkler may be running.)

However, it is exactly equivalent to its *contrapositive*: "If the grass is not wet, it has not rained in the last hour."

The rule can be written using symbols:

$$(A \implies B) \iff (\neg B \implies \neg A)$$

3.12 The Distributive Property of Logic

Many ideas from integer arithmetic have analogues in boolean arithmetic. For example, there is a distributive property for booleans. These two expressions are equivalent:

- A and (B or C)
- (A and B) or (A and C)

So are these:

- A or (B and C)
- (A or B) and (A or C)

3.13 Exclusive Or

The expression "a or b" is true in any of the following conditions:

- a is True and b is False.
- a is False and b is True.
- Both a and b are True.

Sometimes engineers need a way to say "Either a or b is true, but not both." For this, we use *exclusive OR* (or XOR). You may see XOR written symbolically as $\oplus$ or just used as XOR. indexXOR

Here, then, is the logic table for XOR

A	B	XOR(a,b)
F	F	F
F	T	T
T	F	T
T	T	F

In Python, Logical XOR is done using `!=`:

```
just_one = (a != b)
```

(Take 10 seconds to confirm that this is the same as the logic table above.)

CHAPTER 4

Introduction to Graphs

Some data is easier to work with if we imagine it as a set of *nodes* connected by *edges*. For example, on some social networks, each user can follow any number of other users. We can think of each user as a node, and the edge points from the user who follows to the user they follow:

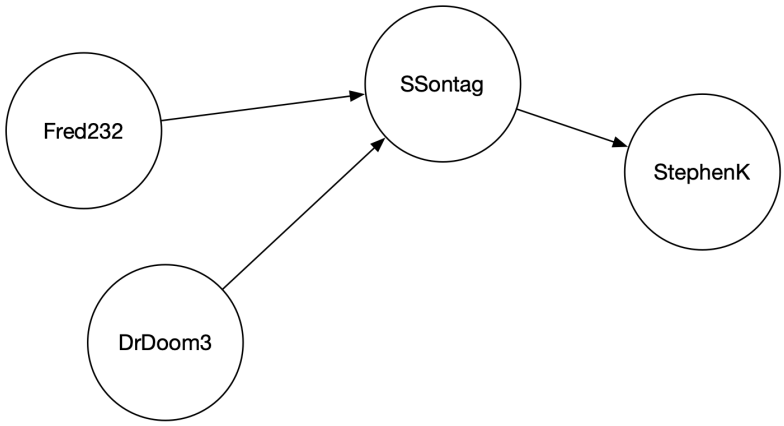


Figure 4.1: A simple directed graph.

This diagram shows four users and three follows. Following is a directed relationship: Fred232 follows SSontag, but SSontag doesn't follow Fred232. So we would way that this is a *directed graph* with four nodes and three edges.

There are also undirected graphs. For example, you can imagine a graph that represents big data lines between cities. All the big data lines allow communications in both directions, like two-way roads:

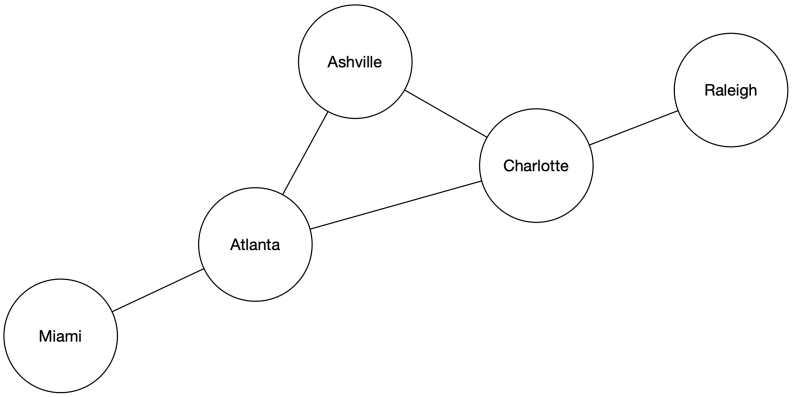


Figure 4.2: A simple undirected graph.

The arrows are gone; if data can flow from Charlotte to Raleigh, then data can flow from Raleigh to Charlotte. Most graphs are shown as undirected.

There is a whole branch of mathematics called *Graph Theory* that studies the properties of graphs. Here are two questions that we might ask about this graph:

- What is the shortest number of edges that we would need to follow to get from Miami to Raleigh?
- Does the graph have any paths where you could end up where you started? This is called a *cycle*. This graph has one cycle: Atlanta → Asheville → Charlotte → Atlanta.

There are even database systems that are specifically designed to hold and analyze graph data. Not surprisingly, these are called *Graph Databases*.

Some graphs are *connected*: You can get from one node to any other node by following edges. Is this graph connected?

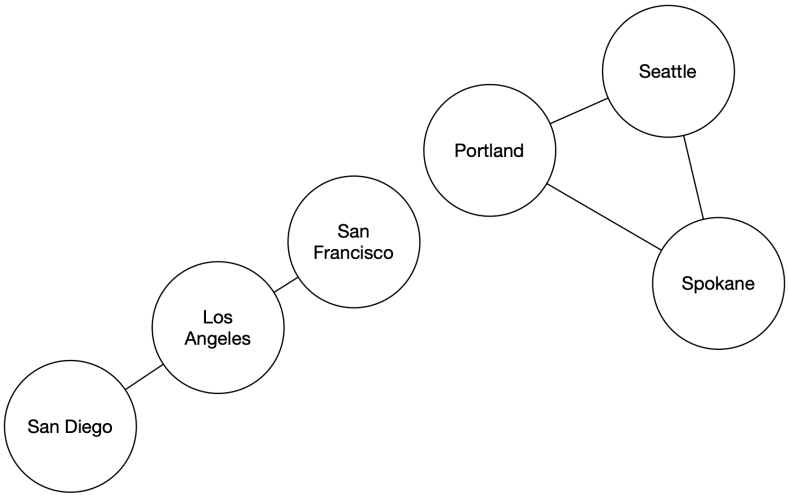


Figure 4.3: A graph with some not connected nodes.

This graph is *not* connected! You can't follow edges from San Diego to Seattle.

In graph data, the nodes and edges often have attributes. For example, a node representing a city might have a name and a population. An edge representing a data line might have a bandwidth (bits per second) and a latency (how many nanoseconds between when you put a bit into the pipe and when it comes out the other end).

