Linear Algebra

Foundations of Machine Learning

Paul A. Jensen
University of Michigan

Fall 2022 Edition

Contents

Introduction

This book has three parts. In Part I, we analyze linear systems that transform a multidimensional vector (**x**) into another vector (**y**). This transformation is often expressed as a linear system via matrix multiplication: $\mathbf{y} = \mathbf{A}\mathbf{x}$. Some linear systems can be solved exactly and uniquely, but we will also consider solution strategies with alternative objectives when the system contains either too much or not enough information. In Part II we shift to nonlinear systems that must be solved approximately via iterative methods. The nonlinear methods include many of the foundational techniques in machine learning. Part III focuses on matrix theory, expanding our understanding of high-dimensional data.. We will learn how to analyze and extract information from matrices without a clear input/output relationship.

Notation

We will distinguish scalars, vectors, and matrices with the following typographic conventions:

Object	Font and Symbol	Examples
Scalars	italicized, lowercase letters	x, α, y
Vectors	bold lowercase letters	$\mathbf{x}$, $\mathbf{y}$, $\mathbf{n}$, $\mathbf{w}$
Matrices	bold uppercase	$\mathbf{A}$, $\mathbf{A}^{-1}$, $\mathbf{B}$, $\mathbf{\Gamma}$

There are many ways to represent rows or columns of a matrix **A**. Since this book uses Matlab, I think it is convenient to use a matrix addressing scheme that reflects Matlab's syntax. So, the ith row of matrix **A** will be $\mathbf{A}(i,:)$, and the jth column will be $\mathbf{A}(:,j)$. Rows or columns of a matrix are themselves vectors, so we choose to keep the boldface font for the matrix **A** even when it is subscripted. We could also use Matlab syntax for vectors ($\mathbf{x}(i)$, for example). However, the form x_i is standard across many fields of mathematics and engineering, so we retain the common notation. The lack of boldface font reminds us that elements of vectors are scalars.

One goal of this book is to increase the precision of your mathematical writing. We will regularly use the symbols in Table 1 to describe mathematical concepts and relations. These symbols succinctly express mathematical ideas. For example, we can define the set of rational numbers as

A number is rational if and only if it can be expressed as the quotient of two integers.

or with the statement

$$r \in \mathbb{Q} \Leftrightarrow \exists\ p, q \in \mathbb{Z} \text{ s.t. } r = p/q.$$

While the latter statement is shorter, it is more difficult to understand. So whenever possible I recommend writing statements with as few symbols as necessary. Rely on mathematical symbols only when a textual definition would be unwieldy or imprecise, or when brevity is important (like when writing on a chalkboard).

Acknowledgements

This book was originally developed for the BIOE 210 course at the University of Illinois at Urbana-Champaign. The course began in 2017 when Michael Insana had the foresight to include linear algebra as part of every bioengineer's training. Gregory Underhill and I co-taught the course in 2018 and helped transform it into the present structure. Several teaching assistants and graders developed problems and solutions, including Dikshant Pradhan, Hyeon Ryoo, Cynthia Liu, and Boeun Hwang. The BIOE 210 students provided enormous feedback to improve the content and delivery.

Table 1: Mathematical notation used in this book.

Symbol	Read As	Description
$\Rightarrow$	implies	$p \Rightarrow q$ means that whenever p is true, q must also be true.
$\Leftrightarrow$	if and only if	A symmetric, stronger version of $\Rightarrow$. The expression $p \Leftrightarrow q$ means $p \Rightarrow q$ and $q \Rightarrow p$.
$\forall$	for all	Remember this symbol as an upside down "A", as in "for **A**ll".
$\exists$	there exists	Remember this symbol as a backwards "E", as in "there **E**xists". To say something does not exist, use $\nexists$.
$\in$ ($\notin$)	is (not) a member of	Used to state that a single element is a member of a set, i.e. $1 \in \mathbb{Z}$. To say that a set is a subset of another set, use $\subset$.
s.t.	such that	Other texts use the symbol | (a vertical pipe) instead of "s.t.". Note that "s.t." is set in normal, not italicized font.
$\mathbb{R}$	the real numbers	The numbers along a line. The reals include both rational and irrational numbers.
$\mathbb{R}^n$	the set of n-dimensional vectors of real numbers	Each value of n is a different set. If $r \in \mathbb{R}^2$ then $r \notin \mathbb{R}^3$. Also, $\mathbb{R}^2$ is not a subset of $\mathbb{R}^3$, etc.
$\mathbb{Z}$	the integers	The integers contain the natural numbers (1, 2, 3, ...), their negatives (-1, -2, -3, ...), and the number zero (0). The symbol comes from the German word for "number" (Zahlen). The word "integer" (Latin for "whole") is used since integers have no fractional part.
$\mathbb{Q}$	the rationals	The rational numbers are all numbers that are the quotient of two integers. The symbol derives from the word "quotient".
$\mapsto$	maps to	Describes the inputs and outputs of an operation. An operation that maps a vector of reals to a real number is $\mathbb{R}^n \mapsto \mathbb{R}$. An operation that maps two integers to a rational is $\mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Q}$.
$\equiv$	is defined as	Two expressions are equivalent because we have defined them as such, not because the relationship can be shown logically. For example, $a/b \equiv a \times b^{-1}$ defines the division operator using the multiplicative inverse.

Part I

Linear Systems

Chapter 1

Fields and Vectors

1.1 Algebra

Algebra is a branch of mathematics that contains symbols and a set of rules to manipulate them.

You are probably familiar with the idea of symbols. We call them variables, and in previous algebra courses you used them to represent unknown real numbers. In this course we will use variables to represent *vectors*. Vectors are collections of elements, such as the real numbers. When we say vector, we assume a *column vector*—a vertical array of elements. A *row vector* is a horizontal array of elements. We will see that column vectors are more convenient. The number of elements in a vector is its dimension. We can write an n-dimensional vector as

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}.$$

We surround the elements of a vector with either parentheses ( ) or square brackets [ ]. Straight lines | | or curly braces { } are not allowed, as these have special meanings. While some vectors have elements that are real numbers, vectors themselves are not numbers. An n-dimensional vector does not belong to the set $\mathbb{R}$ of real numbers; it belongs to a special set $\mathbb{R}^n$ of all other vectors of dimension n with real elements.

We use the rules of algebra to manipulate elements. However, only certain sets of elements are amenable to the rules of algebra. These algebra-compatible sets are called *fields*. A set of conditions, or *axioms*, must be true about a set before we can consider it a field. These field axioms define the rules of algebra.

After spending years studying algebra, you might think that there are many byzantine rules that govern fields. In fact, there are only five. The five axioms describe only two operations (addition and multiplication) and define two special elements that must be in every field (0 and 1).

1.2 The Field Axioms

Given elements a, b, and c in a field:

1. **Associativity**.
$$a+b+c=(a+b)+c=a+(b+c)$$
$$abc=(ab)c=a(bc)$$

2. **Commutativity**.
$$a+b=b+a$$
$$ab=ba$$

3. **Distribution** of multiplication over addition.
$$a(b+c)=ab+ac$$

4. **Identity**. There exist elements 0 and 1, both in the field, such that
$$a+0=a$$
$$1\times a=a$$

5. **Inverses**.
    - For all a, there exists an element $(-a)$ in the field such that $a+(-a)=0$. The element $-a$ is called the *additive inverse* of a.
    - For all $a\neq 0$, there exists an element (a^{-1}) in the field such that $a\times a^{-1}=1$. The element a^{-1} is called the *multiplicative inverse* of a.

It might surprise you that only five axioms are sufficient to recreate everything you know about algebra. For example, nowhere do we state the special property of zero that $a\times 0=0$ for any number a. We don't need to state this property, as it follows from the field axioms.

Theorem. $a\times 0=0$.

Proof.

$$\begin{aligned} a \times 0 &= a \times (1-1) \\ &= a \times 1 + a \times (-1) \\ &= a - a \\ &= 0 \end{aligned}$$

□

Similarly, we can prove corollaries from the field axioms.

A corollary is a statement that follows directly from a theorem.

Corollary. *If $ab = 0$, then either $a = 0$ or $b = 0$ (or both).*

Proof. Suppose $a \neq 0$. Then there exists a^{-1} such that

$$\begin{aligned} a^{-1}ab &= a^{-1} \times 0 \\ 1 \times b &= 0 \\ b &= 0 \end{aligned}$$

A similar argument follows when $b \neq 0$. □

The fundamental theorem of algebra relies on the above corollary when solving polynomials. If we factor a polynomial into the form $(x - r_1)(x - r_2)\cdots(x - r_k) = 0$, then we know the polynomial has roots $r_1, r_2, \ldots, r_k$. This is only true because the left hand side of the factored expression only reaches zero when at least one of the factors is zero, i.e. when $x = r_i$.

1.2.1 Common Fields in Mathematics

The advantage of fields is that once a set is proven to obey the five field axioms, we can operate on elements in the field just like we would operate on real numbers. Besides the real numbers (which the concept of fields was designed to emulate), what are some other fields?

The rational numbers are a field. The numbers 0 and 1 are rational, so they are in the field. Since we add and multiply rational numbers just as we do real numbers, these operations commute, associate, and distribute. All that remains is to show that the rationals have additive and multiplicative inverses in the field. Let us consider a rational number p/q, where p and q are integers.

- We know that $-p/q$ is also rational, since $-p$ is still an integer. The additive inverse of a rational number is in the field of rational numbers.

- The additive inverse of p/q is q/p, which is also rational. The multiplicative inverse of a rational is also in the field.

So the rational numbers are a field. What does this mean? If we are given an algebraic expression, we can solve it by performing any algebraic manipulation and still be assured that the answer will be another rational number.

The integers, by contrast, are not a field. Every integer has a reciprocal ($2 \longrightarrow 1/2$, $-100 \longrightarrow -1/100$, etc.). However, the reciprocals are themselves not integers, so they are not in the same field. The field axioms require that the inverses for every element are members of the field. When constructing a field, every part of every axiom must be satisfied.

Example 1.1. Let's demonstrate why *every* axiom must hold in a field. Imagine the simple equation $y = ax + b$, which we solve for x to yield

$$x = \frac{y - b}{a}.$$

If we wanted to solve this equation using only rational numbers, we would not need to change anything. So long as the values we input for the variables a, b, and y were rational, the value of x would also be rational. We solved the equation using field algebra, and the rationals constitute a field. Everything works out.

Now imagine you wanted only integer solutions. Even if the values for a, b, and y were integers, there is no guarantee that x would be an integer. ($a = 2$, $b = 4$, and $y = 3$ yields $x = -1/2$, for example). Because the integers are not a field, algebra does not work on them. In particular, the integers do not have integer multiplicative inverses (except for 1 and -1). When we divide by a, we assumed that the value $1/a$ exists in the field, which it does not. ■

Interestingly, the integers always have integer additive inverses, so the solution to the equation $y = x - b$ is always an integer (for integer y and b) since we could solve the equation with only additive inverses.

1.3 Vector Addition

Addition of two vectors is defined *elementwise*, or element-by-element.

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{pmatrix}$$

Since this is a direct extension of scalar addition, it is clear that vector addition commutes $[\mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x}]$ and is associative $[\mathbf{x} + \mathbf{y} + \mathbf{z} = (\mathbf{x} + \mathbf{y}) + \mathbf{z} = \mathbf{x} + (\mathbf{y} + \mathbf{z})]$.

The additive inverse of a vector $\mathbf{x}$ (written $-\mathbf{x}$) is also constructed elementwise.

$$-\mathbf{x} = \begin{pmatrix} -x_1 \\ -x_2 \\ \vdots \\ -x_n \end{pmatrix}$$

From our elementwise definition of vector addition, we can construct the zero element for the vectors. We know from the field axioms that $\mathbf{x} + \mathbf{0} = \mathbf{x}$, so the zero element must be a vector of the same dimension with all zero entries.

Notice that each set of n-dimensional vectors has its own zero element. In $\mathbb{R}^2$, $\mathbf{0} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. In $\mathbb{R}^3$, $\mathbf{0} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$.

$$\mathbf{x} + \mathbf{0} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} x_1 + 0 \\ x_2 + 0 \\ \vdots \\ x_n + 0 \end{pmatrix} = \mathbf{x}$$

1.4 Vector Multiplication is not Elementwise

What happens when we try to define multiplication as an elementwise operation? For example

$$\begin{pmatrix} -1 \\ 0 \\ 4 \end{pmatrix} \times \begin{pmatrix} 0 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \times 0 \\ 0 \times 2 \\ 4 \times 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \mathbf{0}.$$

This is bad. Very bad. Here we have an example where $\mathbf{xy} = \mathbf{0}$, but neither $\mathbf{x}$ nor $\mathbf{y}$ is the zero element $\mathbf{0}$. This is a direct violation of a corollary of the field axioms, so **elementwise vector multiplication is not a valid algebraic operation**.

On the bright side, if vectors were a field this class would be far too short.

Sadly, vectors are not a field. There is no way to define multiplication using only vectors that satisfies the field axioms. Nor is there anything close to a complete set of multiplicative inverses, or even the element **1**. Instead, we will settle for a weaker result—showing that vectors live in a *normed inner product space*. The concepts of a vector norm and inner product will let us create most of the operations and elements that vectors need to be a field.

Example 1.2: Do we need multiplication? When you were first taught to multiply, it was probably introduced as a "faster" method of addition, i.e. $4 \times 3 = 3+3+3+3$. If so, why do we need multiplication as a separate requirement for fields? Couldn't we simply require the addition operator and construct multiplication from it? The answer is no, for two reasons. First, the idea of multiplication as a shortcut for addition only makes sense when discussing the nonnegative integers. What, for example, does it mean to have -2.86×-3.2? What do -2.86 or -3.2 groups look like in terms of addition?

Also, the integers are not a field!

Second, we must realize that multiplication is a much stronger relationship between numbers. To understand why, we should start talking about the "linear" part of linear algebra. ■

1.5 Linear Systems

Linear systems have two special properties.

1. **Proportionality**. If the input to a linear system is multiplied by a scalar, the output is multiplied by the same scalar: $f(kx) = kf(x)$.
2. **Additivity**. If two inputs are added, the result is the sum of the original outputs: $f(x_1 + x_2) = f(x_1) + f(x_2)$.

We can combine both of these properties into a single condition for linearity.

Definition. *A system f is linear if and only if*

$$f(k_1x_1 + k_2x_2) = k_1f(x_1) + k_2f(x_2)$$

for all inputs x_1 and x_2 and scalars k_1 and k_2.

Consider a very simple function, $f(x) = x + 3$. Is this function linear? First we calculate the lefthand side of the definition of linearity.

$$f(k_1x_1 + k_2x_2) = k_1x_1 + k_2x_2 + 3$$

We compare this to the righthand side.

$$\begin{aligned} k_1f(x_1) + k_2f(x_2) &= k_1(x_1 + 3) + k_2(x_2 + 3) \\ &= k_1x_1 + k_2x_2 + 3(k_1 + k_2) \\ &\neq f(k_1x_1 + k_2x_2) \end{aligned}$$

This does not follow the definition of linearity. The function $f(x) = x + 3$ is not linear. Now let's look at a simple function involving multiplication: $f(x) = 3x$. Is this function linear?

$$\begin{aligned} f(k_1x_1 + k_2x_2) &= 3(k_1x_1 + k_2x_2) \\ &= k_1(3x_1) + k_2(3x_2) \\ &= k_1f(x_1) + k_2f(x_2) \end{aligned}$$

The function involving multiplication is linear.

These results might not be what you expected, at least concerning the nonlinearity of functions of the form $f(x) = x + b$. This is probably because in earlier math courses you referred to equations of straight lines ($y = mx + b$) as linear equations. In fact, any equation of this form (with $b \neq 0$) is called *affine*, not linear.

Truly linear functions have the property that $f(0) = 0$. Addition is, in a way, not "strong" enough to drive a function to zero. The expression $x + y$ is zero only when both x and y are zero. By contrast, the product xy is zero when either x or y is zero.

This follows from proportionality. If $f(k0) = kf(0)$ for all k, then $f(0)$ must equal zero.

1.6 Vector Norms

One of the nice properties of the real numbers is that they are *well ordered*. Being well ordered means that for any two real numbers, we can determine which number is larger (or if the two numbers are equal). Well orderedness allows us to make all sorts of comparisons between the real numbers.

Vectors are not well ordered. Consider the vectors $\begin{pmatrix} 3 \\ 4 \end{pmatrix}$ and $\begin{pmatrix} 5 \\ 2 \end{pmatrix}$. Which one is larger? Each vector has one element that is larger than the other (4 in the first, 5 in the second). There is no unambiguous way to place all vectors in order.

This doesn't stop us from making comparisons between vector quantities. Consider velocity, which, contrary to how many people use the term, is a vector. Since vectors are not ordered, we should not be able to compare velocities. Instead, we often compare speeds, which are the magnitude of the velocity vectors. Traveling 30 mph due north and 30 mph due east are technically two different velocities. However, they have the same magnitude (30 mph), so most people consider them equivalent.

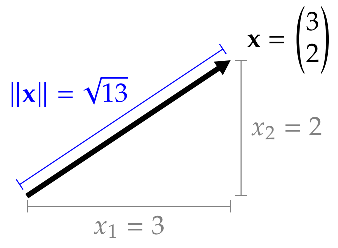

$\mathbf{x} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$
$\|\mathbf{x}\| = \sqrt{13}$
$x_2 = 2$
$x_1 = 3$

Figure 1.1: The vector norm.

Vector magnitudes are calculated by taking a *norm* of the vector. There are many different kinds of norms, but the most commonly used norm is the 2-norm (or Euclidean or Pythagorean norm). We will refer to the 2-norm as simply "the norm" unless we state otherwise. If we treat a vector as a point in n-dimensional space, the norm is the length of the arrow drawn from the origin to that point. We use a pair of two vertical bars ($\|\cdot\|$) to represent the norm. This differentiates the norm from the absolute value (which is, in fact, the 1-norm). Sometimes we use a subscript to identify which norm we are taking, i.e. $\|\mathbf{x}\|_2$ is the 2-norm of $\mathbf{x}$.

In 2D, the norm corresponds to the hypotenuse of the right triangle with sides equivalent to the two elements in the vector—hence the name "Pythagorean norm" since the norm can be calculated by the Pythagorean theorem. In higher dimensions, we generalize the Pythagorean definition of the norm to

$$\|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

In one dimension, taking the 2-norm yields the same result as taking the absolute value:

$$\|-3\| = \sqrt{(-3)^2} = 3 = |-3|.$$

Be careful not to confuse the 2-norm and the absolute value, as they are not the same thing. Their equivalence in one dimension is a coincidence. However, the absolute value is a norm—it returns the magnitude of a number, but strips away the direction (negative or positive).

There are two useful properties of norms that derive directly from its definition. These properties must be true of all norms, not just the 2-norm.

1. **Nonnegativity**. $\|\mathbf{x}\| \geq 0$
2. **Zero Identity**. $\|\mathbf{x}\| = 0 \Leftrightarrow \mathbf{x} = \mathbf{0}$

There are other properties that define norms, such as the triangle inequality ($\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$) or scaling ($\|k\mathbf{x}\| = |k|\,\|\mathbf{x}\|$). The nonnegativity and zero identity properties will be the most useful in this book.

1.6.1 Generalized Norms

We mentioned above that the Euclidean (or 2-norm) is only one type of norm. Some common norms include:

1. The **1-norm**, taxicab, or Manhattan norm, is the sum of the absolute values of the element in a vector:

   $$\|\mathbf{x}\|_1 \equiv |x_1| + |x_2| + \cdots + |x_n|.$$

   The "taxicab" name comes from distance you'd travel between two points when driving in cities with a grid street layout (like Manhattan). If you can't drive diagonally through any city blocks, the distance between two points is the 1-norm.

2. The **∞-norm** is the absolute value of the largest element in the vector:

   $$\|\mathbf{x}\|_\infty \equiv \max\{|x_1|, |x_2|, \ldots, |x_n|\}.$$

3. The **0-norm**, also called the cardinality of a vector, is the number of nonzero element in the vector:

   $$\|\mathbf{x}\|_0 \equiv \#\text{ of nonzero } x_i.$$

   The 0-norm is not a true norm since it does not satisfy all of the properties of norms. Still, the 0-norm is useful in many machine learning applications, so we include it here.

All of the norms listed above are members of the family of p-norms. In general, the p-norm of an n-dimensional vector $\mathbf{x}$ is

$$\|\mathbf{x}\|_p \equiv \left(\sum_{i=1}^{n} x_i^p\right)^{1/p} .$$

Setting $p = 2$ gives the Euclidian norm, and $p = 1$ is the taxicab norm. As p approaches infinity, the p norm becomes the ∞-norm. Setting $p = 0$ would give the 0-norm if we overlook the $1/0$ that appears in the exponent—again, the 0-norm is not a true norm, so some mathematical leniency is needed.

1.6.2 Normalized (Unit) Vectors

A vector contains information about both its magnitude and its orientation. We've seen how to extract the magnitude as a scalar from the vector by taking the norm. Is it possible to similarly separate out the vector's orientation? Yes, by *normalizing* the vector. A normalized vector (or *unit vector*) is any vector with magnitude equal to one. We can convert a vector to a normalized vector by dividing each element by the vector's magnitude. For example

$$\mathbf{x} = \begin{pmatrix} 3 \\ -4 \end{pmatrix} \Rightarrow \|\mathbf{x}\| = \sqrt{3^2 + (-4)^2} = 5.$$

The normalized unit vector ($\hat{\mathbf{x}}$) is

$$\hat{\mathbf{x}} = \begin{pmatrix} 3/\|\mathbf{x}\| \\ -4/\|\mathbf{x}\| \end{pmatrix} = \begin{pmatrix} 3/5 \\ -4/5 \end{pmatrix} .$$

We use the hat symbol (ˆ) over a unit vector to remind us that it has been normalized.

Intuitively, the idea of a normalized unit vector as a direction makes sense. If a vector is a product of both a magnitude and a direction, then the vector divided by the magnitude (the norm) should equal a direction (a unit vector), as shown in Figure 1.2.

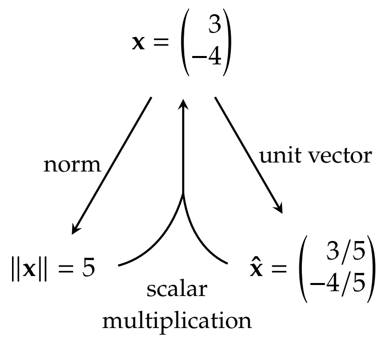


Figure 1.2: Vectors separate into a length (norm) and direction (unit vector). The length and direction can be combined by scalar multiplication

1.7 Scalar Vector Multiplication

We saw earlier that elementwise multiplication was a terrible idea. In fact, defining multiplication this way violates a corollary of the field axioms ($\mathbf{xy} = \mathbf{0}$ implies that $\mathbf{x} = \mathbf{0}$ or $\mathbf{y} = \mathbf{0}$). However, elementwise multiplication does work in one case—*scalar* *multiplication*, or the product between a scalar (real number) and a vector:

$$k\mathbf{x} = k\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} kx_1 \\ kx_2 \\ \vdots \\ kx_n \end{pmatrix}$$

where k is a scalar real number. Notice that scalar multiplication does not suffer from the same problem as elementwise vector multiplication. If $k\mathbf{x} = 0$, then either the scalar k equals zero or the vector $\mathbf{x}$ must be the zero vector.

What happens when you multiply a vector by a scalar? For one, the norm changes:

Remember that $\sqrt{k^2} = |k|$, not k itself. We consider the square root to be the positive root.

$$\begin{aligned} \|k\mathbf{x}\| &= \sqrt{(kx_1)^2 + (kx_2)^2 + \cdots + (kx_n)^2} \\ &= \sqrt{k^2(x_1^2 + x_2^2 + \cdots + x_n^2)} \\ &= |k|\,\|\mathbf{x}\| \end{aligned}$$

Scalar multiplication scales the length of a vector by the scalar. If the scalar is negative, the direction of the vector "reverses".

1.8 Dot (Inner) Product

Now we see why we use the symbol $\times$ for multiplication; the dot $(\cdot)$ is reserved for the dot product.

One way to think of the product of two vectors is to consider the product of their norms (magnitudes). Such operations are common in mechanics. Work, for example, is the product of force and displacement. However, simply multiplying the magnitude of the force vector and the magnitude of the displacement vector disregards the orientation of the vectors. We know from physics that only the component of the force aligned with the displacement should count.

In general, we want an operation that multiplies the magnitude of one vector with the *projection* of a second vector onto the first. We call this operation the *inner product* or the *dot product*. Geometrically, the dot product is a measure of both the product of the vectors' magnitudes and how well they are aligned. For vectors $\mathbf{x}$ and $\mathbf{y}$ the dot product is defined

$$\mathbf{x} \cdot \mathbf{y} = \|\mathbf{x}\|\,\|\mathbf{y}\| \cos\theta$$

where θ is the angle between the vectors.

If two vectors are perfectly aligned, $\theta = 0°$ and the dot product is simply the product of the magnitudes. If the two vectors point in exactly opposite directions,

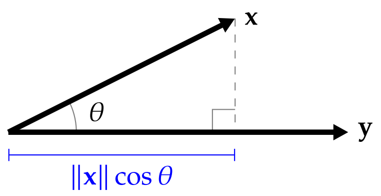


Figure 1.3: The projection of $\mathbf{x}$ onto $\mathbf{y}$ is a scalar equal to $\|\mathbf{x}\|\cos\theta$.

$\theta = 180°$ and the dot product is -1 times the product of the magnitudes. If the vectors are *orthogonal*, the angle between them is $90°$, so $\cos\theta = 0$ and the dot product is zero. Thus, **the dot product of two vectors is zero if and only if the vectors are orthogonal**.

1.8.1 Computing the Dot Product

We know how to calculate norms, but how do we calculate the angle between two n-dimensional vectors? The answer is that we don't need to. There is an easier way to calculate $\mathbf{x}\cdot\mathbf{y}$ than the formula $\|\mathbf{x}\|\,\|\mathbf{y}\|\cos\theta$.

First, we need to define a special set of vectors—the unit vectors $\hat{\mathbf{e}}_i$. Unit vectors have only a single nonzero entry, a 1 at element i. For example,

$$\hat{\mathbf{e}}_1 = \begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix},\ \hat{\mathbf{e}}_2 = \begin{pmatrix}0\\1\\\vdots\\0\end{pmatrix},\ \ldots,\ \hat{\mathbf{e}}_n = \begin{pmatrix}0\\0\\\vdots\\1\end{pmatrix}.$$

Every vector can be written as a sum of scalar products with unit vectors. For example,

$$\begin{aligned}\begin{pmatrix}-3\\6\\2\end{pmatrix} &= -3\begin{pmatrix}1\\0\\0\end{pmatrix} + 6\begin{pmatrix}0\\1\\0\end{pmatrix} + 2\begin{pmatrix}0\\0\\1\end{pmatrix}\\ &= -3\hat{\mathbf{e}}_1 + 6\hat{\mathbf{e}}_2 + 2\hat{\mathbf{e}}_3\end{aligned}$$

In general

$$\begin{aligned}\mathbf{x} &= x_1\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix} + x_2\begin{pmatrix}0\\1\\\vdots\\0\end{pmatrix} + \cdots + x_n\begin{pmatrix}0\\0\\\vdots\\1\end{pmatrix}\\ &= \sum_{i=1}^{n} x_i\hat{\mathbf{e}}_i\end{aligned}$$

Now let's compute the dot product using the unit vector expansion for **x** and **y**.

$$\begin{aligned}\mathbf{x}\cdot\mathbf{y} = &\ (x_1\hat{\mathbf{e}}_1 + x_2\hat{\mathbf{e}}_2 + \cdots + x_n\hat{\mathbf{e}}_n)\cdot\left(y_1\hat{\mathbf{e}}_1 + y_2\hat{\mathbf{e}}_2 + \cdots + y_n\hat{\mathbf{e}}_n\right)\\ = &\ x_1\hat{\mathbf{e}}_1\cdot(y_1\hat{\mathbf{e}}_1 + y_2\hat{\mathbf{e}}_2 + \cdots + y_n\hat{\mathbf{e}}_n)\\ &+ x_2\hat{\mathbf{e}}_2\cdot(y_1\hat{\mathbf{e}}_1 + y_2\hat{\mathbf{e}}_2 + \cdots + y_n\hat{\mathbf{e}}_n)\\ &+\cdots\\ &+ x_n\hat{\mathbf{e}}_n\cdot(y_1\hat{\mathbf{e}}_1 + y_2\hat{\mathbf{e}}_2 + \cdots + y_n\hat{\mathbf{e}}_n)\end{aligned}$$

Consider each of the terms $x_i\hat{\mathbf{e}}_i\cdot(y_1\hat{\mathbf{e}}_1 + y_2\hat{\mathbf{e}}_2 + \cdots + y_n\hat{\mathbf{e}}_n)$. By distribution, this is equivalent to

$$x_iy_1\hat{\mathbf{e}}_i\cdot\hat{\mathbf{e}}_1 + \cdots + x_iy_j\hat{\mathbf{e}}_i\cdot\hat{\mathbf{e}}_j + \cdots + x_iy_n\hat{\mathbf{e}}_i\cdot\hat{\mathbf{e}}_n.$$

The only nonzero term in this entire summation is $x_iy_i\hat{\mathbf{e}}_i\cdot\hat{\mathbf{e}}_i$, which equals x_iy_i. The dot product reduces to

Think about the dot product $\hat{\mathbf{e}}_i\cdot\hat{\mathbf{e}}_j$. If $i = j$, this product is 1 since $\|\hat{\mathbf{e}}_i\| = \left\|\hat{\mathbf{e}}_j\right\| = 1$ and $\theta = 0°$. However, if $i \neq j$, the vectors are always orthogonal and the dot product is 0.

$$\begin{aligned}\mathbf{x}\cdot\mathbf{y} &= x_1y_1 + x_2y_2 + \cdots + x_ny_n\\ &= \sum_{i=1}^{n} x_iy_i\end{aligned}$$

Although the above formula is convenient for computing dot products, it lacks the intuition of our previous method ($\mathbf{x}\cdot\mathbf{y} = \|\mathbf{x}\|\,\|\mathbf{y}\|\cos\theta$). Whenever you use the former method, be sure to remember that you're really calculating the product magnitudes after one vector is projected onto the other.

1.8.2 Dot Product Summary

- Dot products are defined between two vectors with the same dimension.
- Dot products return a scalar from two vectors. This is the projected product of the two magnitudes.
- $\mathbf{x}\cdot\mathbf{y} = \|\mathbf{x}\|\,\|\mathbf{y}\|\cos\theta = x_1y_1 + x_2y_2 + \cdots + x_ny_n$
- $\mathbf{x}\cdot\mathbf{y} = 0 \Leftrightarrow$ **x** and **y** are orthogonal.
- Dot products commute [$\mathbf{x}\cdot\mathbf{y} = \mathbf{y}\cdot\mathbf{x}$] and distribute over addition [$\mathbf{x}\cdot(\mathbf{y}+\mathbf{z}) = \mathbf{x}\cdot\mathbf{y}+\mathbf{x}\cdot\mathbf{z}$]. There is no such thing as an associative property for dot products since the dot product of three vectors is not defined. The dot product between the first two vectors would produce a scalar, and the dot product between this scalar and the third vector is not defined.

Chapter 2

Matrices

2.1 Matrix/Vector Multiplication

Let's take stock of the operations we've defined so far.

- The **norm** (magnitude) maps a vector to a scalar. ($\mathbb{R}^n \mapsto \mathbb{R}$)
- The **scalar product** maps a scalar and a vector to a new vector ($\mathbb{R} \times \mathbb{R}^n \mapsto \mathbb{R}^n$), but can only scale the magnitude of the vector (or flip it if the scalar is negative).
- The **dot product** maps to vectors to a scalar ($\mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}$) by projecting one onto the other and multiplying the resulting magnitudes.

All of these operations appear consistent with the field axioms. Unfortunately, we still do not have a true multiplication operation—one that can transform any vector into any other vector. Can we define such an operation using only the above methods?

Let's construct a new vector **y** from the vector **x**. To be as general as possible we should let each element in **y** be an arbitrary linear combination of the elements in **x**:

$$\begin{aligned} y_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ y_2 &= a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ &\vdots \\ y_n &= a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n \end{aligned}$$

The scalars a_{ij} determine the relative weight of x_j when constructing y_i. There are n^2 scalars required to unambiguously map **x** to **y**. For convenience, we collect the set of weights into an n by n numeric grid called a *matrix*.

If **A** is a real-valued matrix with dimensions $m \times n$, we say $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\dim(\mathbf{A}) = m \times n$.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

Using the matrix **A**, we can write the above system of equations as

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

or, more succinctly

$$\mathbf{y} = \mathbf{A}\mathbf{x}.$$

Writing the equations in this *matrix form* requires a new definition of multiplication between the matrix **A** and the vector **x**. For example, we can write the following linear system

$$\begin{aligned} x_1 - 2x_2 + x3 &= 0 \\ 3x_1 - x_3 &= 4 \\ x_2 + 3x_3 &= -1 \end{aligned}$$

in matrix form as

$$\begin{pmatrix} 1 & -2 & 1 \\ 3 & 0 & -1 \\ 0 & 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ -1 \end{pmatrix}.$$

These equations transform the input vector **x** into a new vector with elements 0, 4, and −1. The first element (0) is the dot product between the first row of the matrix and the vector:

$$\begin{pmatrix} 1 & -2 & 1 \\ 3 & 0 & -1 \\ 0 & 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ -1 \end{pmatrix}.$$

A subtle technical point: we consider a single row of a matrix to be a vector—in fact a column vector—to allow the dot product with the input vector. The distinction between row and column vectors is often glossed over by humans but is important to computers. (See below when we talk about the matrix transpose.)

Similarly, the second element in the output vector (4) is the dot product between the second row of the matrix and the input vector,

$$\begin{pmatrix} 1 & -2 & 1 \\ 3 & 0 & -1 \\ 0 & 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ -1 \end{pmatrix}$$

and the third entry in the output is the dot product between the third row and the input vector:

$$\begin{pmatrix} 1 & -2 & 1 \\ 3 & 0 & -1 \\ 0 & 1 & 3 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ -1 \end{pmatrix}.$$

2.2 Matrix Multiplication

What we have been calling "vectors" all along are really just matrices with only one column. Thinking of vectors as matrices lets us write a simple, yet powerful, definition of multiplication.

Definition. *The product of matrices* **AB** *is a matrix* **C***; each element* c_{ij} *in* **C** *is the dot product between the* i*th row in* **A** *and the* j*th column in* **B***:*

$$c_{ij} = \mathbf{A}(i,:) \cdot \mathbf{B}(:,j).$$

This definition is perfectly consistent with matrix/vector multiplication if we view a vector as a single-column matrix.

Example 2.1. Let's multiply the matrices

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & -2 \\ 4 & 3 & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{B} = \begin{pmatrix} 0 & 1 \\ -2 & 3 \\ 1 & 1 \end{pmatrix}.$$

We'll call the resulting matrix **C**. The entry c_{11} is the dot product between row 1 of matrix **A** and column 1 of matrix **B**.

$$\begin{pmatrix} -2 & \\ & \end{pmatrix} = \begin{pmatrix} 1 & 0 & -2 \\ 4 & 3 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ -2 & 3 \\ 1 & 1 \end{pmatrix}$$

$$c_{11} = \mathbf{A}(1,:) \cdot \mathbf{B}(:,1) = 1 \times 0 + 0 \times (-2) + -2 \times 1 = -2$$

The entry c_{12} is the dot product between row 1 of matrix **A** and column 2 of matrix **B**.

$$\begin{pmatrix} -2 & -1 \\ & \end{pmatrix} = \begin{pmatrix} 1 & 0 & -2 \\ 4 & 3 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ -2 & 3 \\ 1 & 1 \end{pmatrix}$$

$$c_{12} = \mathbf{A}(1,:) \cdot \mathbf{B}(:,2) = 1 \times 1 + 0 \times 3 + -2 \times 1 = -1$$

The entry c_{21} is the dot product between row 2 of matrix **A** and column 1 of matrix **B**.

$$\begin{pmatrix} -2 & -1 \\ -5 & \end{pmatrix} = \begin{pmatrix} 1 & 0 & -2 \\ 4 & 3 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ -2 & 3 \\ 1 & 1 \end{pmatrix}$$

$$c_{21} = \mathbf{A}(2,:) \cdot \mathbf{B}(:,1) = 4 \times 0 + 3 \times (-2) + 1 \times 1 = -5$$

Finally, the entry c_{22} is the dot product between row 2 of matrix **A** and column 2 of matrix **B**.

$$\begin{pmatrix} -2 & -1 \\ -5 & 14 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -2 \\ 4 & 3 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ -2 & 3 \\ 1 & 1 \end{pmatrix}$$

$$c_{22} = \mathbf{A}(2,:) \cdot \mathbf{B}(:,2) = 4 \times 1 + 3 \times 3 + 1 \times 1 = 14$$

■

The length of a *row* in **A** is equal to the number of *columns* in **A**. The length of a single *column* in **B** is equal to the number of *rows* in **B**.

In the previous example, the matrices **A** and **B** did not have the same dimensions. If **C** = **AB**, each entry in the matrix **C** is the dot product between a row of **A** and a column of **B**. Thus the number of columns in **A** must equal the number of rows in **B** since the dot product is only defined for vectors of the same dimension.

MATLAB returns an error that "matrix dimensions must agree" when multiplying non-conformable objects.

Any matrices **A** and **B** are *conformable* for multiplication if the number of columns in **A** matches the number of rows in **B**. If the dimensions of **A** are $m \times n$ and the dimensions of **B** are $n \times p$, then the product will be a matrix of dimensions $m \times p$. To check if two matrices are conformable, write the dimensions next to each other:

For the system $\mathbf{y} = \mathbf{A}\mathbf{x}$, if dim(**A**)=$m \times n$ and dim(**x**)=$n \times 1$, then dim(**y**) = $m \times 1$, i.e. **y** is a column vector in $\mathbb{R}^m$.

$$\overbrace{(m \times \underbrace{n)(n}_{\text{must agree}} \times p)}^{\text{dimensions of product}}$$

The inner two numbers (here n and n) must be the same. The dimensions of the product matrix are the outer two numbers (m and p).