4.1 Finding Good Paths

For many problems, we are trying to find the best path from one node to another. If all the edges are the same, this usually means finding the path that requires walking the fewest edges.

Sometimes the edges have a cost attribute. For example, you might want to find the cheapest way to ship a container from New York City to Long Beach, California. In this case the nodes are train depots. Each train line between the depots has a cost. What is the cheapest path?

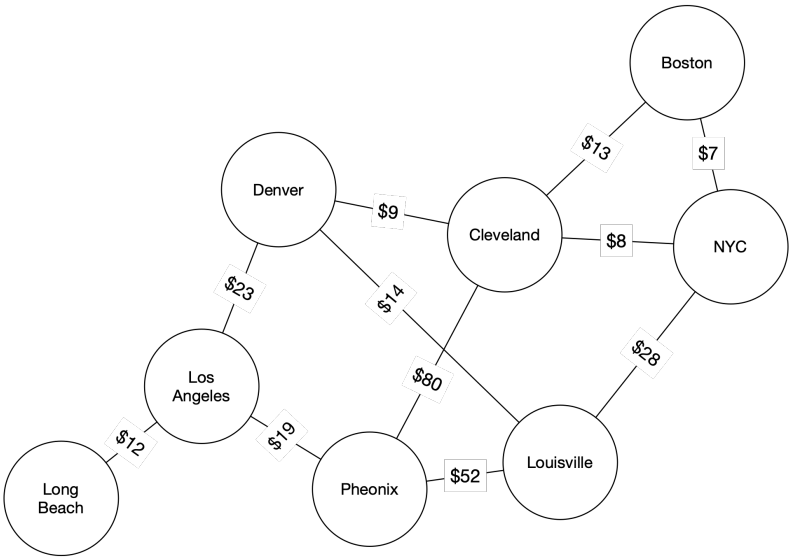


Figure 4.4: A weighted graph shown through train depots.

When edges have costs like this, we call the *weighted edges*, which forms a *weighted graph*.

The graphs that you see here are really small, so finding efficient paths isn't difficult — you could just try all of them! However, in many computer programs, we are working with millions of nodes and edges. Efficient graph algorithms are *really* important.

4.2 Graphs in Python

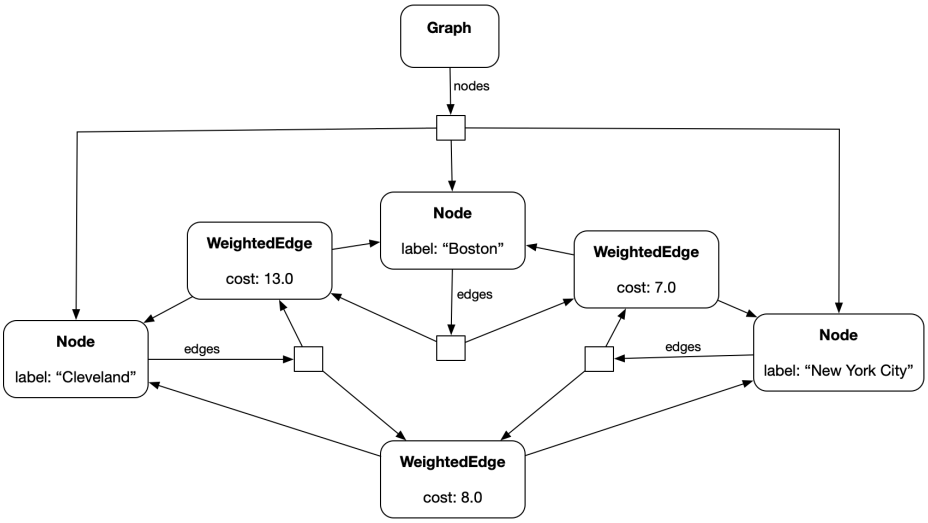
In this section you are going to write Python classes that will let you represent an undirected graph with weighted edges, like the shipping problem above.

(Naturally, things would look a little different if the graph were directed or the edges were unweighted, but this is a good starting place.)

Create a file called `graph.py`. This will hold the code for your `Node` and `WeightedEdge` classes. We will also create a `Graph` class that will just hold onto the list of its nodes.

- A `Node` will have a label string and a list of edges that touch it.
- A `Edge` will have a cost and two nodes: `node_a` and `node_b`.
- A `Graph` will have a list of nodes.

Here is what the object diagram would look like if you had only three cities:



Put this code into `graph.py`

```
class Node:
  def __init__(self, label):
    self.label = label
    self.edges = []

  def __repr__(self):
    return f"(node:{self.label}, edges:{len(self.edges)})"

class WeightedEdge:
  def __init__(self, cost, node_a, node_b):
    self.cost = cost
    self.node_a = node_a
    node_a.edges.append(self)
    self.node_b = node_b
    node_b.edges.append(self)

  def other_end(self, node_from):
    if self.node_a == node_from:
      return self.node_b
    else:
      return self.node_a

class Graph:
  def __init__(self):
    self.nodes = []

  def add_node(self, new_node):
    self.nodes.append(new_node)

  def __repr__(self):
    return f"(Graph:{self.nodes})"
```

Now, let's create some instances of `Node` and `WeightedEdge` and wire them together. Create another file in the same directory called `cities.py`. Put in this code:

```
import graph

# Create an empty graph
network = graph.Graph()

# Create city nodes and add to graph
long_beach = graph.Node("Long Beach")
network.add_node(long_beach)
los_angeles = graph.Node("Los Angeles")
network.add_node(los_angeles)
denver = graph.Node("Denver")
network.add_node(denver)
pheonix = graph.Node("Pheonix")
network.add_node(pheonix)
louisville = graph.Node("Louisville")
network.add_node(louisville)
cleveland = graph.Node("Cleveland")
network.add_node(cleveland)
boston = graph.Node("Boston")
network.add_node(boston)
nyc = graph.Node("New York City")
network.add_node(nyc)

# Create edges
graph.WeightedEdge(12, long_beach, los_angeles)
graph.WeightedEdge(23.0, los_angeles, denver)
graph.WeightedEdge(19, los_angeles, pheonix)
graph.WeightedEdge(52, pheonix, louisville)
graph.WeightedEdge(14, denver, louisville)
graph.WeightedEdge(80, pheonix, cleveland)
graph.WeightedEdge(9, denver, cleveland)
graph.WeightedEdge(8, cleveland, nyc)
graph.WeightedEdge(28, louisville, nyc)
graph.WeightedEdge(7, nyc, boston)
graph.WeightedEdge(13, cleveland, boston)

print(network)
```

Run it:

```
python3 cities.py
```

You should see some rather unexciting output:

```
(Graph:[(node:Long Beach, edges:1), (node:Los Angeles, edges:3), (node:Denver, edges:3),
(node:Pheonix, edges:3), (node:Louisville, edges:3), (node:Cleveland, edges:4),
```

```
(node:Boston, edges:2), (node:New York City, edges:3)])
```

But do not worry; we will make it more exciting in the next chapter!