Matrix multiplication is associative [**ABC**= (**AB**)**C**= **A**(**BC**)] and distributive over addition [**A**(**B**+**C**) = **AB**+**AC**] provided **A**, **B**, and **C** are all conformable. However, matrix multiplication is **not** commutative. To see why, consider an $(m \times n)$ matrix **A** and an $(n \times p)$ matrix **B**. The product **AB** is an $m \times p$ matrix, but the product **BA** is not conformable since $p \neq m$. Even if **BA** were conformable, it is not the same as the product **AB**:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 0 & 1 \\ -1 & 2 \end{pmatrix}$$

$$\mathbf{AB} = \begin{pmatrix} 1\times 0 + 2\times(-1) & 1\times 1 + 2\times 2 \\ 3\times 0 + 4\times(-1) & 3\times 1 + 4\times 2 \end{pmatrix} = \begin{pmatrix} -2 & 5 \\ -4 & 11 \end{pmatrix}$$

$$\mathbf{BA} = \begin{pmatrix} 0\times 1 + 1\times 3 & 0\times 2 + 1\times 4 \\ -1\times 1 + 2\times 3 & -1\times 2 + 2\times 4 \end{pmatrix} = \begin{pmatrix} 3 & 4 \\ 5 & 6 \end{pmatrix}$$

2.3 Identity Matrix

We need to find an element that serves as **1** for vectors. The field axioms define this element by the property that $1 \times x = x$ for all x in the field. For vectors, we defined multiplication to involve matrices, so the element **1** will be a matrix that we call the *identity matrix*, or **I**. We require that

$$\mathbf{Ix} = \mathbf{xI} = \mathbf{x}$$

for all **x**. Assuming that **x** is n-dimensional, **I** must have n columns to be conformable. Also, the output of **Ix** has n elements, so **I** must have n rows. Therefore, we know that **I** is a square $n \times n$ matrix whenever **x** has dimension n.

Consider the first row of **I**, i.e. $\mathbf{I}(1,:)$. We know from the definition of **I** that $\mathbf{I}(1,:) \cdot \mathbf{x} = x_1$, so $\mathbf{I}(1,:) = (1\,0\,0 \cdots 0)$. For the second row, $\mathbf{I}(2,:) \cdot \mathbf{x} = x_2$, so $\mathbf{I}(2,:) = (0\,1\,0 \cdots 0)$. In general, the ith row of **I** has a 1 at position i and zeros everywhere else

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & & 0 \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}.$$

The identity matrix **I** for a vector in $\mathbb{R}^n$ is an $n \times n$ matrix with ones along the diagonal and zeroes everywhere else.

In $\mathbb{R}^1$, $\mathbf{I} = (1)$, which behaves like the real number 1.

Our definition of the identity matrix also works for matrix multiplication. For any square matrix **A**

$$\mathbf{IA} = \mathbf{AI} = \mathbf{A}.$$

The identity matrix works on "both sides" only if the matrix **A** is square. If the matrix **A** were not square, we would need separate identify matrices for left and right multiplication so the dimensions are compatible for multiplication. Consider a nonsquare matrix $\mathbf{A}_{5\times 3}$ with five rows and three columns. It is still true that $\mathbf{I}_{5\times 5}\mathbf{A}_{5\times 3} = \mathbf{A}_{5\times 3}$ and $\mathbf{A}_{5\times 3}\mathbf{I}_{3\times 3} = \mathbf{A}_{5\times 3}$; however, notice how we used a 5×5 identify matrix for left multiplication and a 3×3 identity matrix for right multiplication.

2.4 Matrix Transpose

The transpose operator flips the rows and columns of a matrix. The element a_{ij} in the original matrix becomes element a_{ji} in the transposed matrix. The transpose operator is a superscript "T", as in $\mathbf{A}^\mathsf{T}$. A transposed matrix is reflected about a diagonal drawn from the upper left to the lower right corner.

Other notations for the matrix transpose include $\mathbf{A}^t$ and $\mathbf{A}'$. The latter is used in MATLAB.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}^\mathsf{T} = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

Transposing an $m \times n$ matrix creates an $n \times m$ matrix.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}^\mathsf{T} = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

Transposing a column vector creates a row vector, and vice versa.

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}^\mathsf{T} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}, \quad \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}^\mathsf{T} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Why do we introduce the matrix transpose now? There are connections between the dot product, matrix multiplication, and transposition.

2.4.1 Transposition and the Dot Product

For any vectors $\mathbf{x}$ and $\mathbf{y}$:

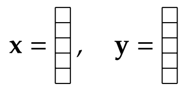
$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^\mathsf{T}\mathbf{y}$$

The dot product between $\mathbf{x}$ and $\mathbf{y}$ is equivalent to the product between the transposed vector $\mathbf{x}$ and $\mathbf{y}$. Said simply, $\mathbf{x}$ "dot" $\mathbf{y}$ equals $\mathbf{x}^\mathsf{T}$ "times" $\mathbf{y}$.

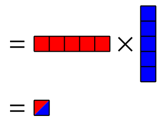
Notice that $\mathbf{x} \cdot \mathbf{y} \neq \mathbf{x}\mathbf{y}$ if we forget to transpose $\mathbf{x}$. Be careful to distinguish dot products and multiplication.

Why does $\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^\mathsf{T}\mathbf{y}$? Computing the dot product between the vectors $\mathbf{x}$ and $\mathbf{y}$ requires both have the same dimension, which we'll call n. Both vectors are column vectors, so we can also view them as a single-column matrix with dimensions $n \times 1$. The transposed vector $\mathbf{x}^\mathsf{T}$ is therefore a row vector with dimensions $1 \times n$, and we know the product of an $1 \times n$ matrix with an $n \times 1$ matrix will be a 1×1 matrix—which is a scalar. By definition of matrix multiplication, $\mathbf{x}^\mathsf{T}\mathbf{y}$ is the dot product between the first row of $\mathbf{x}^\mathsf{T}$ (which has only one row) and the first column of $\mathbf{y}$

(which has only one column). Thus, $\mathbf{x}^\mathsf{T}\mathbf{y}$ is the dot product between vectors $\mathbf{x}$ and $\mathbf{y}$.



$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^\mathsf{T}\mathbf{y}$$



We will use the relationship between dot products and matrix multiplication to solve linear statistical models later in this book. Until then, keep in mind the convergence between these operations.

2.4.2 Transposition and Matrix Multiplication

An interesting identity relates matrix multiplication and matrix transposition. For any matrices $\mathbf{A}$ and $\mathbf{B}$,

$$(\mathbf{AB})^\mathsf{T} = \mathbf{B}^\mathsf{T}\mathbf{A}^\mathsf{T}$$

Said in words, if we want to take the transpose of the product of two matrices, we could equivalently transpose the two matrices first and them multiply them **in reverse order**. The same idea holds when we transpose the product of several matrices. Consider a set of k matrices $\mathbf{A}_1$, $\mathbf{A}_2$, …, $\mathbf{A}_k$. Then

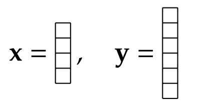
$$(\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_{k-1}\mathbf{A}_k)^\mathsf{T} = \mathbf{A}_k^\mathsf{T}\mathbf{A}_{k-1}^\mathsf{T}\cdots\mathbf{A}_2^\mathsf{T}\mathbf{A}_1^\mathsf{T}.$$

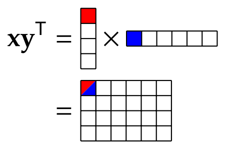
Notice two things about this identify. First, many students confuse this identify with the commutative property. As we've said before, matrix multiplication does not commute. The above identity is unrelated to the commutative property. Second, the identity holds for square and non-square matrices provided the matrices are compatible for multiplication. Try to prove to yourself that if the product $\mathbf{AB}$ is compatible, then the product $\mathbf{B}^\mathsf{T}\mathbf{A}^\mathsf{T}$ is also be compatible.

2.5 Outer Product and Trace

If the product $\mathbf{x}^\mathsf{T}\mathbf{y}$ is a scalar, what about the product $\mathbf{x}\mathbf{y}^\mathsf{T}$? Imagine $\mathbf{x}$ and $\mathbf{y}$ have n elements, making them $n \times 1$ matrices. (Both $\mathbf{x}$ and $\mathbf{y}$ must have the same number of elements or multiplication is impossible.) Then $\mathbf{x}\mathbf{y}^\mathsf{T}$ is the product of an $n \times 1$ matrix and a $1 \times n$ matrix. The product is therefore an $n \times n$ matrix. While $\mathbf{x}^\mathsf{T}\mathbf{y}$ and $\mathbf{x}\mathbf{y}^\mathsf{T}$ may look similar, they produce wildly different results. The former—the dot product—is also known as the *inner product* since it collapses to vectors into a scalar. The latter ($\mathbf{x}\mathbf{y}^\mathsf{T}$) is called the *outer product* since it expands vectors outward into a matrix. While the inner (dot) product requires both vectors have the same dimension, the outer product is compatible with any two vectors.

The matrix formed by the outer product contains all pairwise products between the elements in the two vectors.





There is an interesting connection between the outer and inner products. The *trace* of a square matrix is the sum of its diagonal elements. For example, the matrix

$$\mathbf{A} = \begin{pmatrix} -1 & 3 & 4 \\ 0 & 2 & -3 \\ 1 & 0 & 6 \end{pmatrix}$$

has trace

$$\operatorname{tr}(\mathbf{A}) = -1 + 2 + 6 = 7$$

The trace of a matrix $\mathbf{A}$ is written $\operatorname{tr}(\mathbf{A})$.

For a challenge, show that the trace of the outer product of two vectors is equal to the inner product of the same vectors, or

$$\operatorname{tr}(\mathbf{x}\mathbf{y}^\mathsf{T}) = \mathbf{x}^\mathsf{T}\mathbf{y}$$

Notice that this fact only works when the vectors $\mathbf{x}$ and $\mathbf{y}$ have the same dimension. Otherwise, the inner product is not defined. Similarly, if $\mathbf{x}$ and $\mathbf{y}$ did not have the same dimension, the outer product $\mathbf{x}\mathbf{y}^\mathsf{T}$ would not be a square matrix so the trace would not be defined.

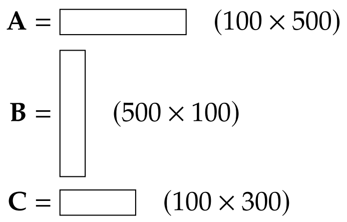
2.6 Computational Complexity of Matrix Multiplication

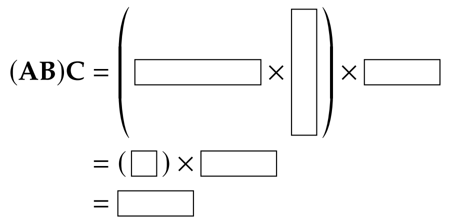
You may be feeling that matrix multiplication is tedious work. Indeed, multiplying two matrices requires computing many dot products, which themselves requires many scalar multiplications. In linear algebra it is important to understand how many operations are required to complete an operation, especially as the matrices become very large. The *computational complexity* of an operation is the relationship between the number of computations and the size of the operands. Let's analyze the computational complexity of matrix multiplication.

Imagine we're multiplying two $n \times n$ matrices. The product matrix will also have dimensions $n \times n$. Each entry in the product matrix is computed with a dot product between a row in the first factor and a column in the second factor. This dot product requires n scalar multiplications and $n-1$ additions. Overall, there are n^2 entries in the product matrix. If each requires a dot product, the total number of operations is $n^2 n$ multiplications and $n^2(n-1)$ additions. We say the number of operations required to multiply two $n \times n$ matrices is $\mathcal{O}(n^3)$. We can perform a similar analysis with two non-square matrices. Multiplying an $m \times n$ matrix by a $n \times p$ matrix requires $\mathcal{O}(mnp)$ operations.

The $\mathcal{O}$, or "big-O" notation indicates the rate of growth of a function for large values. Any polynomial of degree d is $\mathcal{O}(d)$.

The number of operations required to multiply three or more matrices depends on the order of the multiplication. Matrix multiplication is associative, so the product **ABC** can be computed as (**AB**)**C** or **A**(**BC**). Let's count the operations for both methods using the following sizes for **A**, **B**, and **C**.

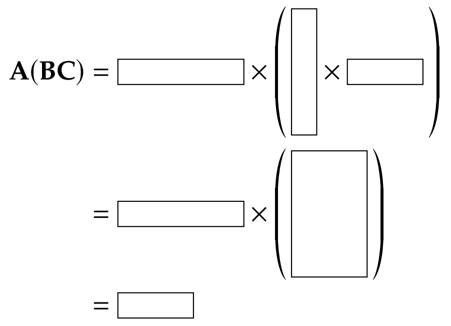




$100 \times 500 \times 100$ operations

$+\, 100 \times 100 \times 300$ operations

$=$ **8,000,000 total operations**



$500 \times 100 \times 300$ operations

$+\, 100 \times 500 \times 300$ operations

$=$ **30,000,000 total operations**

In general, the best order for matrix multiplications depends on the dimensions of the matrices. If the matrices **A**, **B**, **C** have dimensions

$$\begin{aligned}\dim(\mathbf{A}) &= m \times n\\ \dim(\mathbf{B}) &= n \times p\\ \dim(\mathbf{C}) &= p \times q\end{aligned}$$

it is more efficient to compute (**AB**)**C** if

$$mnp + mpq < npq + mnq$$

Otherwise, it is more efficient to compute **A**(**BC**).

Chapter 3

Rotation and Translation Matrices

A common problem in motion biomechanics is determining the position of a multi-bar linkage arm. For example, a two-bar linkage arm is shown in Figure 3.1. The arm contains two rigid bars of lengths l_1 and l_2. The bars are bent at angles θ_1 and θ_2. Given the bar lengths and the angles, can we calculate the position of the end of the arm (the point labeled **p** in Figure 3.1)?

Multi-bar linkages might seem like nightmarish geometry problems steeped in trigonometry; probably because they are. However, this chapter shows how matrix multiplication provides a straightforward, consistent method for solving problems with complex linkage arms. We will introduce two new operations: *rotation* and *translation*. Both rotation and translation can be described using matrix multiplication, and all linkage arms can be assembled from a series of rotations and translations.

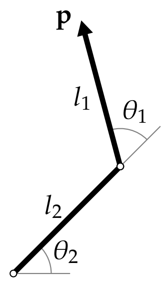


Figure 3.1: A two-bar linkage arm.

3.1 Rotation

We begin by rotating vectors as shown in Figure 3.2. We start with a vector that ends at the point **a**. If we rotate the vector about the origin (leaving its length the same), where is the new endpoint (called **b** in Figure 3.2)? Let's call the angle of rotation θ and define positive rotation ($\theta > 0$) to be in the counter-clockwise direction. The counter-clockwise definition is consistent with the righthand rule from physics. Now we can build a *rotation matrix* as follows:

$$\mathbf{R}(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

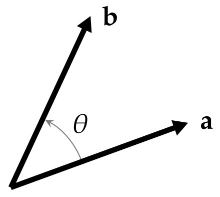


Figure 3.2: The vector **b** is equal to the vector **a** rotated counter-clockwise by the angle θ.

Given an angle θ, the rotation matrix $\mathbf{R}(\theta)$ is a 2×2 matrix containing sines and cosines of θ. Multiplying a vector by a rotation matrix $\mathbf{R}(\theta)$ is equivalent to rotating the vector by θ. Using the labels in Figure 3.2, the position after rotation is

$$\mathbf{b} = \mathbf{R}(\theta)\mathbf{a}$$

Let's do an example where we rotate the vector $\mathbf{a} = \begin{pmatrix}1\\1\end{pmatrix}$ by the angle $\theta = -135°$. Our first step is constructing the rotation matrix:

$$\mathbf{R}(-135°) = \begin{pmatrix}\cos(-135°) & -\sin(-135°)\\ \sin(-135°) & \cos(-135°)\end{pmatrix} = \begin{pmatrix}-\sqrt{2}/2 & \sqrt{2}/2\\ -\sqrt{2}/2 & -\sqrt{2}/2\end{pmatrix}$$

Now we can rotate the vector $\mathbf{a}$ by multiplying it by our rotation matrix.

$$\mathbf{R}(-135°)\mathbf{a} = \begin{pmatrix}-\sqrt{2}/2 & \sqrt{2}/2\\ -\sqrt{2}/2 & -\sqrt{2}/2\end{pmatrix}\begin{pmatrix}1\\1\end{pmatrix} = \begin{pmatrix}0\\-\sqrt{2}\end{pmatrix}$$

The rotated vector points along the negative horizontal axis, as shown in Figure 3.3.

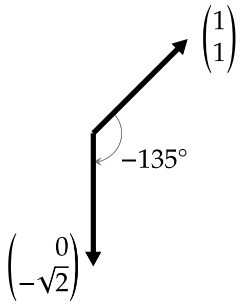

$\begin{pmatrix}1\\1\end{pmatrix}$
$-135°$
$\begin{pmatrix}0\\-\sqrt{2}\end{pmatrix}$

Figure 3.3: Negative angles rotate vectors in the clockwise direction.

3.1.1 Sequential Rotations

We can apply a sequence of rotations using successive multiplications by rotation matrices. Let's rotate a vector $\mathbf{x}$ by angle θ_1 followed by a second rotation of angle θ_2. The first rotation is the product $\mathbf{R}(\theta_1)\mathbf{x}$. Applying the second rotation gives $\mathbf{R}(\theta_2)\,(\mathbf{R}(\theta_1)\mathbf{x})$. We can drop the parentheses by the associative property of matrix multiplication and write the result of both rotations as simply $\mathbf{R}(\theta_2)\mathbf{R}(\theta_1)\mathbf{x}$.

In general, we can rotate a vector $\mathbf{x}$ by k angles $\theta_1, \theta_2, \ldots, \theta_k$ with the expression

$$\mathbf{R}(\theta_k)\cdots\mathbf{R}(\theta_2)\mathbf{R}(\theta_1)\mathbf{x}$$

Pay attention to the order of the rotation matrices. The first rotation $\mathbf{R}(\theta_1)$ appears closest to the vector $\mathbf{x}$. The final rotation $\mathbf{R}(\theta_k)$ is farthest to the left side since it is applied last.

If the matrix $\mathbf{R}(\theta)$ rotates a vector by the angle θ, then the matrix $\mathbf{R}(-\theta)$ should undo the rotation since it rotates the same amount in the opposite direction. Indeed,

Remember that $\sin(-\theta) = -\sin\theta$, $\cos(-\theta) = \cos\theta$, and $\sin^2\theta + \cos^2\theta = 1$.

$$\begin{aligned}
\mathbf{R}(\theta)\mathbf{R}(-\theta) &= \begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix}\begin{pmatrix}\cos(-\theta) & -\sin(-\theta)\\ \sin(-\theta) & \cos(-\theta)\end{pmatrix}\\
&= \begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix}\begin{pmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{pmatrix}\\
&= \begin{pmatrix}\cos^2\theta+\sin^2\theta & \cos\theta\sin\theta-\sin\theta\cos\theta\\ \sin\theta\cos\theta-\cos\theta\sin\theta & \sin^2\theta+\cos^2\theta\end{pmatrix}\\
&= \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = \mathbf{I}
\end{aligned}$$

so

$$\mathbf{R}(\theta)\mathbf{R}(-\theta)\mathbf{x} = \mathbf{I}\mathbf{x} = \mathbf{x}$$

Also, notice that $\mathbf{R}(-\theta)$ is the transpose of $\mathbf{R}(\theta)$. We will explore these special properties of rotation matrices in the coming chapters.