CHAPTER 5

Dijkstra's Algorithm

Edsger W. Dijkstra was a great Dutch computer scientist. He came up with an algorithm for finding the cheapest path through a graph with weighted edges. Today it is known as *Dijkstra's Algorithm*. It is used in a wide variety of common problems, and is also both simple and elegant.

5.1 Algorithm Description

You are going to mark each node with how much it would cost to get there from some origin node. For example, if you are shipping a container from Long Beach, you will mark each city with the cost of getting the container to that city.

You start by marking the price for Long Beach to zero (the container is already there). You then mark each adjacent city with the cost on the edge. Next, you declare Long Beach to be "visited".

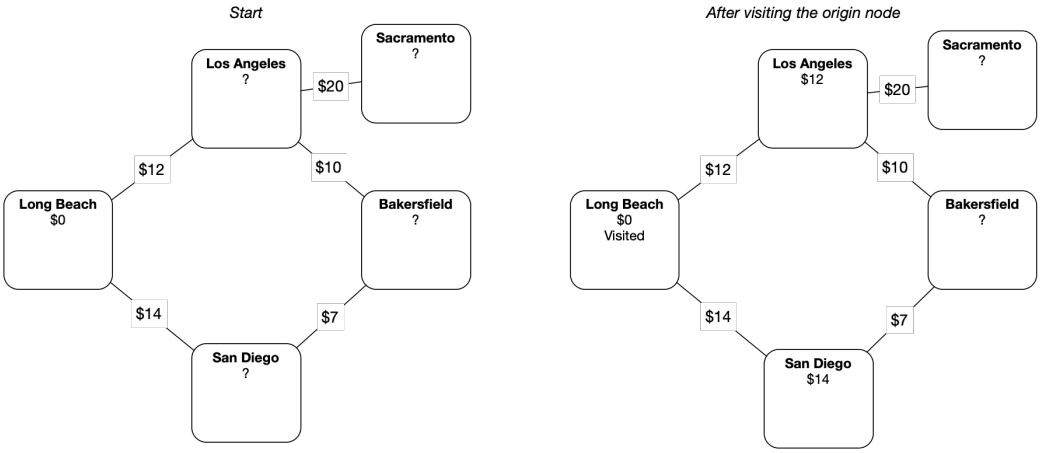


Figure 5.1: A map of weighted cities.

After that, you find the cheapest of the unvisited nodes. In this case, Los Angeles is cheaper than San Diego, so that is the node you will visit next.

You mark all of the unvisited nodes adjacent to Los Angeles, with the price to ship it to Los Angeles plus the cost of shipping the container from Los Angeles to that city. Note that Bakersfield is marked with $22.

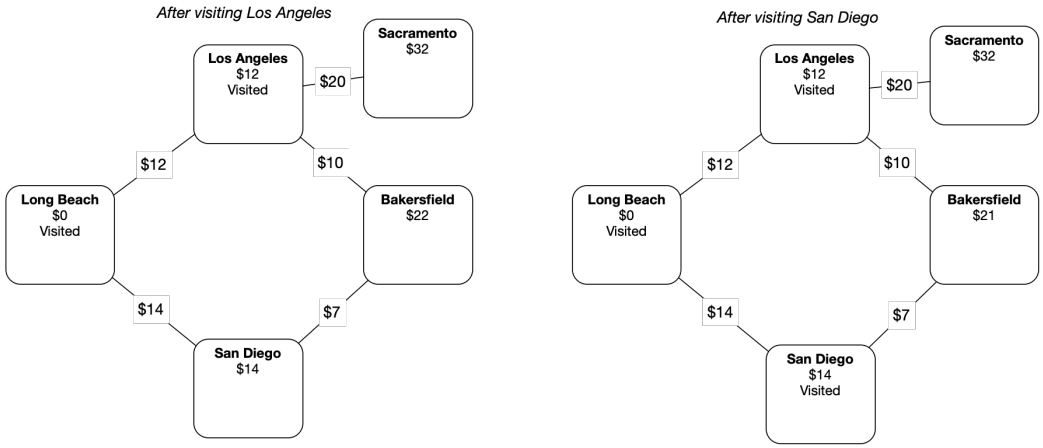


Figure 5.2: Marking each node with the cost to get to that city from the origin.

Now, the cheapest unvisited node is San Diego, so you mark its neighbors with the cost to ship to San Diego, plus the price to ship from San Diego to the neighbor.

Notice that Bakersfield is already labeled with $22 from a route through Los Angeles. But the price would be $21 if you shipped it to Bakersfield via San Diego. Because the new route is cheaper, you change the price to the lower value.

(What does it mean that a node is "visited"? If a node is marked visited, it is marked with a price that won't get any smaller.)

You continue visiting the cheapest unvisited node until all the nodes have been visited. Then you know every node has been marked with its lowest price.

In a big graph, each node may be marked several times in this process — each time with a lower price from a cheaper router.

Of course, once you have the price, you will ask, "What is the route that gets me that price?" So, we will also mark each node with the neighbor from which it would receive the shipment — the previous node. This is easy to do as we execute the algorithm.

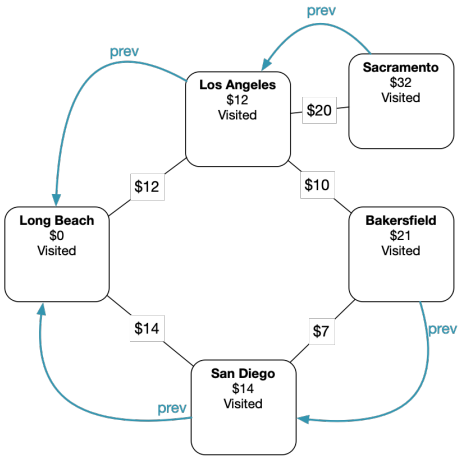


Figure 5.3: Marking each previous node.

Now, to figure out the cheapest route from San Diego to Bakersfield, we start at the destination and follow the `prev` pointer back through San Diego, then to Long Beach.

5.2 Implementation

We do not actually want to sully our graph objects with the three additional pieces of information we need:

- The current minimal cost from the origin node. This is usually called the `dist`, from "distance".
- The neighbor who gives us the current minimal cost. This is usually called `prev`, from "previous".
- Whether the city is visited or now.

So, we will keep them in collections external to the graph.

For example, to keep track of the `dist`, we will have a `dist` dictionary. Each node will be a key, the current minimal cost will be the value. If the node hasn't received even a first cost, we will put in infinity as the cost.

(After the algorithm is run, if the cost of a node is still infinity, that means that it cannot be reached from the origin node.)

We will also have a `prev` dictionary. The final node will be the key, and its previous neighbor will be the value.

Finally, the graph has a list of all the nodes, so we can just keep a set of the unvisited nodes.

Add a method to the `Graph` class that implements Dijkstra's algorithm:

```
    def cost_from_node(self, origin_node):
        # Cost of cheapest path from origin node discovered so far
        # Initially the origin is zero and all the other are infinity
        dist = {k: math.inf for k in self.nodes}
        dist[origin_node] = 0.0

        # The previous city on that cheapest path
        prev = {}

        # All the nodes start as unvisited
        unvisited = set(self.nodes)

        # While there are still unvisited nodes
        while unvisited:

            # Find unvisited node with lowest cost
            min_cost = math.inf
            for u in unvisited:
                if dist[u] < min_cost:
                    current_node = u
                    min_cost = dist[u]

            # If none are less than inf, we are done
            # This happens in graphs that are not connected
            if min_cost == math.inf:
                return (dist, prev)

            # Remove the lowest cost node from the unvisited list
            unvisited.remove(current_node)

            # Update all the unvisited neighbors
            for edge in current_node.edges:

                # What node is at the other end of this edge?
                v = edge.other_end(current_node)

                # Visited nodes are already minimized, skip them
                if v not in unvisited:
                    continue

                # Is this a shorter route?
                alt = dist[current_node] + edge.cost
                if alt < dist[v]:

                    # Update the distance and prev dicts
                    dist[v] = alt
```

```
                    prev[v] = current_node

        return (dist, prev)
```

Append some code to your `cities.py` that tests this method:

```
(cost_from_long_beach, prev) = network.cost_from_node(long_beach)
print(f"\nMinimum costs from Long Beach = {cost_from_long_beach}")
print(f"\nLast city before = {prev}")

nyc_cost = cost_from_long_beach[nyc]

if nyc_cost < math.inf:
    print(f"\n*** Total cost from Long Beach to NYC: ${nyc_cost:.2f} ***")
else:
    print("You can't get to NYC from Long Beach")
```

When you run it, you should get a list of how much it costs to ship a container to each city from Long Beach:

```
Minimum costs from Long Beach = {(node:Long Beach, edges:1): 0.0,
(node:Los Angeles, edges:3): 12.0, (node:Denver, edges:3): 35.0,
(node:Pheonix, edges:3): 31.0, (node:Louisville, edges:3): 49.0,
(node:Cleveland, edges:4): 44.0, (node:Boston, edges:2): 57.0,
(node:New York City, edges:3): 52.0}
```

You will also get a collection of node pairs. What are these? For each node, you get the node that you would pass through on the cheapest route from Long Beach:

```
Last city before = {(node:Los Angeles, edges:3):(node:Long Beach, edges:1),
(node:Denver, edges:3):(node:Los Angeles, edges:3),
(node:Pheonix, edges:3):(node:Los Angeles, edges:3),
(node:Louisville, edges:3):(node:Denver, edges:3),
(node:Cleveland, edges:4):(node:Denver, edges:3),
(node:New York City, edges:3):(node:Cleveland, edges:4),
(node:Boston, edges:2): (node:Cleveland, edges:4)}
```

Your users will not want to read this; give them the shortest path as a list. Add a function to `graph.py` that turns the `prev` table into a path of nodes that lead from the origin to the destination:

```
def shortest_path(prev, destination):

    # Include the destination in the path
    path = [destination]
    current_node = destination
```

```
    # Keep stepping backward in the path
    while current_node in prev:

        # What node should come before the current node?
        previous_node = prev[current_node]

        # Insert it at the start of the list
        path.insert(0, previous_node)
        current_node = previous_node

    return path
```

Test that out:

```
if nyc_cost < math.inf:
    print(f"*** Total cost from Long Beach to NYC: $nyc_cost:.2f ***")

    path_to_nyc = graph.shortest_path(prev, nyc)
    print(f"*** Cheapest path from Long Beach to NYC: path_to_nyc ***")
else:
    print("You can't get to NYC from Long Beach")
```

This should look like this:

```
*** Cheapest path from Long Beach to NYC: [(node:Long Beach, edges:1),
(node:Los Angeles, edges:3), (node:Denver, edges:3), (node:Cleveland, edges:4),
(node:New York City, edges:3)] ***
```

5.3 Making it faster

On really big networks, doing a full Dijkstra's algorithm would take too long; luckily, there are many methods for getting similar results quickly. When you ask for directions from Google Maps, it does not do a full Dijkstra's Algorithm for every possible route — it would just take too long.

However, there is a way to speed up this implementation. Look at this snippet:

```
            # Find unvisited node with lowest cost
            min_cost = math.inf
            for u in unvisited:
                if dist[u] < min_cost:
                    current_node = u
                    min_cost = dist[u]
```

We are scanning through the list of all unvisited nodes, one by one, looking for the one with the lowest cost. If we kept this list sorted by cost, then the next one to visit would always be the first one in the list. This is done with a *priority queue* – a list that keeps itself sorted by some priority number – in this case the cost. In python, the standard priority queue is `heapq`.

(So why didn't I implement this using `heapq`? For Dijkstra's Algorithm, the nodes' priority – the current cost – changes as we find cheaper routes. `heapq` doesn't handle the changing priority very gracefully.)

In the next chapter, you will make a priority queue class that will work in this case.