3.2 Translation

The second operation performed by linkage arms is translation, or shifting a point along each axis. In two dimensions, a point can be translated to a new location by shifting it a distance Δx along the horizontal axis and a distance Δy along the vertical axis (see Figure 3.4). Our goal is to find a *translation matrix* $\mathbf{T}(\Delta x, \Delta y)$ that can translate vectors with matrix multiplication. Using the labels in Figure 3.4, we are looking for a matrix $\mathbf{T}(\Delta x, \Delta y)$ such that

$$\mathbf{b} = \mathbf{T}(\Delta x, \Delta y)\mathbf{a}$$

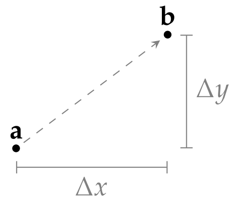


Figure 3.4: Translating point **a** to point **b** includes a horizontal shift (Δx) and a vertical shift (Δy).

Unlike rotation, translation cannot be easily expressed using matrix multiplication. Translation adds a constant to a vector, and this is not a linear operation. Recall from Section 1.5 that adding a constant is an affine operation (like the function $y = x + 3$). Matrix multiplication is a strictly linear operation. For two-dimensional vectors there is no 2×2 matrix that behaves like a translation matrix.

The good news is that we can cheat. We know how to rotate vectors, and translation in two dimensions is equivalent to a rotation in three dimensions. Consider the two dimensional plane shown in Figure 3.5. Our goal is to translate from point **a** on the plane to point **b**, which is also on the plane. If we look beyond the plane and into the third dimension, we see that a vector that points to **a** can be rotated to point to **b**. (Depending on the orientation of the plane, we might need to change the length of the three dimensional vector; this is easily done by scalar multiplication, which is a linear operation.)

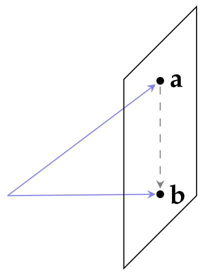


Figure 3.5: Translating point **a** to point **b** in two dimensions is equivalent to rotating the blue vectors in three dimensions.

It is possible to define a 3×3 translation matrix that shifts both the horizontal and vertical dimensions of a vector. The translation matrix is

$$\mathbf{T}(\Delta x, \Delta y) = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix}$$

The 3×3 translation matrix is designed to translate in two dimensions. For two dimensional vectors to be compatible we need to add a third, or "dummy" dimension that contains the value 1. For example, the two dimensional vector $\begin{pmatrix} x \\ y \end{pmatrix}$ becomes the three dimensional dummy vector $\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$. We can use dummy vectors to prove that the translation matrix does, in fact, translate:

$$\mathbf{T}(\Delta x, \Delta y)\mathbf{x} = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + \Delta x \\ y + \Delta y \\ 1 \end{pmatrix}$$

Notice how the translation matrix regenerates the value 1 in the dummy dimension. The value of the dummy dimension should never change upon either translation or rotation. It is an arbitrary offset, analogous to the distance to the rotation point in the dummy dimension (i.e. the distance between the plane and the starting point of the blue vectors in Figure 3.5).

3.2.1 Combining Rotation and Translation

Unlike translation, rotation in two dimensions did not require a dummy dimension. We can add a dummy dimension to the rotation matrix to make it compatible with translation and dummy vectors. The dummy version of the rotation matrix is

$$\mathbf{R}(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The dummy rotation matrix also regenerates the value 1 in the dummy dimension. Using dummy dimensions makes rotation matrices and translation matrices compatible for multiplication. In the next section we will combine rotation and translation to analyze multi-bar linkage arms.

3.3 Multi-bar Linkage Arms

Rotation and translation provide all the tools needed to analyze multi-bar linkage arms like the one shown in Figure 3.1. Pay attention two details in Figure 3.1. First, the segments and angles are numbered starting from the far end of the arm, so the far end of segment 1 is the endpoint of the arm. Second, the angles are defined as counter-clockwise rotations relative to a line that extends from the previous segment. For example, the first angle (θ_1) measures the rotation of segment 1 starting if segment 1 were perfectly in line with segment 2. Bearing these two details in mind, we now show how to calculate the final position of the linkage arm using sequential translations and rotations.

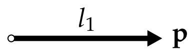
We begin with the end of the arm (point $\mathbf{p}$) at the origin:

$$\mathbf{p} = \mathbf{0}_\mathrm{d}$$

We use a subscript "d" to remind us that $\mathbf{0}_\mathrm{d}$ is not a true zero vector, but rather a dummy vector with the value 1 in the final dimension.

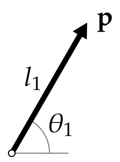
Then we translate $\mathbf{p}$ horizontally by the length of the first arm (l_1).

$$\mathbf{p} = \mathbf{T}(l_1, 0)\mathbf{0}_\mathrm{d}$$



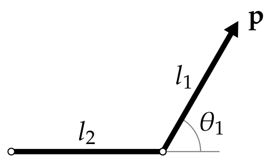
Next we rotate the first segment by the first angle (θ_1).

$$\mathbf{p} = \mathbf{R}(\theta_1)\mathbf{T}(l_1, 0)\mathbf{0}_\mathrm{d}$$



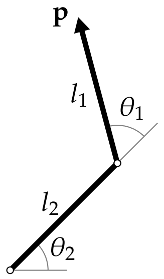
The first arm is translated horizontally by the length of the second arm (l_2).

$$\mathbf{p} = \mathbf{T}(l_2, 0)\mathbf{R}(\theta_1)\mathbf{T}(l_1, 0)\mathbf{0}_\mathrm{d}$$



Finally we rotate both arms by the second angle (θ_2).

$$\mathbf{p} = \mathbf{R}(\theta_2)\mathbf{T}(l_2, 0)\mathbf{R}(\theta_1)\mathbf{T}(l_1, 0)\mathbf{0}_\text{d}$$



Let's do an example where $l_1 = l_2 = 3$ cm, $\theta_1 = 30°$, and $\theta_2 = 45°$. The position of the end of the arm is

$$\begin{aligned}
\mathbf{p} &= \mathbf{R}(45°)\mathbf{T}(3,0)\mathbf{R}(30°)\mathbf{T}(3,0)\mathbf{0}_\text{d} \\
&= \begin{pmatrix} \sqrt{2}/2 & -\sqrt{2}/2 & 0 \\ \sqrt{2}/2 & \sqrt{2}/2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \sqrt{3}/2 & -1/2 & 0 \\ 1/2 & \sqrt{3}/2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} 0.259 & -0.966 & 2.90 \\ 0.966 & 0.259 & 5.02 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} 2.90 \\ 5.02 \\ 1 \end{pmatrix}
\end{aligned}$$

The final position of the arm is 2.90 cm along the horizontal and 5.02 cm along the vertical, as shown in Figure 3.6.

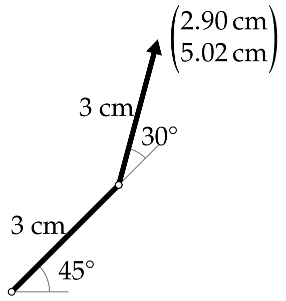

$\begin{pmatrix} 2.90\,\text{cm} \\ 5.02\,\text{cm} \end{pmatrix}$
3 cm
30°
3 cm
45°

Figure 3.6: Example two-bar linkage arm.

3.4 Rotating Shapes

The matrix formalism for rotation and translation naturally extends to shapes, paths, or other collections of points. Consider the triangle shown in Figure 3.7 with vertices at (1, 0), (2, 0), and (2, 1). We can represent the triangle by collecting the vertices in a matrix with one column for each vertex:

$$\mathbf{X} = \begin{pmatrix} 1 & 2 & 2 \\ 0 & 0 & 1 \end{pmatrix}.$$

We can rotate counter-clockwise by 30° with the rotation matrix

$$\mathbf{R}(30°) = \begin{pmatrix} \cos 30° & -\sin 30° \\ \sin 30° & \cos 30° \end{pmatrix}.$$

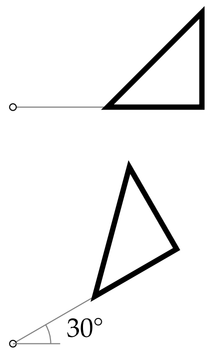


Figure 3.7: Rotating a triangle.

We are not translating the shape—only rotating it—so there is no need for a dummy dimension in this case. The translated shape is the product of the rotation matrix and the matrix of vertices.

$$\begin{aligned}\mathbf{R}(30^\circ)\mathbf{X} &= \begin{pmatrix} \cos 30^\circ & -\sin 30^\circ \\ \sin 30^\circ & \cos 30^\circ \end{pmatrix} \begin{pmatrix} 1 & 2 & 2 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 0.866 & 1.732 & 1.232 \\ 0.500 & 1 & 1.866 \end{pmatrix}\end{aligned}$$

The new vertices are $(0.866, 0.500)$, $(1.732, 1)$, and $(1.232, 1.866)$. The matrix formalism allowed us to rotate all three points simultaneously. Computer graphics software uses rotation and translation matrices to efficiently manipulate shapes by matrix multiplication.

Chapter 4

Solving Linear Systems

Remember back to algebra when you were asked to solve small systems of equations like

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= y_1 \\ a_{21}x_1 + a_{22}x_2 &= y_2 \end{aligned}$$

Your strategy was to manipulate the equations until they reached the form

$$\begin{aligned} x_1 &= y_1' \\ x_2 &= y_2' \end{aligned}$$

We often use the prime symbol ($'$) to indicate that an unspecified new value is based on an old one. For example, y_1' is a new value calculated from y_1. In this case,

$$y_1' = \frac{y_1 a_{22} - a_{12} y_2}{a_{11}a_{22} - a_{12}a_{21}}$$

In matrix form, this process transforms the coefficient matrix **A** into the identity matrix

$$\begin{pmatrix} a_{11} & a_{21} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1' \\ y_2' \end{pmatrix}$$

This leads us to our first strategy for solving linear systems of the form $\mathbf{Ax}= \mathbf{y}$. We manipulate both sides of the equation (**A** and **y**) until **A** becomes the identity matrix. The vector **x** then equals the transformed vector $\mathbf{y}'$. Because we will be applying the same transformations to both **A** and **y**, it is convenient to collect them both into an *augmented matrix* $(\mathbf{A} \quad \mathbf{y})$. For 2×2 system above, the augmented matrix is

$$\begin{pmatrix} a_{11} & a_{12} & y_1 \\ a_{21} & a_{22} & y_2 \end{pmatrix}$$

What operations can we use to transform **A** into the identity matrix? There are three operations, called the *elementary row operations*, or EROs:

1. **Exchanging two rows.** Since the order of the equations in our system is arbitrary, we can re-order the rows of the augmented matrix at will. By working with the augmented matrix we ensure that both the left- and right-hand sides move together.

$$\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{pmatrix} \xrightarrow{R_2 \leftrightarrow R_3} \begin{pmatrix}1 & 2 & 3\\7 & 8 & 9\\4 & 5 & 6\end{pmatrix}$$

2. **Multiplying any row by a scalar.** Again, since we are working with the augmented matrix, multiplying a row by a scalar multiplies both the left- and right-hand sides of the equation by the same factor.

$$\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{pmatrix} \xrightarrow{3R_2} \begin{pmatrix}1 & 2 & 3\\12 & 15 & 18\\7 & 8 & 9\end{pmatrix}$$

3. **Adding a scalar multiple of any row to any other row.** We use the notation $kR_i \longrightarrow R_j$ to denote multiplying row R_i by the scalar k and adding this scaled row to row R_j. For example:

$$\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{pmatrix} \xrightarrow{3R_2 \rightarrow R_1} \begin{pmatrix}13 & 17 & 21\\4 & 5 & 6\\7 & 8 & 9\end{pmatrix}$$

Let's solve the following system of equations using elementary row operations.

$$\begin{aligned}4x_1 + 8x_2 - 12x_3 &= 44\\3x_1 + 6x_2 - 8x_3 &= 32\\-2x_1 - x_2 &= -7\end{aligned}$$

This is a linear system of the form **Ax**= **y** where the coefficient matrix **A** and the vector **y** are

$$\mathbf{A} = \begin{pmatrix}4 & 8 & -12\\3 & 6 & -8\\-2 & -1 & 0\end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix}44\\32\\-7\end{pmatrix}$$

The augmented matrix is therefore

$$\begin{pmatrix}4 & 8 & -12 & 44\\3 & 6 & -8 & 32\\-2 & -1 & 0 & -7\end{pmatrix}$$

Now we apply the elementary row operations.

$$
\xrightarrow{\frac{1}{4}R_1}\begin{pmatrix} 1 & 2 & -3 & 11 \\ 3 & 6 & -8 & 32 \\ -2 & -1 & 0 & -7 \end{pmatrix}
$$
$$
\xrightarrow{-3R_1\to R_2}\begin{pmatrix} 1 & 2 & -3 & 11 \\ 0 & 0 & 1 & -1 \\ -2 & -1 & 0 & -7 \end{pmatrix}
$$
$$
\xrightarrow{2R_1\to R_3}\begin{pmatrix} 1 & 2 & -3 & 11 \\ 0 & 0 & 1 & -1 \\ 0 & 3 & -6 & 15 \end{pmatrix}
$$

Notice that after three steps we have a zero at position (2,2). We need to move this row farther down the matrix to continue; otherwise we can't cancel out the number 3 below it. This operation is called a *pivot*.

$$
\xrightarrow{R_2\leftrightarrow R_3}\begin{pmatrix} 1 & 2 & -3 & 11 \\ 0 & 3 & -6 & 15 \\ 0 & 0 & 1 & -1 \end{pmatrix}
$$
$$
\xrightarrow{\frac{1}{3}R_2}\begin{pmatrix} 1 & 2 & -3 & 11 \\ 0 & 1 & -2 & 5 \\ 0 & 0 & 1 & -1 \end{pmatrix}
$$

At this point we have a matrix in *row echelon form*. The bottom triangle looks like the identity matrix. We could stop here and solve the system using back substitution:

$$
\begin{aligned}
x_3 &= -1 \\
x_2 + -2(-1) = 5 \Rightarrow\ x_2 &= 3 \\
x_1 + 2(3) - 3(-1) = 11 \Rightarrow\ x_1 &= 2
\end{aligned}
$$

Or, we could keep going and place the augmented matrix into *reduced row echelon* *form*.

$$
\xrightarrow{-2R_2\to R_1}\begin{pmatrix} 1 & 0 & 1 & 1\\ 0 & 1 & -2 & 5\\ 0 & 0 & 1 & -1 \end{pmatrix}
$$
$$
\xrightarrow{-R_3\to R_1}\begin{pmatrix} 1 & 0 & 0 & 2\\ 0 & 1 & -2 & 5\\ 0 & 0 & 1 & -1 \end{pmatrix}
$$
$$
\xrightarrow{2R_3\to R_2}\begin{pmatrix} 1 & 0 & 0 & 2\\ 0 & 1 & 0 & 3\\ 0 & 0 & 1 & -1 \end{pmatrix}
$$

The left three columns are the identity matrix, so the resulting system of equations has been simplified to

$$
\begin{aligned}
x_1 &= 2\\
x_2 &= 3\\
x_3 &= -1
\end{aligned}
$$

4.1 Gaussian Elimination

Using EROs to transform the augmented matrix into the identity matrix is called *Gaussian elimination*. Let's develop an algorithm for Gaussian elimination for a general system of equations $\mathbf{Ax} = \mathbf{y}$ when $\mathbf{A}$ is an $n \times n$ coefficient matrix. We begin with the augmented matrix

$$
\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} & y_1\\ a_{21} & a_{22} & \cdots & a_{2n} & y_2\\ \vdots & & \ddots & & \vdots\\ a_{n1} & a_{n2} & \cdots & a_{nn} & y_n \end{pmatrix}
$$

We need the number 1 in the a_{11} position.

$$
\xrightarrow{a_{11}^{-1}R_1}\begin{pmatrix} 1 & a_{11}^{-1}a_{12} & \cdots & a_{11}^{-1}a_{1n} & a_{11}^{-1}y_1\\ a_{21} & a_{22} & \cdots & a_{2n} & y_2\\ \vdots & & \ddots & & \vdots\\ a_{n1} & a_{n2} & \cdots & a_{nn} & y_n \end{pmatrix}
$$

Now we zero out the a_{21} position using the first row multiplied by $-a_{21}$.

$$\xrightarrow{-a_{21}R_1 \to R_2} \begin{pmatrix} 1 & a_{11}^{-1}a_{12} & \cdots & a_{11}^{-1}a_{1n} & a_{11}^{-1}y_1 \\ 0 & a_{22} - a_{21}a_{11}^{-1}a_{12} & \cdots & a_{2n} - a_{21}a_{11}^{-1}a_{1n} & y_2 - a_{21}a_{11}^{-1}y_1 \\ \vdots & & \ddots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & y_n \end{pmatrix}$$

We keep zeroing out the entries a_{31} through a_{n1} using the first row. We end up with the matrix

$$\xrightarrow{-a_{n1}R_1 \to R_n} \begin{pmatrix} 1 & a_{11}^{-1}a_{12} & \cdots & a_{11}^{-1}a_{1n} & a_{11}^{-1}y_1 \\ 0 & a_{22} - a_{21}a_{11}^{-1}a_{12} & \cdots & a_{2n} - a_{21}a_{11}^{-1}a_{1n} & y_2 - a_{21}a_{11}^{-1}y_1 \\ \vdots & & \ddots & & \vdots \\ 0 & a_{n2} - a_{n1}a_{11}^{-1}a_{12} & \cdots & a_{nn} - a_{n1}a_{11}^{-1}a_{1n} & y_n - a_{n1}a_{11}^{-1}y_1 \end{pmatrix}$$

This is looking a little complicated, so let's rewrite the matrix as

$$\begin{pmatrix} 1 & a'_{12} & \cdots & a'_{1n} & y'_1 \\ 0 & a'_{22} & \cdots & a'_{2n} & y'_2 \\ \vdots & & \ddots & & \vdots \\ 0 & a'_{n2} & \cdots & a'_{nn} & y'_n \end{pmatrix}$$

The first column looks like the identity matrix, which is exactly what we want. Our next goal is to put the lower half of an identify matrix in the second column by setting $a'_{22} = 1$ and $a_{32}, \ldots, a_{n2} = 0$. Notice that this is the same as applying the above procedure to the sub-matrix

$$\begin{pmatrix} a'_{22} & \cdots & a'_{2n} & y'_2 \\ \vdots & \ddots & & \vdots \\ a'_{n2} & \cdots & a'_{nn} & y'_n \end{pmatrix}$$

After that, we can continue recursively until the left part of the augmented matrix is the identity matrix. We can formalize the Gaussian elimination algorithm as follows:

```
function GAUSSIAN ELIMINATION
    for $j = 1$ to $n$ do                              ▷ For every column
        $a_{jj}^{-1} R_j$                               ▷ Set the element on the diagonal to 1
        for $i = j + 1$ to $n$ do                      ▷ For every row below the diagonal
            $-a_{ij} R_j \longrightarrow R_i$          ▷ Zero the below-diagonal element
        end for
    end for
end function
```

We're ignoring pivoting here. In general, we need to check that $a_{jj} \neq 0$; if it is, we swap the row for one below that has a nonzero term in the jth column.

4.2 Computational Complexity of Gaussian Elimination

How many computational operations are needed to perform Gaussian elimination on an $n \times n$ matrix? Let's start by counting operations when reducing the first column. Scaling the first row ($a_{11}^{-1} R_1$) requires n operations. (There are $n+1$ entries in the first row of the augmented matrix if you include the value y_1; however, we know the result in the first column, $a_{11}^{-1} a_{11}$, will always equal the number 1; we don't need to compute it.) Similarly, zeroing out a single row below requires n multiplications and n subtractions. (Again, there are $n + 1$ columns, but we know the result in column 1 will be zero.) In the first column, there are $n - 1$ rows below to zero out, so the total number of operations is

$$\underbrace{n}_{a_{11}^{-1} R_1} + \underbrace{2(n-1)n}_{-a_{i1} R_1 \to R_i} = 2n^2 - n$$

After we zero the bottom of the first row, we repeat the procedure on the $(n-1) \times (n-1)$ submatrix, and so on until we reach the 1×1 submatrix that includes only a_{nn}. We add up the number of operations for each of these n submatrices

Remember that

$$\sum_{k=1}^{n} 1 = n$$

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\begin{aligned}
\text{total operations} &= \sum_{k=1}^{n} (2k^2 - k) \\
&= 2\sum_{k=1}^{n} k^2 - \sum_{k=1}^{n} k \\
&= \frac{n(n+1)(2n+1)}{3} - \frac{n(n+1)}{2} \\
&= \mathcal{O}(n^3)
\end{aligned}$$

Thus the number of operations required to bring an $n \times n$ matrix into row-echelon form is on the order of n^3. The number of operations needed to perform back substitution and solve the system is $O(n^2)$. This raises two important points.

1. Gaussian elimination scales cubically. A system with twice as many equations takes eight times longer to solve.

2. The computational bottleneck is generating the row echelon matrix. Back substitution (or creating the reduced row echelon matrix) is significantly faster.

4.3 Solving Linear Systems in Matlab

Matlab has multiple functions for solving linear systems. Given variables `A` and `y`, you can use

- `linsolve(A,y)` to solve using LU decomposition, a variant of Gaussian elimination.

- `(A \ y)` to let Matlab choose the best algorithm based on the size and structure of `A`.

- `rref([A y])` to compute the reduced row echelon form of the augmented matrix.

Chapter 5

The Finite Difference Method

When you first learned about derivatives, they were probably introduced as the limit of a finite difference

$$\frac{du}{dx} = \lim_{a \to b} \frac{u(b) - u(a)}{b - a}$$

As the distance between points a and b shrinks to zero, the finite difference becomes infinitesimal. The resulting differential equations must be solved with integral calculus. This chapter presents an alternative, numerical method for solving differential equations. Rather than shrink the finite differences all the way to zero, we leave a small but finite gap. The resulting algebraic equations approximate the differential equation and can be solved without any tools from calculus.

5.1 Finite Differences

Solving a differential equation analytically yields a solution over the entire domain (the region between the boundary conditions). By contrast, numerical methods find solutions to a differential equation at only a discrete set of points, or *nodes*, in the domain. The derivatives in the differential equation are descretized by converting them into *finite differences*. For example, we can approximate a first derivative as the change between two nodes divided by the distance between the nodes:

$$\frac{du}{dx} \approx \frac{u^{(k+1)} - u^{(k)}}{\Delta x}$$

where $u^{(k)}$ is the value of the variable at the kth node. To approximate a second derivative, we calculate the finite difference between nodes $u^{(k+1)}$ and $u^{(k)}$; and

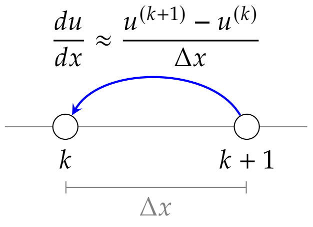

$\frac{du}{dx} \approx \frac{u^{(k+1)} - u^{(k)}}{\Delta x}$
k
$k+1$
Δx

Figure 5.1: First-order finite difference approximation.

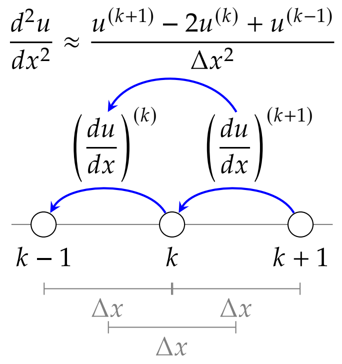
We write the value of u at node k as $u^{(k)}$. Using a superscript with parentheses avoids confusion with expressions u^k (the kth power of u) and u_k (the kth element of a vector named $\mathbf{u}$).

nodes $u^{(k)}$ and $u^{(k-1)}$. We divide this "difference between differences" by the distance between the centers of the nodes:

$$\frac{d^2u}{dx^2} \approx \frac{\left(\frac{du}{dx}\right)^{(k+1)} - \left(\frac{du}{dx}\right)^{(k)}}{\Delta x}$$

$$= \frac{\frac{u^{(k+1)} - u^{(k)}}{\Delta x} - \frac{u^{(k)} - u^{(k-1)}}{\Delta x}}{\Delta x}$$

$$= \frac{u^{(k+1)} - 2u^{(k)} + u^{(k-1)}}{\Delta x^2}$$



$\frac{d^2u}{dx^2} \approx \frac{u^{(k+1)} - 2u^{(k)} + u^{(k-1)}}{\Delta x^2}$
$\left(\frac{du}{dx}\right)^{(k)}$
$\left(\frac{du}{dx}\right)^{(k+1)}$
$k-1$
k
$k+1$
Δx
Δx
Δx

Figure 5.2: The second-order finite difference approximation is computed using two adjacent first-order approximations.

5.2 Linear Differential Equations

In order to generate a set of linear algebraic equations, the starting ODE or PDE must be linear. Linearity for differential equations means that the dependent variable (i.e. u) only appears in linear expressions; there can be nonlinearities involving only the independent variables. Remember that differentiation is a linear operator, so all derivatives of u are linear.

The finite difference method will still work on nonlinear PDEs; however, the resulting set of equations are nonlinear.

For example, consider the PDE with dependent variable $u(t,x)$:

$$t^2\frac{\partial u}{\partial t} = c_1\frac{\partial^2 u}{\partial x^2} + c_2\sin(x)\frac{\partial u}{\partial x} + c_3e^{tx}$$

This PDE is linear in u. However, the ODE

$$u\frac{du}{dx} = 0$$

is not linear. In general, for a variable $u(t,x)$, any terms of the form

$$f(t,x)\frac{\partial^n u}{\partial t^n} \quad \text{or} \quad f(t,x)\frac{\partial^n u}{\partial x^n}$$

are linear.

As a rule of thumb, you can tell if a PDE is linear in u by ignoring the derivative operators and seeing if the resulting algebraic equation is linear in u.

You might be wondering why we only require that a PDE be linear in the dependent variable. Why do nonlinearities in the independent variables not lead to nonlinear algebraic equations? Remember that in the finite difference method we discretize the independent variables across a grid and solve for the dependent variable at each node. Before solving, the value of the dependent variable is unknown. However, the value of all the independent variables are known, i.e. we know the location of every node in space and time. We can evaluate all terms involving the independent variables when setting up our equations.

5.3 Discretizing a Linear Differential Equation

Converting a linear differential equation into a set of linear algebraic equations requires three steps:

1. Divide the domain into n equally-sized intervals. Creating n intervals requires $n+1$ points, or *nodes*, labeled 0, 1, …, n. The spacing between each node is $\Delta x = l/n$ where l is the length of the domain.

2. Starting with the interior nodes $(1, 2, \ldots, n-1)$, we rewrite the differential equation at each node using finite differences.

$$\begin{aligned} u &\longrightarrow u^{(k)} \\ \frac{du}{dx} &\longrightarrow \frac{u^{(k+1)} - u^{(k)}}{\Delta x} \\ \frac{d^2u}{dx^2} &\longrightarrow \frac{u^{(k+1)} - 2u^{(k)} + u^{(k-1)}}{\Delta x^2} \end{aligned}$$

3. Add equations to enforce the boundary conditions at the boundary nodes.

For example, consider the ordinary differential equation

$$\frac{d^2u}{dx^2} + \frac{du}{dx} - 6u = 0, \quad u(0) = 0, \quad u(1) = 3$$

If we divide the domain $[0,1]$ into four sections, then $n = 4$ and $\Delta x = l/n = 1/4 = 0.25$. We have five nodes (0, 1, 2, 3, 4), three of which are interior nodes (1, 2, 3). We rewrite the ODE using finite differences at each of the interior nodes.

$$\frac{u^{(2)} - 2u^{(1)} + u^{(0)}}{(0.25)^2} + \frac{u^{(2)} - u^{(1)}}{0.25} - 6u^{(1)} = 0 \quad \text{[node 1]}$$
$$\frac{u^{(3)} - 2u^{(2)} + u^{(1)}}{(0.25)^2} + \frac{u^{(3)} - u^{(2)}}{0.25} - 6u^{(2)} = 0 \quad \text{[node 2]}$$
$$\frac{u^{(4)} - 2u^{(3)} + u^{(2)}}{(0.25)^2} + \frac{u^{(4)} - u^{(3)}}{0.25} - 6u^{(3)} = 0 \quad \text{[node 3]}$$

which simplifies to the equations

$$\begin{aligned} 16u^{(0)} - 42u^{(1)} + 20u^{(2)} &= 0 \\ 16u^{(1)} - 42u^{(2)} + 20u^{(3)} &= 0 \\ 16u^{(2)} - 42u^{(3)} + 20u^{(4)} &= 0 \end{aligned}$$

Now we can add equations for the boundary nodes. Node 0 corresponds to $x = 0$, so the boundary condition tells us that $u^{(0)} = 0$. Similarly, node 4 corresponds to $x = 1$, so $u^{(4)} = 3$. Combining these two boundary equations with the equations for the interior nodes yields the final linear system

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 16 & -42 & 20 & 0 & 0 \\ 0 & 16 & -42 & 20 & 0 \\ 0 & 0 & 16 & -42 & 20 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u^{(0)} \\ u^{(1)} \\ u^{(2)} \\ u^{(3)} \\ u^{(4)} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 3 \end{pmatrix}$$

Solving by Gaussian elimination yields

$$\begin{pmatrix} u^{(0)} \\ u^{(1)} \\ u^{(2)} \\ u^{(3)} \\ u^{(4)} \end{pmatrix} = \begin{pmatrix} 0.00 \\ 0.51 \\ 1.07 \\ 1.84 \\ 3.00 \end{pmatrix}$$

We can compare our numerical solution to the exact solution for this expression, which is

$$u(x) = \frac{3}{e^2 - e^{-3}} \left(e^{2x} - e^{-3x} \right)$$

x	Numerical Solution	Exact Solution	Error
0	0.00	0.00	0.0%
0.25	0.51	0.48	5.7%
0.50	1.07	1.02	4.7%
0.75	1.84	1.79	2.6%
1	3.00	3.00	0.0%

We used only five nodes to solve this ODE, yet our relative error is less than 6%!

5.4 Boundary Conditions

There are two ways to write a finite difference approximation for the first derivative at node k. We can write the *forward difference* using node $k + 1$:

$$\frac{du}{dx} \to \frac{u^{(k+1)} - u^{(k)}}{\Delta x}$$

or we can use the *backward difference* using the previous node at $k - 1$:

$$\frac{du}{dx} \to \frac{u^{(k)} - u^{(k-1)}}{\Delta x}$$

The choice of forward or backward differences depends on the boundary conditions (Figure 5.3). For a first order ODE, we are given a boundary condition at either $x = 0$ or $x = l$. If we are given $u(0)$, we use backward differences. If we are given $u(l)$, we use forward differences. Otherwise, we run out of nodes when writing equations for the finite differences.

Second order finite differences require nodes on both sides. This is related to the necessity of two boundary conditions, since we cannot write finite difference approximations for nodes at either end. If your ODE comes with two boundary conditions, you can choose either forward or backward differences to approximate any first order derivatives.

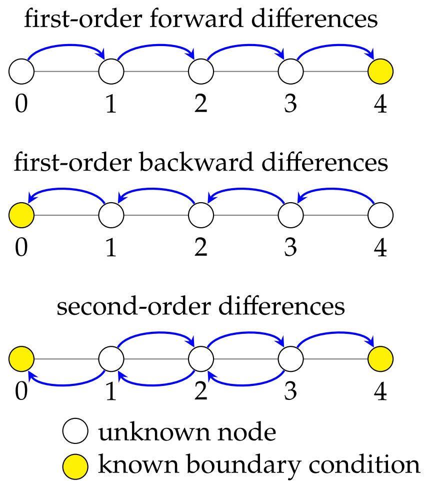


Figure 5.3: First-order equations use either forward or backward differences depending on the location of the boundary condition. Second-order equations use both forward and backward differences and require two boundary conditions.

You cannot avoid the issue by using the difference $u^{(0)} - u^{(1)}$ for node 0 and $u^{(1)} - u^{(0)}$ for node 1. These equations are *linearly dependent*, which leaves us with too little information when solving the system.

Chapter 6

Matrix Inverse

6.1 Defining the Matrix Inverse

So far we've demonstrated how Gaussian elimination can solve linear systems of the form $\mathbf{Ax} = \mathbf{y}$. Gaussian elimination involves a series of elementary row operations to transform the coefficient matrix $\mathbf{A}$ into the identity matrix. While Gaussian elimination works well, our initial goal of defining an algebra for vectors requires something stronger – a multiplicative inverse. For vectors, the multiplicative inverse is called a *matrix inverse*. For any square matrix, a matrix inverse (if it exists) is a square matrix $\mathbf{A}^{-1}$ such that

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I} = \mathbf{A}\mathbf{A}^{-1}$$

This definition is analogous to the field axiom that there exists a^{-1} such that $a^{-1}a = 1$ for all nonzero a. Since scalar multiplication always commutes, $a^{-1}a = aa^{-1}$. Matrix multiplication does not commute, so we need to state this property separately.

If we could prove the existence of a *matrix inverse* for $\mathbf{A}$, we could solve a wide variety of linear systems, including $\mathbf{Ax}= \mathbf{y}$.

$$\begin{aligned}
\mathbf{Ax} &= \mathbf{y} \\
\mathbf{A}^{-1}\mathbf{Ax} &= \mathbf{A}^{-1}\mathbf{y} \\
\mathbf{I}\,\mathbf{x} &= \mathbf{A}^{-1}\mathbf{y} \\
\mathbf{x} &= \mathbf{A}^{-1}\mathbf{y}
\end{aligned}$$

Being amenable to Gaussian elimination is related to the existence of the matrix inverse. While the end result is the same (a transformation of the coefficient matrix into the identity matrix), the processes are different. The Gaussian elimination algorithm applies a series of elementary row operations while pivoting to avoid numerical issues. A matrix inverse (if it exists) must capture all of these operations in a single matrix multiplication. This condensing of Gaussian elimination is not a trivial task.

Our first goal for this chapter is to prove the existence of the matrix inverse for any coefficient matrix that can be solved by Gaussian elimination. Then we will derive a method to construct the matrix inverse if it exists. The following chapter formalizes the requirements for solvability of a linear system of equations, which is related to the existence of the matrix inverse.

6.2 Elementary Matrices

Before proving the existence of the matrix inverse, we need to add another matrix manipulation tool to our arsenal — the *elementary matrix*. An elementary matrix is constructed by applying any single elementary row operation to the identity matrix. For example, consider swapping the second and third rows of the 3×3 identity matrix:

We use the notation $\mathbf{E}_r$ to denote an elementary matrix constructed using the elementary row operation r.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \xrightarrow{R_2 \leftrightarrow R_3} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \mathbf{E}_{R_2 \leftrightarrow R_3}$$

Notice what happens when we left multiply a matrix with an elementary matrix.

$$\mathbf{E}_{R_2 \leftrightarrow R_3} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \\ 4 & 5 & 6 \end{pmatrix}$$

Multiplication by the elementary matrix exchanges the second and third rows — the same operation that created the elementary matrix. The same idea works for other elementary row operations, such as scalar multiplication

Multiplication by an elementary matrix on the right applies the same operation to the columns of the matrix.

$$\mathbf{E}_{3R_2} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{E}_{3R_2} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 12 & 15 & 18 \\ 7 & 8 & 9 \end{pmatrix}$$

and addition by a scaled row

$$\mathbf{E}_{2R_3 \to R_2} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{E}_{2R_3\to R_2}\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}=\begin{pmatrix}1&0&0\\0&1&2\\0&0&1\end{pmatrix}\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}=\begin{pmatrix}1&2&3\\18&21&24\\7&8&9\end{pmatrix}$$

6.3 Proof of Existence for the Matrix Inverse

We are now ready to prove the existence of the inverse for any coefficient matrix that is solvable by Gaussian elimination, i.e. any square matrix that can be transformed into the identity matrix with elementary row operations. We will prove existence in three steps:

1. Construct a matrix $\mathbf{P}$ that looks like a left inverse ($\mathbf{PA} = \mathbf{I}$).
2. Show that this left inverse is also a right inverse ($\mathbf{AP} = \mathbf{I}$).
3. Show that the matrix inverse is unique, implying that $\mathbf{P}$ must be the inverse of $\mathbf{A}$.

Theorem. *Suppose matrix* $\mathbf{A}$ *can be reduced to the identity matrix* $\mathbf{I}$ *by elementary row operations. Then there exists a matrix* $\mathbf{P}$ *such that* $\mathbf{PA} = \mathbf{I}$.

Proof. We assume that reducing $\mathbf{A}$ to $\mathbf{I}$ requires k elementary row operations. Let $\mathbf{E}_1, \dots, \mathbf{E}_k$ be the associated elementary matrices. Then

$$\begin{aligned}\mathbf{E}_k\mathbf{E}_{k-1}\cdots\mathbf{E}_2\mathbf{E}_1\mathbf{A} &= \mathbf{I}\\ (\mathbf{E}_k\mathbf{E}_{k-1}\cdots\mathbf{E}_2\mathbf{E}_1)\mathbf{A} &= \mathbf{I}\\ \mathbf{PA} &= \mathbf{I}\end{aligned}$$

where $\mathbf{P} = \mathbf{E}_k\mathbf{E}_{k-1}\cdots\mathbf{E}_2\mathbf{E}_1$. □

Theorem. *If* $\mathbf{PA} = \mathbf{I}$ *then* $\mathbf{AP} = \mathbf{I}$.

Proof.

$$\begin{aligned}\mathbf{PA} &= \mathbf{I}\\ \mathbf{PAP} &= \mathbf{IP}\\ \mathbf{P}(\mathbf{AP}) &= \mathbf{P}\end{aligned}$$

Since $\mathbf{P}$ multiplied by $\mathbf{AP}$ gives $\mathbf{P}$ back again, $\mathbf{AP}$ must equal the identity matrix ($\mathbf{AP} = \mathbf{I}$). □

Theorem. *The inverse of a matrix is unqiue.*

Proof. Let $\mathbf{A}^{-1}$ be the inverse of $\mathbf{A}$, i.e. $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I} = \mathbf{A}\mathbf{A}^{-1}$. Suppose there exists another matrix $\mathbf{P} \neq \mathbf{A}^{-1}$ such that $\mathbf{PA} = \mathbf{I} = \mathbf{AP}$.

$$\begin{aligned} \mathbf{PA} &= \mathbf{I} \\ \mathbf{PAA}^{-1} &= \mathbf{IA}^{-1} \\ \mathbf{PI} &= \mathbf{A}^{-1} \\ \mathbf{P} &= \mathbf{A}^{-1} \end{aligned}$$

This contradicts our supposition that $\mathbf{P} \neq \mathbf{A}^{-1}$, so $\mathbf{A}^{-1}$ must be unique. □

6.4 Computing the Matrix Inverse

Our proof of existence for the matrix inverse provided a method to construct the inverse. We performed Gaussian elimination on a matrix and built an elementary matrix for each step. These elementary matrices were multiplied together to form the matrix inverse. In practice this method would be wildly inefficient. Transforming an $n \times n$ matrix to reduced row echelon form requires $\mathcal{O}(n^2)$ elementary row operations, so we would need to construct and multiply $\mathcal{O}(n^2)$ elementary matrices. Since naive matrix multiplication requires $\mathcal{O}(n^3)$ operations per matrix, constructing a matrix inverse with this method requires $\mathcal{O}(n^5)$ operations! Gaussian elimination is $\mathcal{O}(n^3)$, so we would be far better off avoiding the matrix inverse entirely.