Chapter 6

Binary Search

As mentioned in the last chapter, you are going to make a priority queue for use with Dijkstra's Algorithm. Using it will look like this:

```
import kpqueue

myqueue = pqueue.PriorityQueue()
myqueue.add(long_beach, 0) # Inserts first city and its cost
myqueue.add(san_diego, 14) # Puts San Diego after Long Beach
myqueue.add(los_angeles,12) # Inserts LA between Long Beach and San Diego
current_city = myqueue.pop() # Returns first city (Long Beach) and removes it
```

Now, if an item gets a new priority, we need to remove it and reinsert it in the new spot.

```
myqueue.add(city_a, 16) # Puts it last in the queue
myqueue.update(city_a, 16, 13) # Moves it to between LA and San Diego
```

6.1 A Naive Implementation of the Priority Queue

Create a file called `kpqueue.py`. Let's do a simple implementation that stores the priority and the data as tuple. And we will keep it sorted by the priority. If two tuples have the same priority, we will sort by the data.

Type this in to `kpqueue.py`:

```
class PriorityQueue:
    def __init__(self):
        self.list = []

    # Return and remove the first item
    def pop(self):
        if len(self.list) > 0:
            return self.list.pop(0)
        else:
            return None

    def __len__(self):
        return len(self.list)

    def update(self, value, old_priority, new_priority):
```

```
        old_pair = (old_priority, value)
        self.list.remove(old_pair)
        self.add(value, new_priority)

    def add(self, value, priority):
        pair = (priority, value)
        # Add it at the end
        self.list.append(pair)
        # Resort the list
        self.list.sort()
```

This will work fine, but it could be much more efficient:

- Every time we add a single element, we resort the whole list.
- The function `remove` is searching the list sequentially for the item to delete.

In a minute, we will revisit these inefficiencies and make them better.

6.2 Using the Priority Queue

We are going to change `graph.py` to use the priority queue. While we are doing so, why don't we also shrink the memory footprint of our program a bit.

Notice that as the algorithm is running, each node is in one of three states:

- Unseen: In the earlier implementation, these were the nodes with `math.inf` as their cost.
- Seen, but not finalized: These are "unvisited", but don't have `math.inf` as their cost.
- Finalized: These are the "visited" nodes — we know that their cost won't decrease any more.

We can shrink the memory foot print by not putting the unseen into the dist dictionary at all. And instead of a separate set for "unvisited", what if we moved finalized nodes and their distances into a separate dictionary?

Rewrite the `cost_from_node` function in `graph.py`:

```
            # Visited nodes are already minimized, skip them
              if v in finalized_dist:
                  continue

              # What is the cost to this neighbor?
              alt = current_node_cost + edge.cost
```

```
                # Is this the first time I am seeing the node?
                if v not in seen_dist:

                    # Insert into the seen_dict, prev, and priority queue
                    seen_dist[v] = alt
                    prev[v] = current_node
                    pqueue.add(v, alt)

                else: # v has been seen. Is this a cheaper route?
                    old_dist = seen_dist[v]
                    if alt < old_dist:
                        # Update the seen_dict, prev, and priority queue
                        seen_dist[v] = alt
                        prev[v] = current_node
                        pqueue.update(v, old_dist, alt)

        return (finalized_dist, prev)
```

This should be have exactly the same result, except for the unreachable nodes. If you have a graph that is not connected, there will be nodes that cannot be reached from the origin. In the old version, these had a cost of `math.inf`. Now, they will just not be in the dictionary at all. So, change `cities.py` to deal with this:

```
if nyc in cost_from_long_beach:
    nyc_cost = cost_from_long_beach[nyc]
    print(f"\n*** Total cost from Long Beach to NYC: ${nyc_cost:.2f} ***")

    path_to_nyc = graph.shortest_path(prev, nyc)
    print(f"\n*** Cheapest path from Long Beach to NYC: {path_to_nyc} ***")
else:
    print("You can't get to NYC from Long Beach")
```

If you run `cities.py` now, it should behave exactly like the old version.

However, there is a bug. It will rear its head if two cities with the same cost are in the priority queue together. Change `cities.py` so that Denver and Pheonix have the same cost:

```
graph.WeightedEdge(12, long_beach, los_angeles)
graph.WeightedEdge(19, los_angeles, denver)
graph.WeightedEdge(19, los_angeles, pheonix)
```

Now try running it. You should get an error:

```
TypeError: '<' not supported between instances of 'Node' and 'Node'
```

What happened? The `loc_for_pair` method is comparing tuples made up of a `float` and a `Node`. The `float` comes first in the tuple, so that is compared first. However, if the two tuples have the same priority, it then compares nodes.

The error statement says, "Nodes don't have a less-than method; I don't know how to compare them."

Each `Node` lives at an address in memory. You can get that address as a number using the `id` function. The ID is unique and constant over the life of the object. It is a rather arbitary ordering, but it will work for this problem. Add a method to your `Node` class:

```
# Nodes will be ordered by their location in memory
def __lt__(self, other):
    return id(self) < id(other)
```

Fixed.

Now, let's make the priority queue more efficient.

6.3 Binary Search

The phone company in every town used to print a thing called a phone book. The names and phone numbers were arranged alphabetically. As you might imagine, these books often had more than a thousand pages.

If you were looking for "John Jeffers", you would not start at the first page and read sequentially until you reached his name. You would open the book in the middle, and see a name like "Mac Miller", then think "Jeffers comes before Miller". You would then split the pages in your left hand in half and see a name like "Hester Hamburg" and think "Jeffers comes after Hamburg". You would then split the pages in your right hand, and continue on like this until you found the page with "John Jeffers" on it. That is binary search.

Binary Search is a search algorithm that finds the position of a target value within a sorted array. The binary search algorithm works by repeatedly dividing the search interval in half. If the target value is equal to the middle element of the array, the position is returned. If the target value is less than or greater than the middle element, the search continues in the lower or upper half of the array, respectively.

6.4 Algorithm

The binary search algorithm can be described as follows:

1. If the array is empty, the search is unsuccessful, so return "Not Found".
2. Otherwise, compare the target value to the middle element of the array.
3. If the target value matches the middle element, return the middle index.
4. If the target value is less than the middle element, repeat the search with the lower half of the array.
5. If the target value is greater than the middle element, repeat the search with the upper half of the array.
6. Repeat steps 2-5 until the target value is found or the array is exhausted.

```
class PriorityQueue:
    def __init__(self):
        self.list = []

    # Return and remove the first item
    def pop(self):
        if len(self.list) > 0:
            return self.list.pop(0)
        else:
            return None

    def __len__(self):
        return len(self.list)

    def add(self, value, priority):
        pair = (priority, value)
        i = self.loc_for_pair(pair)
        self.list.insert(i, pair)

    def update(self, value, old_priority, new_priority):
        old_pair = (old_priority, value)
        i = self.loc_for_pair(old_pair)
        del self.list[i]
        self.add(value, new_priority)

    def loc_for_pair(self, pair):
        # The range where it could be is [lower, upper)
        # Start with the whole list
        lower = 0
        upper = len(self.list)

        while upper > lower:
            next_split = (upper + lower) // 2
            v = self.list[next_split]
            if pair < v:  # pair is to the left
                upper = next_split
            elif pair > v:  # pair is to the right
                lower = next_split + 1
```

```
            else: # Found pair!
                return next_split
        return lower
```

If you try running it now, it should work perfectly.

You now have a graph class that will find the cheapest path quickly, even if it has thousands of nodes with thousands of edges.

Chapter 7

Other Graph Algorithms

Now that you are familiar with Dijkstra's algorithm for finding the shortest path in a graph, you are well-equipped to understand more graph algorithms. This document will discuss two other important algorithms: Depth-First Search (DFS) and the Bellman-Ford algorithm.

7.1 Depth-First Search

Depth-First Search (DFS) is an algorithm for traversing or searching tree or graph data structures. DFS uses a stack (or sometimes recursion, which uses the system stack implicitly) to explore the graph in a depthward motion until it hits a node with no unvisited adjacent nodes, then it backtracks.

The procedure is as follows:

1. Push the root node into the stack.
2. Pop a node from the stack, and mark it as visited.
3. Push all unvisited adjacent nodes into the stack.
4. Repeat steps 2 and 3 until the stack is empty.

DFS is particularly useful for solving problems like connected-component detection in graphs and maze-solving.

7.2 Bellman-Ford Algorithm

The Bellman-Ford Algorithm is another shortest path algorithm like Dijkstra's. However, unlike Dijkstra's Algorithm, Bellman-Ford can handle graphs with negative weight edges.

The algorithm works as follows:

1. Assign a tentative distance value for every node: set it to zero for our initial node and to infinity for all other nodes.
2. For each edge (u, v) with weight w, if the current distance to v is greater than the

distance to u plus w, update the distance to v to be the distance to u plus w.

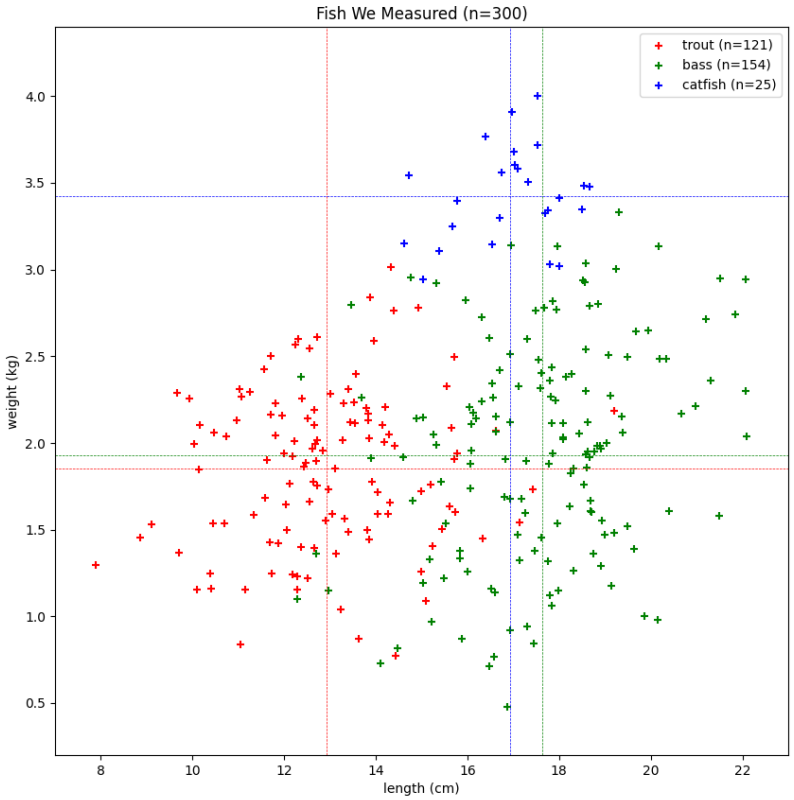
3. Repeat the previous step $|V| - 1$ times, where $|V|$ is the number of vertices in the graph.
4. After the above steps, if you can still find a shorter path, there exists a negative cycle.

If the graph does not contain a negative cycle reachable from the source, the shortest paths are well-defined, and Bellman-Ford will correctly calculate them. If a negative cycle is reachable, no solution exists, but Bellman-Ford will detect it.

Chapter 8

Bayesian Classifiers

You drop a net in several places in a lake. You measure, weigh, and identify every fish to be one of the following: `trout`, `bass`, or `catfish`. Here is a scatter plot of what you find:



(Data: `make_data.py`, Graph: `1_scatter_fish.py`)

The vertical and horizontal lines represent the mean length and width (respectively) of each type of fish.

You are writing a system that someone on the lake could use to identify their fish.

8.1 The Prior

Even before the person measures and weighs the fish, they can make a decent guess at what any randomly selected fish is:

Fish	Count	Percent
trout	121/300	40.3%
bass	154/300	51.3%
catfish	25/300	8.3%

If you know nothing about the fish, there is a 51.3% chance it is a bass. The probability without any observations is known as the *prior*:

$$\mathrm{P}(\text{fish} = \text{trout}) = .403$$
$$\mathrm{P}(\text{fish} = \text{bass}) = .513$$
$$\mathrm{P}(\text{fish} = \text{trout}) = .083$$

However, now the user makes an observation: The fish is 16 inches long and weighs 2.1 kg. Can we update our beliefs based on this? Sounds like a job for conditional probability.