Gaussian elimination requires $\mathcal{O}(n^2)$ row operations but $\mathcal{O}(n^3)$ total operations. Each row operation requires $\mathcal{O}(n)$ individual operations — one for each column.

The fastest matrix multiplication algorithm is $\mathcal{O}(n^{2.7373})$, although this is not a big help in practice.

Fortunately, there are better methods for constructing matrix inverses. One of the best is called the *side-by-side method*. To see how the side-by-side method works, let's construct an inverse for the square matrix $\mathbf{A}$. We can use Gaussian elimination to transform $\mathbf{A}$ into the identity matrix, which we can represent with a series of k elementary matrices.

$$\underbrace{\mathbf{E}_k\mathbf{E}_{k-1}\cdots\mathbf{E}_2\mathbf{E}_1}_{\mathbf{A}^{-1}}\mathbf{A} = \mathbf{I}$$

What would happen if we simultaneously apply the same elementary row operations to another identity matrix?

$$\underbrace{\mathbf{E}_k\mathbf{E}_{k-1}\cdots\mathbf{E}_2\mathbf{E}_1}_{\mathbf{A}^{-1}}\mathbf{I} = \mathbf{A}^{-1}\mathbf{I} = \mathbf{A}^{-1}$$

In the side-by-side method, we start with an augmented matrix containing the $n \times n$ matrix $\mathbf{A}$ and an $n \times n$ identity matrix. Then we apply Gaussian elimination to transform $\mathbf{A}$ into $\mathbf{I}$. The augmented matrix ensures that the same elementary row operations will be applied to the identity matrix, yielding the inverse of $\mathbf{A}$, or

$$\begin{pmatrix}\mathbf{A} & \mathbf{I}\end{pmatrix} \xrightarrow{\text{EROs}} \begin{pmatrix}\mathbf{I} & \mathbf{A}^{-1}\end{pmatrix}.$$

Like the augmented matrix $(\mathbf{A}\,\mathbf{y})$, there is no direct interpretation of $(\mathbf{A}\,\mathbf{I})$. It is simply a convenient way to apply the same EROs to both $\mathbf{A}$ and $\mathbf{I}$.

Let's solve the following system by constructing the matrix inverse.

$$\begin{aligned} 3x_1 + 2x_2 &= 7 \\ x_1 + x_2 &= 4 \end{aligned}$$

In matrix form,

$$\mathbf{A} = \begin{pmatrix} 3 & 2 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} 7 \\ 4 \end{pmatrix}$$

We start with the augmented matrix for the side-by-side method.

$$\begin{aligned}
\begin{pmatrix} 3 & 2 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} &\xrightarrow{R_1 \leftrightarrow R_2} \begin{pmatrix} 1 & 1 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{pmatrix} \\
&\xrightarrow{-3R_1 \to R_2} \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & -1 & 1 & -3 \end{pmatrix} \\
&\xrightarrow{-R_2} \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & -1 & 3 \end{pmatrix} \\
&\xrightarrow{-R_2 \to R_1} \begin{pmatrix} 1 & 0 & 1 & -2 \\ 0 & 1 & -1 & 3 \end{pmatrix}
\end{aligned}$$

Thus, the matrix inverse is

$$\mathbf{A}^{-1} = \begin{pmatrix} 1 & -2 \\ -1 & 3 \end{pmatrix}$$

which we can verify by multiplication on the left and right.

$$\begin{aligned}
\mathbf{A}^{-1}\mathbf{A} &= \begin{pmatrix} 1 & -2 \\ -1 & 3 \end{pmatrix}\begin{pmatrix} 3 & 2 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \mathbf{I} \\
\mathbf{A}\mathbf{A}^{-1} &= \begin{pmatrix} 3 & 2 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & -2 \\ -1 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \mathbf{I}
\end{aligned}$$

Finally, we can compute the solution by matrix multiplication.

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{y} = \begin{pmatrix} 1 & -2 \\ -1 & 3 \end{pmatrix}\begin{pmatrix} 7 \\ 4 \end{pmatrix} = \begin{pmatrix} -1 \\ 5 \end{pmatrix}$$

The advantage of having a matrix inverse is that if only the right hand side of a linear system changes, we do not need to retransform the coefficient matrix. For example, to solve

$$\begin{aligned} 3x_1 + 2x_2 &= 1 \\ x_1 + x_2 &= 3 \end{aligned}$$

we can re-use $\mathbf{A}^{-1}$ since $\mathbf{A}$ is unchanged (only $\mathbf{y}$ is different).

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{y} = \begin{pmatrix} 1 & -2 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 3 \end{pmatrix} = \begin{pmatrix} -5 \\ 8 \end{pmatrix}$$

6.5 Numerical Issues

Matrix inversion is a powerful method for solving linear systems. However, calculating a matrix inverse should always be your last resort. There are far more efficient and numerically stable methods for solving linear systems. Reasons against using a matrix inverse include:

1. **Computation time.** The side-by-side method is more efficient than multiplying elementary matrices for constructing matrix inverses. Regardless, both methods are much slower than Gaussian elimination for solving linear systems of the form $\mathbf{Ax} = \mathbf{y}$. Both the side-by-side method and Gaussian elimination reduce an augmented matrix. For an $n \times n$ matrix $\mathbf{A}$, the augmented matrix for Gaussian elimination $(\mathbf{A}\,\mathbf{y})$ is $n \times (n+1)$. The augmented matrix for the side-by-side method $(\mathbf{A}\,\mathbf{I})$ is $n \times 2n$. Solving for the inverse requires nearly twice the computation as solving the linear system directly. Having the inverse allows us to "resolve" the system with a new right hand side vector for only the cost of a matrix multiplication. However, there are variants of Gaussian elimination – such as *LU* decomposition – that allow resolving without repeating the entire reduction of the coefficient matrix $\mathbf{A}$.

2. **Memory.** Most large matrices in engineering are *sparse*. Sparse matrices contain very few nonzero entries; matrices with less than 0.01% nonzero entries are not uncommon. Examples of sparse matrices include matrices generated from finite difference approximations or matrices showing connections between nodes in large networks. Computers store sparse matrices by only storing the nonzero entries and their locations. However, there is no guarantee that the inverse of a sparse matrix will also be sparse. Consider the arrow

Imagine the connection matrix for Facebook. It would have hundreds of millions of rows and columns, but each person (row) would only have nonzero entries for a few hundred people (columns) that they knew.

matrix, a matrix with ones along the diagonal and last column and row.

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

An $n \times n$ arrow matrix has n^2 entries but only $3n - 2$ nonzeros. However, the inverse of an arrow matrix always has 100% nonzeros. For example, the inverse of the 8×8 matrix above is

A 1000×1000 arrow matrix has less than 0.3% nonzeros.

$$\mathbf{A}^{-1} = \frac{1}{6} \begin{pmatrix} 5 & -1 & -1 & -1 & -1 & -1 & -1 & 1 \\ -1 & 5 & -1 & -1 & -1 & -1 & -1 & 1 \\ -1 & -1 & 5 & -1 & -1 & -1 & -1 & 1 \\ -1 & -1 & -1 & 5 & -1 & -1 & -1 & 1 \\ -1 & -1 & -1 & -1 & 5 & -1 & -1 & 1 \\ -1 & -1 & -1 & -1 & -1 & 5 & -1 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & 5 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 \end{pmatrix}$$

Calculating the inverse of a large, sparse matrix could requires orders of magnitude more memory than the original matrix. Storing – much less computing – the inverse is impossible for some matrices.

Despite its disadvantages, the matrix inverse is still a powerful tool. The matrix inverse is necessary for many algebraic manipulations, and the inverse can be used to simply or prove many matrix equations. Just remember to think critically about the need for a matrix inverse before calculating one.

6.6 Inverses of Elementary Matrices

We conclude with another interesting property of elementary matrices. We said before that left multiplication by an elementary matrix performs an elementary row operation (the same ERO that was used to construct the elementary matrix). Left multiplication by the inverse of an elementary matrix "undoes" the operation of the elementary matrix. For example, the elementary matrix $\mathbf{E}_{3R_2}$ scales the second row by two. The inverse $\mathbf{E}_{3R_2}^{-1}$ would scale the second row by $1/2$, undoing the scaling by two. Similarly, $\mathbf{E}_{R_2 \leftrightarrow R_3}$ swaps rows two and three, and $\mathbf{E}^{-1}_{R_2 \leftrightarrow R_3}$ swaps them back. The proof of this property is straightforward.

Theorem. *If the elementary matrix $\mathbf{E}_r$ performs the elementary row operation r, then left multiplication by the inverse $\mathbf{E}_r^{-1}$ undoes this operation.*

Proof.

$$\mathbf{E}_r^{-1}(\mathbf{E}_r \mathbf{A}) = (\mathbf{E}_r^{-1}\mathbf{E}_r)\mathbf{A} = (\mathbf{I})\mathbf{A} = \mathbf{A}$$

□

Chapter 7

Rank and Solvability

7.1 Rank of a Matrix

Consider the linear system

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 2 & -4 \\ -2 & 0 & -6 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -2 \\ -4 \end{pmatrix}$$

and the row echelon form of the associated augmented matrix

$$\begin{pmatrix} 1 & 0 & 3 & 2 \\ 0 & 2 & -4 & -2 \\ -2 & 0 & -6 & -4 \end{pmatrix} \xrightarrow{\frac{1}{2}R_2} \xrightarrow{2R_1 \to R_3} \begin{pmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & -2 & -1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Notice that the last row is all zeros. We have no information about the last entry (x_3). However, this does not mean we cannot solve the linear system. Since x_3 is unknown, let us assign its value the symbol α. Then, by back substitution

$$\begin{aligned} & & x_3 &= \alpha \\ x_2 - 2x_3 = -1 &\Rightarrow & x_2 &= 2\alpha - 1 \\ x_1 + 3x_3 = 2 &\Rightarrow & x_1 &= 2 - 3\alpha \end{aligned}$$

The above linear system has not one solution, but infinitely many. There is a solution for every value of the parameter α, so we say the system has a *parameterized solution*.

Parameterized solutions are necessary any time row echelon reduction of a matrix leads to one or more rows with all zero entries. The number of nonzero rows in the row echelon form of a matrix is the matrix's *rank*. The rank of a matrix can be calculated by counting the number of nonzero rows after the matrix is transformed into row echelon form by Gaussian elimination. In general, if a matrix with n columns has rank n, it is possible to find a unique solution to the system $\mathbf{Ax} = \mathbf{y}$. If $\text{rank}(\mathbf{A}) < n$, there may be infinitely many solutions. These solutions require that we specify $n - \text{rank}(\mathbf{A})$ parameters.

We denote the rank of a matrix $\mathbf{A}$ as $\text{rank}(\mathbf{A})$.

Matrices have both a *row rank* (the number of nonzero rows in row-reduced echelon form) and a *column rank* (the number of nonzero columns in a column-reduced echelon form). Thus the concept of rank also applies to nonsquare matrices. However, the row and column ranks are always equivalent, even if the matrix is not square.

Theorem. *The row rank of a matrix equals the column rank of the matrix, i.e.* $\text{rank}(\mathbf{A}) = \text{rank}(\mathbf{A}^\mathsf{T})$.

Proof. We will defer the proof of this theorem until after we learn about matrix decompositions in Part III. □

The equivalence of the row and column ranks implies an upper bound on the rank of nonsquare matrices.

Theorem. *The rank of a matrix is less than or equal to the smallest dimension of the matrix, i.e.* $\text{rank}(\mathbf{A}) \le \min(\dim \mathbf{A})$.

Proof. The row rank of $\mathbf{A}$ is the number of nonzero rows in the row-reduced $\mathbf{A}$, so the rank of $\mathbf{A}$ must be less than the number of rows in $\mathbf{A}$. Since the row rank is also equal to the column rank, there must also be $\text{rank}(\mathbf{A})$ nonzero columns in the column-reduced $\mathbf{A}$. So the rank of $\mathbf{A}$ must never be larger than either the number of rows or number of columns in $\mathbf{A}$. □

We say that a matrix is *full rank* if it has the maximum possible rank (rank n for an $n \times n$ square matrix or rank $\min(m, n)$ for an $m \times n$ rectangular matrix). A matrix that is not full rank is *rank deficient*.

7.1.1 Properties of Rank

The rank of a matrix is connected with many other matrix operations. Here are a few properties of matrices and their ranks. For any $m \times n$ matrix $\mathbf{A}$:

1. $\text{rank}(\mathbf{A}) \le \min(m, n)$.
2. $\text{rank}(\mathbf{A}) = \text{rank}(\mathbf{A}^\mathsf{T})$, i.e. row rank equals column rank.

3. $\text{rank}(\mathbf{A}) = 0$ if and only if $\mathbf{A}$ is the zero matrix.
4. $\text{rank}(\mathbf{A}^\mathsf{T}\mathbf{A}) = \text{rank}(\mathbf{A}\mathbf{A}^\mathsf{T}) = \text{rank}(\mathbf{A})$.
5. $\text{rank}(\mathbf{AB}) \le \min\{\text{rank}(\mathbf{A}), \text{rank}(\mathbf{B})\}$ provided $\mathbf{A}$ and $\mathbf{B}$ are conformable.
6. $\text{rank}(\mathbf{A} + \mathbf{B}) \le \text{rank}(\mathbf{A}) + \text{rank}(\mathbf{B})$ for any $m \times n$ matrix $\mathbf{B}$.

7.2 Linear Independence

The notion of rank is deeply tied to the concept of *linear independence*. A vector $\mathbf{x}_i$ is linearly dependent on a set of n vectors $\mathbf{x}_1$, $\mathbf{x}_2$, . . ., $\mathbf{x}_n$ if there exits coefficients c_1, c_2, . . ., c_n such that

$$\mathbf{x}_i = c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \cdots + c_n\mathbf{x}_n$$

A set of vectors are *linearly dependent* if one of the vectors can be expressed as a linear combination of some of the others. This is analogous to saying there exists a set of coefficients $c_1, \ldots, c_n$, not all equal to zero, such that

$$c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \cdots + c_n\mathbf{x}_n = \mathbf{0}$$

Note the difference between $\mathbf{x}_1, \ldots, \mathbf{x}_n$, a set of n vectors; and $x_1, \ldots, x_n$, a set of n scalars that form the elements of a vector $\mathbf{x}$.

Imagine that a coefficient $c_i \neq 0$. If we move the term $c_i\mathbf{x}_i$ to the other side of the equation and divide by $-c_i$, we are back to the $\mathbf{x}_i = c_1\mathbf{x}_1 + \cdots + c_n\mathbf{x}_n$ definition of linear independence (although the coefficients will have changed by a factor of $-c_i$).

If a matrix with n rows has rank $k < n$, then $n - k$ of the rows are linearly dependent on the other k rows. Going back to our previous example, the matrix

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 2 & -4 \\ -2 & 0 & -6 \end{pmatrix} \xrightarrow{\text{EROs}} \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & 0 \end{pmatrix}$$

has rank 2 since there are two nonzero rows in the row-reduced matrix. Therefore, one of the rows must be linearly dependent on the other rows. Indeed, we see that the last row $\begin{pmatrix} -2 & 0 & -6 \end{pmatrix}$ is -2 times the first row $\begin{pmatrix} 1 & 0 & 3 \end{pmatrix}$. During row reduction by Gaussian elimination, any linearly dependent rows will be completely zeroed out, revealing the rank of the matrix.

Rank and linear dependence tell us about the information content of a coefficient matrix. If some of the rows of the coefficient matrix are linearly dependent, then matrix is rank deficient and no unique solution exists. These matrices are also information deficient – we do not have one independent expression for each variable. Without a separate piece of information for each variable, we cannot uniquely map between the input $\mathbf{x}$ and the output $\mathbf{y}$. However, if we introduce a separate parameter for each zeroed row, we are artificially providing the missing information. We can find a new solution every time we specify values for the parameters.

7.3 Homogeneous Systems ($\mathbf{Ax} = \mathbf{0}$)

A linear systems of equations is *homogeneous* if and only if the right hand side vector ($\mathbf{y}$) is equal to the zero vector ($\mathbf{0}$). Homogeneous systems always have at least one solution, $\mathbf{x} = \mathbf{0}$, since $\mathbf{A0} = \mathbf{0}$. The zero solution to a homogeneous system is called the *trivial solution*. Some homogeneous systems have a nontrivial solution, i.e. a solution $\mathbf{x} \neq \mathbf{0}$. If a homogeneous system has a nontrivial solution, then it has infinitely many solutions. We prove this result below.

Theorem. *Any linear combination of nontrivial solutions to a homogeneous linear system is also a solution.*

This proof is equivalent to showing that $\mathbf{Ax} = \mathbf{0}$ satisfies our definition of a linear system: $f(k_1x_1 + k_2x_2) = k_1f(x_1) + k_2f(x_2)$.

Proof. Suppose we had two solutions, $\mathbf{x}$ and $\mathbf{x}'$, to the homogeneous system $\mathbf{Ax} = \mathbf{0}$. Then

$$\begin{aligned}
\mathbf{A}(k\mathbf{x} + k'\mathbf{x}') &= \mathbf{A}(k\mathbf{x}) + \mathbf{A}(k'\mathbf{x}') \\
&= k(\mathbf{Ax}) + k'(\mathbf{Ax}') \\
&= k(\mathbf{0}) + k'(\mathbf{0}) \\
&= \mathbf{0}
\end{aligned}$$

Since there are infinitely many scalars k and k', we can generate infinitely many solutions to the homogeneous system $\mathbf{Ax} = \mathbf{0}$. □

There is a connection between the solvability of nonhomogeneous systems $\mathbf{Ax} = \mathbf{y}$ and the corresponding homogeneous system $\mathbf{Ax} = \mathbf{0}$. If there exists at least one solution to $\mathbf{Ax} = \mathbf{y}$ and a nontrivial solution to $\mathbf{Ax} = \mathbf{0}$, then there are infinitely many solutions to $\mathbf{Ax} = \mathbf{y}$. To see why, let $\mathbf{x}_{\text{nh}}$ be the solution to the nonhomogeneous system ($\mathbf{Ax}_{\text{nh}} = \mathbf{y}$) and $\mathbf{x}_{\text{h}}$ be a nontrivial solution to the homogeneous system $\mathbf{Ax}_{\text{h}} = \mathbf{0}$. Then any of the infinite linear combinations $\mathbf{x}_{\text{nh}} + k\mathbf{x}_{\text{h}}$ is also a solution to $\mathbf{Ax} = \mathbf{y}$ since

$$\mathbf{A}(\mathbf{x}_{\text{nh}} + k\mathbf{x}_{\text{h}}) = \mathbf{Ax}_{\text{nh}} + k\mathbf{Ax}_{\text{h}} = \mathbf{y} + k\mathbf{0} = \mathbf{y}$$

7.4 General Solvability

We can use the rank of the coefficient matrix and the augmented matrix to determine the existence and number of solutions for any linear system of equations. The relationship between solvability and rank is captured by the Rouché-Capelli theorem:

Theorem. *A linear system* $\mathbf{Ax} = \mathbf{y}$ *where* $\mathbf{x} \in \mathbb{R}^n$ *has a solution if and only if the rank of the coefficient matrix equals the rank of the augmented matrix, i.e.* $\operatorname{rank}(\mathbf{A}) = \operatorname{rank}([\mathbf{A}\,\mathbf{y}])$. *Furthermore, the solution is unique if* $\operatorname{rank}(\mathbf{A}) = n$*; otherwise there are infinitely many solutions.*

Proof. We will sketch several portions of this proof to give intuition about the theorem. A more rigorous proof is beyond the scope of this class.

1. **Homogeneous systems.** For a homogeneous system $\mathbf{Ax} = \mathbf{0}$, we know that $\operatorname{rank}(\mathbf{A}) = \operatorname{rank}([\mathbf{A}\,\mathbf{0}])$. (Since the rank of $\mathbf{A}$ is equal to the number of nonzero columns, adding another column of zeros will never change the rank.) Thus, we know that homogeneous systems are always solvable, at least by the trivial solution $\mathbf{x} = \mathbf{0}$. If $\operatorname{rank}(\mathbf{A}) = n$, then the trivial solution is unique and is the only solution. If $\operatorname{rank}(\mathbf{A}) < n$, there are infinitely many parameterized solutions.

2. **Full rank, nonhomogeneous systems**. For a nonhomogeneous system ($\mathbf{Ax} = \mathbf{y}$, $\mathbf{y} \neq \mathbf{0}$), we expect a unique solution if and only if adding the column $\mathbf{y}$ to the coefficient matrix doesn't change the rank. For this to be true, $\mathbf{y}$ must be linearly dependent on the other columns in $\mathbf{A}$; otherwise, adding a new linearly independent column would increase the rank. If $\mathbf{y}$ is linearly dependent on the n columns of $\mathbf{A}$, it must be true that there exists weights $c_1, c_2, \ldots, c_n$ such that

$$c_1\mathbf{A}(:,1) + c_2\mathbf{A}(:,2) + \cdots + c_n\mathbf{A}(:,n) = \mathbf{y}$$

   based on the definition of linear dependence. But the above expression can be rewritten in matrix form as

$$\left(\mathbf{A}(:,1)\,\mathbf{A}(:,2)\,\cdots\,\mathbf{A}(:,n)\right)\begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \mathbf{Ac} = \mathbf{y}$$

   which shows that the system has a unique solution $\mathbf{x} = \mathbf{c}$.

Take time to understand the connection between the solvability of $\mathbf{Ax} = \mathbf{y}$ and the ability to express $\mathbf{y}$ as a linear combination of the columns in $\mathbf{A}$. This relationship is important for Part III of the course.

3. **Rank deficient, nonhomogeneous systems**. Let $\operatorname{rank}(\mathbf{A}) = k < n$. Then the row-reduced form of $\mathbf{A}$ will have k rows that resemble the identity matrix

and $n-k$ rows of all zeros:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2k} & \cdots & a_{2n} \\ & \vdots & & \vdots & & \\ a_{k1} & a_{k2} & \cdots & a_{kk} & \cdots & a_{kn} \\ a_{k+1,1} & a_{k+1,2} & \cdots & a_{k+1,k} & \cdots & a_{k+1,n} \\ & \vdots & & \vdots & & \\ a_{n1} & a_{n2} & \cdots & a_{nk} & \cdots & a_{nn} \end{pmatrix} \xrightarrow{\text{EROs}} \begin{pmatrix} 1 & a'_{12} & \cdots & a'_{1k} & \cdots & a'_{1n} \\ 0 & 1 & \cdots & a'_{2k} & \cdots & a'_{2n} \\ & \vdots & & & \vdots & \\ 0 & 0 & \cdots & 1 & \cdots & a'_{kn} \\ 0 & 0 & \cdots & 0 & \cdots & 0 \\ & \vdots & & & \vdots & \\ 0 & 0 & \cdots & 0 & \cdots & 0 \end{pmatrix}$$

Now imagine we performed the same row reduction on the augmented matrix $(\mathbf{A}\,\mathbf{y})$. We would still end up with $n-k$ rows with zeros in the first n columns (the columns of $\mathbf{A}$):

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} & \cdots & a_{1n} & y_1 \\ a_{21} & a_{22} & \cdots & a_{2k} & \cdots & a_{2n} & y_2 \\ & \vdots & & \vdots & & & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} & \cdots & a_{kn} & y_k \\ a_{k+1,1} & a_{k+1,2} & \cdots & a_{k+1,k} & \cdots & a_{k+1,n} & y_{k+1} \\ & \vdots & & \vdots & & & \\ a_{n1} & a_{n2} & \cdots & a_{nk} & \cdots & a_{nn} & y_n \end{pmatrix} \xrightarrow{\text{EROs}} \begin{pmatrix} 1 & a'_{12} & \cdots & a'_{1k} & \cdots & a'_{1n} & y'_1 \\ 0 & 1 & \cdots & a'_{2k} & \cdots & a'_{2n} & y'_2 \\ & \vdots & & & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 & \cdots & a'_{kn} & y'_k \\ 0 & 0 & \cdots & 0 & \cdots & 0 & y'_{k+1} \\ & \vdots & & & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 & \cdots & 0 & y'_n \end{pmatrix}$$

We know that if $y'_{k+1}, \ldots, y'_n = 0$, we can solve this system by designating $n-k$ parameters for the variables $x_{k+1}, \ldots, x_n$ for which we have no information. However, notice what happens if any of the values $y'_{k+1}, \ldots, y'_n$ are nonzero. Then we have an expression of the form $0 = y'_i \neq 0$, which is nonsensical. Therefore, the only way we can solve this system is by requiring that $y'_{k+1}, \ldots, y'_n = 0$. This is exactly the requirement that the rank of the augmented matrix equal k, the rank of the matrix $\mathbf{A}$ by itself. If any of the $y'_{k+1}, \ldots, y'_n$ are nonzero, then the augmented matrix has one fewer row of zeros, so the rank of the augmented matrix would be greater than the rank of the original matrix. There are two ways to interpret this result. First, we require that the right hand side ($\mathbf{y}$) doesn't “mess up" our system by introducing a nonsensical expression. Second, if a row i in the matrix $\mathbf{A}$ is linearly dependent on the other rows in $\mathbf{A}$, the corresponding values y_i must have the same dependency on the other values in $\mathbf{y}$. If so, when the row i is zeroed out during row reduction, the value y_i will also be zeroed out, avoiding any inconsistency.

$\square$

7.5 Rank and Matrix Inversion

For a general nonhomogeneous system $\mathbf{Ax} = \mathbf{y}$ with $\mathbf{A} \in \mathbb{R}^{n\times n}$, we know that a unique solution only exists if $\text{rank}(\mathbf{A}) = \text{rank}([\mathbf{A}\,\mathbf{y}]) = n$. If $\text{rank}(\mathbf{A}) = n$, we know that $\mathbf{A}$ can be transformed into reduced row form without generating any rows with all zero entries. We also know that if an inverse $\mathbf{A}^{-1}$ exists for $\mathbf{A}$, we can use the inverse to uniquely solve for $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$. Putting these facts together, we can now definitely state necessary and sufficient conditions for matrix inversion:

An $n \times n$ matrix $\mathbf{A}$ has an inverse if and only if $\text{rank}(\mathbf{A}) = n$.

7.6 Summary

We've shown in this chapter the tight connections between matrix inversion, solvability, and the rank of a matrix. We will use these concepts many times to understand the solution of linear systems. However, we've also argued that despite their theoretical importance, these concepts have limited practical utility for solving linear systems. For example, computing the rank of a matrix requires transforming the matrix into reduced echelon form. This requires the same computations as solving a linear system involving the matrix, so one would rarely check the rank of a coefficient matrix before attempting to solve a linear system. Instead, we will see rank emerge as a useful tool only when considering matrices by themselves in Part III of this course.

Chapter 8

Linear Models and Regression

In Chapter 7 we learned that not all linear systems are solvable, depending on the amount and consistency of the information contained in the corresponding systems of equations. Too little information yielded an infinite number of solutions, and some systems have no solution at all. In all cases, we treated the entries of the coefficient matrix **A** and the output vector **y** as exact. We never doubted the accuracy of our measurements of either the system or its outputs. This view may be sufficient for linear algebra textbooks, but it is detached from the realities of the real world faced by engineers.

This chapter addresses two related problems. First, how do we solve linear systems when our measurements of either the system or its outputs are noisy? The practical solution for noisy data is to simply make more measurements, but leads to a second problem: How do we solve a linear system when we have more information than is required by the solvability conditions described in Chapter 7? The answer to both of these questions is *linear regression*, a powerful tool that combines linear algebra with statistics. This chapter introduces regression and the tools used to fit linear systems to noisy data. Linear regression is the first machine learning technique we've encountered, so we will present linear regression within a general framework of minimizing a loss function. This is not the usual way to present linear regression, but it will enable us to connect regression with other machine learning techniques.

The following chapter (Chapter 9) focuses on the practice of linear modeling, i.e. how to build and interpret models. You may be surprised by the flexibility of linear models and how many systems — both linear and nonlinear — can be analyzed with this single technique. Furthermore, all of these models can be solved using the matrix formalism described in this chapter.

8.1 The Problems of Real Data

Imagine you measured an input (x) and the corresponding output (y). You hypothesized a linear relationship between x and y, i.e.

$$y = \beta x.$$

We refer to the unknown quantity β as a *parameter*. Our goal is to estimate the value of β using our input and output measurements. If $y = 2.4$ when $x = 2$, then you can calculate the value of the parameter β with simple algebra.

$$\beta = y/x = 2.4/2 = 1.2$$

You could plot the relationship between x and y, which is just the line $y = 1.2x$. Given an input x, it is easily to calculate the corresponding value y. All of this works because we've assumed our measurements of the input x and output y are exact. With only one unknown (β), we have sufficient information to calculate a value with only one pair of input/output observations. However, life is messy. Measurements are noisy and filled with error and uncertainty. Even if the underlying relationship between y and x was really a factor of 1.2, we would never observe a value of y that was exactly 1.2 times x.

To compensate for the imprecision of the real world, engineers make multiple observations of their systems. Imagine we made five "noisy" measurements of the input x and output y, which we will call x^{true} and y^{true}.

x^{true}	y^{true}
0.07	−0.05
0.16	0.40
0.48	0.66
0.68	0.65
0.83	1.12

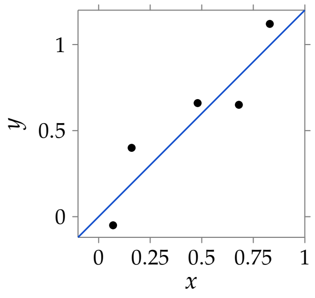


Figure 8.1: Five noisy observations (points) of the linear relationship $y = 1.2\,x$.

We refer to our observations of x and y as "true" because they are values we have actually measured. They are true in the sense that they appeared in the real world. Unfortunately, each pair of measurements gives an different estimate for the value of the parameter β:

1. $\beta = -0.05/0.07 = -0.7$
2. $\beta = 0.40/0.16 = 2.5$
3. $\beta = 0.66/0.48 = 1.3$
4. $\beta = 0.65/0.68 = 0.9$
5. $\beta = 1.12/0.83 = 1.3$

Using Figure 8.1 we can be reasonably certain that $y = 1.2x$ is a good representation of the relationship between y and x, so we expect that β is approximately 1.2. But our estimates for β vary wildly, from -0.7 to 2.5, and none of the estimates match the true value of 1.2. We could average the individual estimates to produce a single composite result ($\beta = 1.09$), but we would still miss the true value by nearly 9%.

Using the same five observations and the statistical techniques in this chapter, we can estimate β to be 1.21, an error of less than 1%.

This leads us to the central question in statistical modeling: "How accurate is our model?" So far we've been comparing our estimate of the parameter β to its "real" value, but this cannot be a solution. If we knew the actual value of β beforehand, then we don't need to estimate β from data! In practice we will never know the actual values of the parameters, so we cannot judge our models by the estimates of their parameters. Instead, the measure of a model is the accuracy of its predictions, and predictive accuracy is something we can easily quantify.

8.2 The Loss Function

After we estimate a value for the parameter β, we can make predictions using our model. The outputs of the model are only predictions; they have not been observed in the real world and exist only as informed guesses made by the model. We will call the predicted outputs from the model y^{pred}.

Our model's predictions (y^{pred}) will never match the observed values (y^{true}) exactly. Remember that the observed values include noise from both random fluctuations in the system and uncertainty in our measurement devices. We can compare the measured outputs (y^{true}) and the predicted outputs (y^{pred}) that we calculate using our model and the observed inputs ($y^{\text{pred}} = 1.2\,x^{\text{true}}$). The differences between the predicted outputs and the observed outputs ($y^{\text{pred}} - y^{\text{true}}$) are called the *residuals*.

		Prediction	Residual
x^{true}	y^{true}	$(y^{\text{pred}} = 1.2\,x^{\text{true}})$	$(y^{\text{pred}} - y^{\text{true}})$
0.07	−0.05	0.084	0.134
0.16	0.40	0.192	−0.208
0.48	0.66	0.576	−0.084
0.68	0.65	0.816	0.166
0.83	1.12	0.996	−0.124

Our goal when fitting a model (finding values for the unknown parameters) is to minimize the residuals. But "minimizing the residuals" is subjective. We need a method to penalize large residuals and encourage small ones, and there are many choices for converting a residual into a penalty. For example, we might accept one very large residual if doing so makes all the other residuals tiny. Alternatively, we might want all of our residuals to be of a similar size so our model is equally uncertain over all possible inputs. The choice of penalties is entirely up to us. There is no wrong answer, but different penalties will give different parameter estimates for the same data.

The penalties calculated from the residuals is called the *loss*. There are many *loss functions*, each with strengths and weaknesses; however, all loss functions must have two properties. First, the loss must never be negative. Negative penalties are nonsensical and lead to problems during minimization. Second, the loss should be zero only when the corresponding residual is zero (i.e. when $y^{\text{pred}} = y^{\text{true}}$). If there is any discrepancy between a model's predictions and an observed value, there must also be a nonzero loss to signal that the model's predictions could be improved. Conversely, any improvement in the model's predictions — no matter how small — must also lead to a decrease in the loss.

There are two common loss functions that satisfy the above properties. The first is the *absolute loss*

$$\text{absolute loss}(y^{\text{true}}, y^{\text{pred}}) = \left|y^{\text{true}} - y^{\text{pred}}\right|.$$

The second is the *quadratic loss*

$$\text{quadratic loss}(y^{\text{true}}, y^{\text{pred}}) = \left(y^{\text{true}} - y^{\text{pred}}\right)^2.$$

For fitting linear models we will use the quadratic loss because it has several advantages over the absolute loss:

1. Quadratic functions are continuously differentiable, while the derivative of the absolute value is discontinuous at zero. A continuous first derivative makes it easy to optimize functions involving the squared error.

2. The quadratic loss more harshly penalizes predictions that are far from the observed values. If a prediction is twice as far from an observation, the quadratic loss increases by a factor of four while the absolute loss only doubles. We will see shortly that assigning large penalties to far away points is more intuitive.

3. For linear models, there is always a single solution when minimizing quadratic loss, but there can be infinitely many solutions that minimize the absolute loss. We prefer having a unique solution whenever possible.

8.2.1 Loss Depends on Parameters, Not Data

The loss function quantifies how we've chosen to penalize the differences between the model predictions and the observed data. We must always remember that the loss is a function of the model's parameters, not the data. We will refer to the loss function as L, so the quadratic loss for a single piece of data is

$$L_i = \left(y_i^{\text{pred}} - y_i^{\text{true}}\right)^2 .$$

The subscript i reminds us that we are only considering the loss of one $(x^{\text{true}}, y^{\text{true}})$ pair in the dataset. Our simple linear model has the form $y^{\text{pred}} = \beta x^{\text{true}}$, which we substitute into the loss function.

$$L_i = \left(\beta x_i^{\text{true}} - y_i^{\text{true}}\right)^2$$

Consider the second observation from the dataset for Figure 8.1: $x_2^{\text{true}} = 0.16$, $y_2^{\text{true}} = 0.40$. The loss for this single point is

$$L_2 = \left(0.16\beta - 0.40\right)^2 .$$

The loss L_2 depends on β, not the quantities x_2^{true} and y_2^{true}. We minimize the model's loss by adjusting β. The training data x^{true} and y^{true} are fixed — we measured them before we started fitting the model. This can seem unusual, as previous math courses have conditioned us to always think of x and y as variables, not fixed quantities. We will write the loss function as $L(\beta)$ to remind us that loss is a function of the model's parameters.

8.2.2 Minimizing the Total Loss

In the previous section we discussed the loss for a single point. Let's write out the loss for all five points in the dataset from Figure 8.1.

$$
\begin{aligned}
L_1(\beta) &= \left(0.07\beta + 0.05\right)^2 \\
L_2(\beta) &= \left(0.16\beta - 0.40\right)^2 \\
L_3(\beta) &= \left(0.48\beta - 0.66\right)^2 \\
L_4(\beta) &= \left(0.68\beta - 0.65\right)^2 \\
L_5(\beta) &= \left(0.83\beta - 1.12\right)^2
\end{aligned}
$$

There is only one parameter β that appears in all five losses. When we fit a model by selecting a value for β, our goal is to minimize the total loss across all five points:

$$\text{total loss} = L(\beta) = L_1 + L_2 + L_3 + L_4 + L_5 = \sum_i L_i.$$

All of the individual losses depend on β, and this shared dependence complicates our search for a single "best" parameter value. Imagine we focused only on the first loss, L_1. The value of β that minimizes L_1 is $\beta = -0.05/0.07 \approx -0.714$. This value for β is ideal for the loss L_1, setting this individual loss to zero. However, the value $\beta = -0.714$ is a terrible choice for the other points in the dataset.

$$
\begin{aligned}
L_1(-0.714) &= [0.07(-0.714) + 0.05]^2 = 0 \\
L_2(-0.714) &= [0.16(-0.714) - 0.40]^2 = 0.264 \\
L_3(-0.714) &= [0.48(-0.714) - 0.66]^2 = 1.005 \\
L_4(-0.714) &= [0.68(-0.714) - 0.65]^2 = 1.289 \\
L_5(-0.714) &= [0.83(-0.714) - 1.12]^2 = 2.933
\end{aligned}
$$

The total loss at for the parameter value $\beta = -0.714$ is

$$L(-0.714) = 0 + 0.264 + 1.005 + 1.289 + 2.933 = 5.492.$$

Compare this loss to that of the optimal parameter value $\beta = 1.21$ (which we will find shortly):

$$L(1.21) = 0.018 + 0.043 + 0.006 + 0.230 + 0.013 = 0.110.$$

In section 8.3 we will learn how to find parameter values that minimize the total loss across the entire dataset. Until then, remember that the "best" parameter values minimize the total loss, and the best parameters are rarely optimal for minimizing the loss of an individual point taken alone.