By the definition of conditional probability:

$$\mathrm{P}(\text{fish} = \text{trout}|\text{length} = 16, \text{weight} = 2.1) = \frac{\mathrm{p}(\text{fish} = \text{trout}, \text{length} = 16, \text{weight} = 2.1)}{\mathrm{p}(\text{length} = 16, \text{weight} = 2.1)}$$

At this rate, we are going to run out of space, so we will use F to represent fish, L to represent length, and W to represent weight.

We can calculate the denominator by summing up all the possibilities. Now, the right-hand side is:

$$\frac{\mathrm{p}(\mathrm{F}{=}\text{trout}, \mathrm{L}{=}16, \mathrm{W}{=}2.1)}{\mathrm{p}(\mathrm{F}{=}\text{trout}, \mathrm{L}{=}16, \mathrm{W}{=}2.1) + \mathrm{p}(\mathrm{F}{=}\text{bass}, \mathrm{L}{=}16, \mathrm{W}{=}2.1) + \mathrm{p}(\mathrm{F}{=}\text{catfish}, \mathrm{L}{=}16, \mathrm{W}{=}2.1)}$$

We can calculate the probability of all three possibilities (which will add up to 1.0).

8.2 Naive Bayes

The assumption in naive Bayes is that the probability distributions of length and weight are conditionally independent given the breed of the fish. That is:

$$\mathrm{p}(\mathrm{F}{=}\text{trout}, \mathrm{L}{=}16, \mathrm{W}{=}2.1) = \mathrm{p}(\mathrm{L}{=}16|\mathrm{F}{=}\text{trout})\mathrm{p}(\mathrm{W}{=}2.1|\mathrm{F}{=}\text{trout})\mathrm{P}(\mathrm{F}{=}\text{trout})$$

Within a particular species, things like length and weight, which represent of the sum of a lot of genetic and environmental factors, are often normally distributed.

What if we assume that length has a normal distribution for each kind of fish? What is the most likely estimator for the mean of each trait?

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

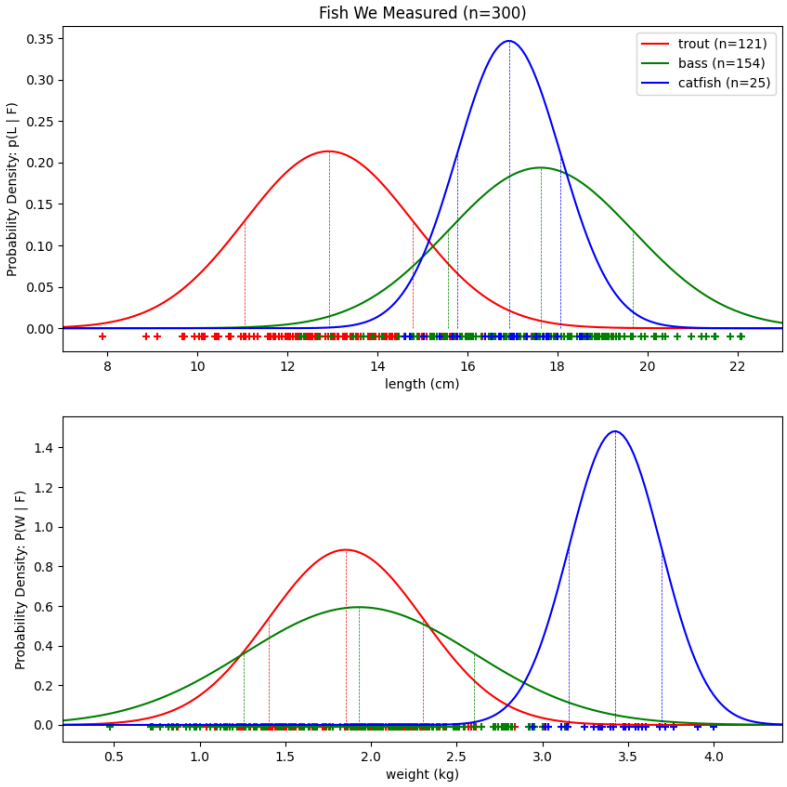
And variance?

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (\mu - x_i)^2$$

We can compute those pretty easily:

Fish	Trait	μ	σ^2
Trout	Length	12.91522571	3.49054363
	Weight	1.85290281	0.20387574
Bass	Length	17.62628327	4.23881747
	Weight	1.92960088	0.45092986
Catfish	Length	16.92187387	1.32301935
	Weight	3.42423185	0.07244546

If we plot those probability distributions out:



(Source:`2_univariates.py`)

Note that the area under each one of those curves is exactly 1.0. They are probability distributions.

Now we can use the formula for the normal distribution:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Thus,

Given	$p(L = 16\|F)$	$p(W = 2.1\|F)$
F=Trout	.0546	.7607
F=Bass	.1418	.5753
F=Catfish	.2516	0.01

Using the naive assumption, we can now compute the joint probability for any type of fish:

$$p(F = ?, L = 16, W = 2.1) = p(W = 2.1|F = ?)p(L = 16|F = ?)p(F = ?)$$

For	$p(L = 16, W = 2.1, F = ?)$
F=Trout	.016763
F=Bass	.041886
F=Catfish	0.00001

The sum of these is $p(L = 16, W = 2.1)$: .0586.

By the definition of conditional probability:

$$p(F = ?|L = 16, W = 2.1) = \frac{p(F = ?, L = 16, W = 2.1)}{p(L = 16, W = 2.1)}$$

So,

For	$p(F = ?|L = 16, W = 2.1)$
F=Trout	.286
F=Bass	.714
F=Catfish	≈ 0

If you pull a random fish out of the lake and its length is 16 and its weight is 2.1, there is a 71.4% chance that it is bass. There is a 28.6% chance it is a trout.

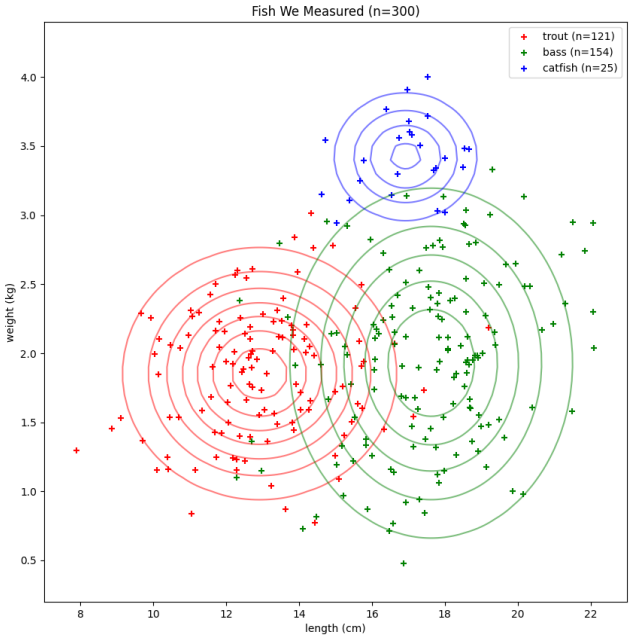
8.3 Using the multivariate Gaussian distribution

Given the kind of fish, length and width are independent!? That makes no sense. Longer fish tend to be heavier, right?

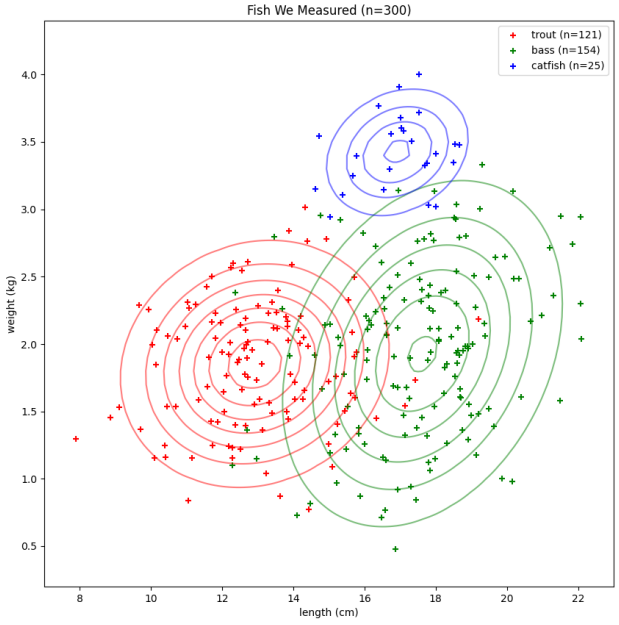
The multivariate Gaussian distribution is a generalization of the normal distribution::

- It deals with data of any dimension d.
- It can include the idea of covariance. For example, length and width are not independent.

If we use your naive Bayes, we can get two-dimensional density plots that would look like this:



With independence, there are ovals that are circles, ovals that go up, or ovals that sideways. If we use the multivariate Gaussian distribution, we get:



When there is covariance, the ovals can slant in any direction.

If we discard the naive assumption, we should get slightly more accurate results.

The multivariate Gaussian distribution uses vectors and matrices. The mean $\vec{\mu}$ is a vector

of dimension d. The variance of each variable and its covariance with the other variables is captured in a $d \times d$ matrix called *the covariance matrix* is usually called $\mathbf{\Sigma}$. (Yes, this is also the symbol for summation. This creates some confusion at times, but you can usually figure out which is intended by the context.)

When you know $\vec{\mu}$ and $\mathbf{\Sigma}$, the probability density for any vector $\vec{x}$ is:

$$p(\vec{x}) = (2\pi)^{-d/2} \det(\mathbf{\Sigma})^{-1/2} \exp\left(-\frac{1}{2}(\vec{x} - \vec{\mu})^T \mathbf{\Sigma}^{-1} (\vec{x} - \vec{\mu})\right)$$

If you have several data points $\vec{x_1}, ..., \vec{x_n}$, how do you compute the values of $\vec{\mu}$ and $\mathbf{\Sigma}$ for which the observations $\vec{x_1}, ..., \vec{x_n}$ would be most likely?

$$\vec{\mu} = \frac{1}{n} \sum_{i=1}^{d} \vec{x_i}$$

And the covariance matrix? (Here, we think of each $\vec{x_i}$ and $\vec{\mu}$ as column vectors.)

$$\mathbf{\Sigma} = \frac{1}{n} \sum_{i=1}^{n} (\vec{x_i} - \vec{\mu})(\vec{x_i} - \vec{\mu})^T$$

Notice that the diagonal entries of the matrix are the variance of each variable. Notice also that this is a symmetric matrix.

Applying this to our data we get:

Fish	$\vec{\mu}$	$\mathbf{\Sigma}$
Trout	$\begin{bmatrix} 12.91522571 \\ 1.85290281 \end{bmatrix}$	$\begin{bmatrix} 3.51963149 & 0.09975744 \\ 0.09975744, & 0.20557471 \end{bmatrix}$
Bass	$\begin{bmatrix} 17.62628327 \\ 1.92960088 \end{bmatrix}$	$\begin{bmatrix} 4.26652216 & 0.39553268 \\ 0.39553268 & 0.45387711 \end{bmatrix}$
Catfish	$\begin{bmatrix} 16.92187387 \\ 3.42423185 \end{bmatrix}$	$\begin{bmatrix} 1.37814515 & 0.07281453 \\ 0.07281453 & 0.07546403 \end{bmatrix}$

We can use these to compute the likelihood of seeing a fish with length = 16 and weight = 2.1 for each type of fish:

Given	$p(L = 16, W = 2.1 \mid F = ?)$
F=Trout	.04578
F=Bass	.07732
F=Catfish	.000001

Multiplied by the prior to get the joint probability:

Given	$p(L = 16, W = 2.1, F = ?)$
F=Trout	.01847
F=Bass	.03969
F=Catfish	≈ 0

We sum those to get $p(L = 16, W = 2.1) = .0582$.

Given	$p(F = ? \mid L = 16, W = 2.1)$
F=Trout	.318
F=Bass	.682
F=Catfish	≈ 0

Given a randomly selected fish that is 19 inches long and 2.1 kg in weight, you would guess that it is a bass with a confidence of 68.2%.

Appendix A

Answers to Exercises

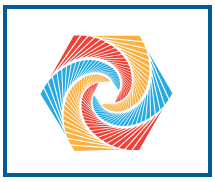
Answer to Exercise 1 (on page 23)

A	B	not (A or B)	(not A) and (not B)
F	F	T	T
F	T	F	F
T	F	F	F
T	T	F	F

Notice that the two expressions are equivalent!

DeMorgan's Rule says "not (A or B)" is equivalent to "(not A) and (not B)".

It also says "not (A and B)" is equivalent to "(not A) or (not B)".



Index