8.2.3 Loss vs. Error

The machine learning community almost exclusively uses the term loss to describe the penalized difference between a model's outputs and the training data. Although linear regression is a type of machine learning, the linear modeling field often refers to loss as "error" and quadratic loss as "squared error". Minimizing the quadratic loss is therefore known as "least squares" estimation. All of these terms are correct, and you should be familiar with both conventions. We have decided to use loss exclusively throughout the book for two reasons.

1. **Consistency**. One of our goals is to unify the presentation of machine learning and draw parallels between linear and nonlinear models. Subsequent chapters on machine learning use "loss" as it is standard in those fields, and we want to avoid duplicate terms.

2. **Clarity**. Some students find it difficult to distinguish the many types of error in linear models. There is error in the measured training data, and the parameter estimates have an associated standard error. Referring to the loss as "error" only adds to the confusion. Also, students often associate the term "error" with uncertainty, which is not a correct interpretation of a model's loss. Models have loss not because their predictions are uncertain (at least for deterministic models), but because they simplified representations of the real world. We like the term "loss" as a reminder that switching from data to a model causes an inherent loss of information.

All nomenclature is preference, and there is no unique solution. We will use margin notes to remind the reader of alternative names.

8.3 Fitting Linear Models

The total loss function provides a quantitative measure of how well our model fits the training data. There are three steps for minimizing the total loss of a linear model.

1. Choose a model that you think explains the relationship between inputs (x) and outputs (y). The models should contain unknown parameters (β) that

you will fit to a set of observations. **The model should be linear with respect to the parameters (β). It does not need to be linear with respect to the inputs (x) or outputs (y).** We will discuss the linearity of models in Chapter 9.

2. To find values for the unknown parameters (β), we will minimize the total loss between the observed outputs (y^{true}) and the outputs predicted from the model
$$\min_{\beta} \sum_{\text{data}} L_i(\beta)$$
or, for the specific case when we choose the quadratic loss,
$$\min_{\beta} \sum_{i} \left(y_i^{\text{pred}} - y_i^{\text{true}}\right)^2.$$
Substitute the model you selected in Step 1 in place of y^{pred} in the above minimization.

3. Minimize the function by taking the derivative of the total loss and setting it equal to zero. Solve for the unknown parameters β. You should also check that your solution is a minimum, not a maximum or inflection point by checking that the second derivative is positive.

8.3.1 Single Parameter, Constant Models: $y = \beta_0$

The simplest linear model has only a single parameter and no dependence on the inputs:
$$y^{\text{pred}} = \beta_0.$$
This might seem like a silly model. Given an input observation x^{true}, the model ignores the input and predicts that y^{pred} will always be equal to β_0. For example, imagine we are predicting someone's height (y^{pred}) given their age (x^{true}). Rather than make a prediction based on the person's age, we simply guess the same height for everyone (β_0).

Regardless of the utility of such a simple model, let's fit it to a set of n pairs of observations ($x_i^{\text{true}}, y_i^{\text{true}}$). We've already completed Step 1 by choosing the model $y^{\text{pred}} = \beta_0$. We begin Step 2 by writing our objective: to choose a value for β_0 using the n observations that minimizes the total loss:
$$\min_{\beta_0} \sum_{i=1}^{n} \left(y_i^{\text{pred}} - y_i^{\text{true}}\right)^2.$$

Now we substitute our model in place of y^{pred}, making our objective

$$\min_{\beta_0} \sum_{i=1}^{n} \left(\beta_0 - y_i^{\text{true}}\right)^2 .$$

To minimize the loss we find where the derivative of the total loss **with respect to the parameter** β_0 is zero.

Remember that the derivative of a sum is the sum of the derivative.

Also, the sum

$$\sum_i^n C = nC$$

for any value C that does not depend on i.

$$\begin{aligned}
\frac{d}{d\beta_0}\left(\sum_{i=1}^{n}(\beta_0 - y_i^{\text{true}})^2\right) &= 0\\
\sum_{i=1}^{n}\left(\frac{d}{d\beta_0}(\beta_0 - y_i^{\text{true}})^2\right) &= 0\\
\sum_{i=1}^{n}2(\beta_0 - y_i^{\text{true}})(1) &= 0\\
2\sum_{i=1}^{n}(\beta_0 - y_i^{\text{true}}) &= 0\\
\sum_{i=1}^{n}\beta_0 - \sum_{i=1}^{n}y_i^{\text{true}} &= 0\\
n\beta_0 - \sum_{i=1}^{n}y_i^{\text{true}} &= 0
\end{aligned}$$

We can rearrange the final equation and discover that the optimal value for the parameter β_0 is

$$\beta_0 = \frac{1}{n}\sum_{i=1}^{n} y_i^{\text{true}} = \text{mean}[y^{\text{true}}].$$

If our strategy is to always guess the same output value (β_0), the best value to guess is the mean of the observed outputs y^{true}. Said another way, the mean is the best fit of a constant model to a set of data. If we need to represent a set of numbers with a single number, choosing the mean minimizes the quadratic loss.

The quadratic loss is also called the "squared error", so the least squares estimator minimizes is also said to minimize the "sum of squares".

A couple interesting things come from this result. First, now you know where the mean comes from. It is the *least squares* estimate for a set of points. (The phase "least squares" is a convenient way of saying "minimizes the sum of the quadratic loss".) Second, the mean does not minimize the absolute value of the residuals —

this is a common misconception! If we repeated the same calculation using the absolute loss instead of the quadratic loss we would discover that the least absolute estimator for a set of numbers is the median, not the mean.

Going further, if we make our loss function binary (the loss is zero if $y^{\text{pred}} = y^{\text{true}}$ and one otherwise), the best estimator is called the *mode* — the most frequent value in the set of observed outputs.

8.3.2 Two Parameter Models: $y = \beta_0 + \beta_1 x$

Let's fit a more complicated model that uses the observed inputs x^{true} when making output predictions y^{true}. Our model has the form

$$y^{\text{pred}} = \beta_0 + \beta_1 x^{\text{true}}$$

with two unknown parameters β_0 and β_1. This is a linear model with respect to the parameters. We discovered earlier that the functions of the form $y = \beta_0 + \beta_1 x$ are not linear with respect to x (they are affine), but remember that x^{true} is not an independent variable in the model. It is a known, observed value — a constant. The unknowns in a linear model are the parameters, not y or x.

Now that we've chosen our model, we write our objective: to minimize the total quadratic loss.

$$\min_{\beta_0,\beta_1} \sum_{i=1}^{n} \left(y_i^{\text{pred}} - y_i^{\text{true}}\right)^2$$

Notice we are minimizing over both parameters β_0 and β_1. Substituting our model for the value y^{pred} yields

$$\min_{\beta_0,\beta_1} \sum_{i=1}^{n} \left(\beta_0 + \beta_1 x_i^{\text{true}} - y_i^{\text{true}}\right)^2$$

The total loss is minimized when the derivatives with respect to both β_0 and β_1 are zero. Let's start by taking the derivative or the loss with respect to β_0.

We are using partial derivatives since our loss is a function of more than one unknown parameter.

$$\begin{aligned}\frac{\partial}{\partial \beta_0}\sum_{i=1}^{n}\left(\beta_0+\beta_1 x_i^{\text{true}}-y_i^{\text{true}}\right)^2 &= \sum_{i=1}^{n}\frac{\partial}{\partial \beta_0}\left(\beta_0+\beta_1 x_i^{\text{true}}-y_i^{\text{true}}\right)^2\\ &= 2\sum_{i=1}^{n}\left(\beta_0+\beta_1 x_i^{\text{true}}-y_i^{\text{true}}\right)\\ &= 2\left(\sum_{i=1}^{n}\beta_0+\sum_{i=1}^{n}\beta_1 x_i^{\text{true}}-\sum_{i=1}^{n}y_i^{\text{true}}\right)\\ &= 2\left(n\beta_0+\beta_1\sum_{i=1}^{n}x_i^{\text{true}}-\sum_{i=1}^{n}y_i^{\text{true}}\right)\end{aligned}$$

We set this derivative equal to zero and solve for β_0.

We call β_0 (the affine parameter in a linear model) the *grand mean* since it equals the mean of the outputs when all inputs are zero.

$$\begin{aligned}\beta_0 &= \frac{1}{n}\sum_{i=1}^{n}y_i^{\text{true}}-\beta_1\frac{1}{n}\sum_{i=1}^{n}x_i^{\text{true}}\\ &= \text{mean}[y^{\text{true}}]-\beta_1\text{mean}[x^{\text{true}}]\end{aligned}$$

We see that β_0 depends on the mean input, the mean output, and the parameter β_1. Let's substitute our value for β_0 into the total loss function.

$$\begin{aligned}\sum_{i=1}^{n}\left(\beta_0+\beta_1 x_i^{\text{true}}-y_i^{\text{true}}\right)^2 &= \sum_{i=1}^{n}\left(\text{mean}[y^{\text{true}}]-\beta_1\text{mean}[x^{\text{true}}]+\beta_1 x_i^{\text{true}}-y_i^{\text{true}}\right)^2\\ &= \sum_{i=1}^{n}\Big(\beta_1\left(x_i^{\text{true}}-\text{mean}[x^{\text{true}}]\right)-\left(y_i^{\text{true}}-\text{mean}[y^{\text{true}}]\right)\Big)^2\end{aligned}$$

Now we find the optimal value for the parameter β_1. First we differentiate the total loss with respect to β_1.

$$\frac{\partial}{\partial \beta_1}\sum_{i=1}^{n}\Big(\beta_1\left(x_i^{\text{true}}-\text{mean}[x^{\text{true}}]\right)-\left(y_i^{\text{true}}-\text{mean}[y^{\text{true}}]\right)\Big)^2$$

$$
\begin{aligned}
&= \sum_{i=1}^{n} \frac{\partial}{\partial \beta_1} \left(\beta_1 \left(x_i^{\text{true}} - \text{mean}[x^{\text{true}}]\right) - \left(y_i^{\text{true}} - \text{mean}[y^{\text{true}}]\right)\right)^2 \\
&= 2 \sum_{i=1}^{n} \left(\beta_1 \left(x_i^{\text{true}} - \text{mean}[x^{\text{true}}]\right) - \left(y_i^{\text{true}} - \text{mean}[y^{\text{true}}]\right)\right) \left(x_i^{\text{true}} - \text{mean}[x^{\text{true}}]\right) \\
&= 2 \sum_{i=1}^{n} \left(\beta_1 \left(x_i^{\text{true}} - \text{mean}[x^{\text{true}}]\right)^2 - \left(x_i^{\text{true}} - \text{mean}[x^{\text{true}}]\right) \left(y_i^{\text{true}} - \text{mean}[y^{\text{true}}]\right)\right)
\end{aligned}
$$

We set the derivative equal to zero and solve for the parameter β_1.

$$
\beta_1 = \frac{\sum_{i=1}^{n} \left(x_i^{\text{true}} - \text{mean}[x^{\text{true}}]\right) \left(y_i^{\text{true}} - \text{mean}[y^{\text{true}}]\right)}{\sum_{i=1}^{n} \left(x_i^{\text{true}} - \text{mean}[x^{\text{true}}]\right)^2}
$$

Let's try fitting the expression $y^{\text{pred}} = \beta_0 + \beta_1 x^{\text{true}}$ to the data from earlier in this chapter.

x^{true}	y^{true}
0.07	−0.05
0.16	0.40
0.48	0.66
0.68	0.65
0.83	1.12

First we calculate the means of the inputs and outputs.

$$
\begin{aligned}
\text{mean}[x^{\text{true}}] &= (1/5)(0.07 + 0.16 + 0.48 + 0.68 + 0.83) = 0.44 \\
\text{mean}[y^{\text{true}}] &= (1/5)(-0.05 + 0.40 + 0.66 + 0.65 + 1.12) = 0.56
\end{aligned}
$$

Now we can calculate the value for the parameter β_1. It's easiest to make a table.

x^{true}	y^{true}	$(x^{\text{true}} - \text{mean}[x^{\text{true}}])(y^{\text{true}} - \text{mean}[y^{\text{true}}])$	$(x^{\text{true}} - \text{mean}[x^{\text{true}}])^2$
0.07	−0.05	$(0.07 - 0.44)(-0.05 - 0.56) = 0.23$	$(0.07 - 0.44)^2 = 0.14$
0.16	0.40	$(0.16 - 0.44)(0.40 - 0.56) = 0.044$	$(0.16 - 0.44)^2 = 0.081$
0.48	0.66	$(0.48 - 0.44)(0.66 - 0.56) = 0.0037$	$(0.48 - 0.44)^2 = 0.0013$
0.68	0.65	$(0.68 - 0.44)(0.65 - 0.56) = 0.022$	$(0.68 - 0.44)^2 = 0.056$
0.83	1.12	$(0.83 - 0.44)(1.12 - 0.56) = 0.22$	$(0.83 - 0.44)^2 = 0.15$

$$\begin{aligned}
\beta_1 &= \frac{\sum_{i=1}^{n}\left(x_i^{\text{true}} - \text{mean}[x^{\text{true}}]\right)\left(y_i^{\text{true}} - \text{mean}[y^{\text{true}}]\right)}{\sum_{i=1}^{n}\left(x_i^{\text{true}} - \text{mean}[x^{\text{true}}]\right)^2} \\
&= \frac{0.23 + 0.044 + 0.0037 + 0.022 + 0.22}{0.14 + 0.081 + 0.0013 + 0.056 + 0.15} \\
&= 1.21
\end{aligned}$$

We can use the value of the parameter β_1 to find the other parameter β_0.

$$\begin{aligned}
\beta_0 &= \text{mean}[y^{\text{true}}] - \beta_1\text{mean}[x^{\text{true}}] \\
&= 0.56 - (1.21)(0.44) \\
&= 0.020
\end{aligned}$$

According to our five observations, the best fit least squares estimate is

$$y = 0.020 + 1.21x.$$

This agrees well with our hypothesized relationship that $y = 1.2x$.

8.4 Matrix Formalism for Linear Models

You might be thinking that there has to be an easier method for fitting linear models. Finding formulae for the parameters is unwieldy, and the problem only worsens as the number of parameters grows. Fortunately, linear algebra can help.

Let's return to our two parameter model $y = \beta_0 + \beta_1 x$. Using the five data points from the previous section, we can write five linear equations, one for each point.

$$\begin{aligned}
-0.05 &= \beta_0 + 0.07\beta_1 + \epsilon_1 \\
0.40 &= \beta_0 + 0.16\beta_1 + \epsilon_2 \\
0.66 &= \beta_0 + 0.48\beta_1 + \epsilon_3 \\
0.65 &= \beta_0 + 0.68\beta_1 + \epsilon_4 \\
1.12 &= \beta_0 + 0.83\beta_1 + \epsilon_5
\end{aligned}$$

All we've done to write these equations is substituted the observed values for x and y and added an *residual term* (ϵ_i). Remember that each observation is imprecise, so the observed value of y will never exactly equal the predicted value $\beta_0 + \beta_1 x$. Since our equations must be exact, we add a term to each equation to hold the residual between the predicted and observed values. The same equations can be written in

The parameters β_0 and β_1 are the same for every equation, but each equation has its own residual term. In some fields the residual terms are called "errors".

matrix form as

$$\begin{pmatrix}-0.05\\0.40\\0.66\\0.65\\1.12\end{pmatrix} = \begin{pmatrix}1 & 0.07\\1 & 0.16\\1 & 0.48\\1 & 0.68\\1 & 0.83\end{pmatrix}\begin{pmatrix}\beta_0\\\beta_1\end{pmatrix} + \begin{pmatrix}\epsilon_1\\\epsilon_2\\\epsilon_3\\\epsilon_4\\\epsilon_5\end{pmatrix}.$$

Or, more succinctly,

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}.$$

There are several noteworthy things about the above expression.

- The variable $\mathbf{y}$ is a vector of the outputs (responses), $\boldsymbol{\epsilon}$ is a vector of residuals, and $\boldsymbol{\beta}$ is a vector of the unknown parameters.
- The inputs (or predictor variables) form a matrix $\mathbf{X}$ called the *model matrix*.
- The first column in $\mathbf{X}$ is all ones. This column corresponds to the constant parameter in the model, β_0.
- The unknowns in the equation are the parameters in the vector $\boldsymbol{\beta}$, not the values in the matrix $\mathbf{X}$. The values in $\mathbf{X}$ are known inputs from our dataset.

The residuals $\boldsymbol{\epsilon}$ are also unknown, but we do not solve for these explicitly. Once we have estimates for the parameters, we can calculate the residuals using $\boldsymbol{\epsilon} = \mathbf{y} - \mathbf{X}\boldsymbol{\beta}$.

Fitting a linear model involves finding a set of values for the vector $\boldsymbol{\beta}$. There are a few complications to solving the linear system $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$. First, our goal is not to find just any values for the parameters in $\boldsymbol{\beta}$, but to find the values that minimize the square of the residual terms in $\boldsymbol{\epsilon}$ (i.e. the least squares solution). Second, the matrix $\mathbf{X}$ is almost never square. We often have more rows (observations) that we have columns (parameters) to compensate for the noise in our measurements.

Fortunately, there is a tool from matrix theory — the *pseudoinverse* — that overcomes both these difficulties. The least squares solution to the problem $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$ is

$$\boldsymbol{\beta} = \mathbf{X}^{+}\mathbf{y}$$

where the matrix $\mathbf{X}^{+}$ is the pseudoinverse of the matrix $\mathbf{X}$.

8.5 The Pseudoinverse

Recall that a matrix is invertible if and only if it is square and full rank. For linear models the model matrix $\mathbf{X}$ is almost never square, so the matrix inverse $\mathbf{X}^{-1}$ does not exist. However, all nonsquare matrices have a pseudoinverse that somewhat approximates the behavior of the true inverse. Below are some properties of the pseudoinverse.

The pseudoinverse is also called the Moore-Penrose inverse.

1. The pseudoinverse $\mathbf{X}^+$ exists for *any* matrix $\mathbf{X}$.
2. The pseudoinverse of a matrix is unique.
3. If a square matrix $\mathbf{X}$ has a true inverse $\mathbf{X}^{-1}$, then $\mathbf{X}^+ = \mathbf{X}^{-1}$.
4. The pseudoinverse of a pseudoinverse is the original matrix: $(\mathbf{X}^+)^+ = \mathbf{X}$.
5. If $\dim(\mathbf{X}) = m \times n$, then $\dim(\mathbf{X}^+) = n \times m$.
6. It is **not** generally true that $\mathbf{X}^+\mathbf{X} = \mathbf{X}\mathbf{X}^+ = \mathbf{I}$. However, if $\mathbf{X}$ has full column rank then $\mathbf{X}^+\mathbf{X} = \mathbf{I}$, and if $\mathbf{X}$ has full row rank then $\mathbf{X}\mathbf{X}^+ = \mathbf{I}$.
7. If a matrix has only real entries, then so does its pseudoinverse.
8. If a matrix $\mathbf{X}$ is full rank, then $\mathbf{X}^+ = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}$.

It is always true that $\mathbf{X}^+\mathbf{X}\mathbf{X}^+ = \mathbf{X}^+$ and $\mathbf{X}\mathbf{X}^+\mathbf{X} = \mathbf{X}$; this is part of the definition of the pseudoinverse, along with the requirements that $(\mathbf{X}\mathbf{X}^+)^\mathsf{T} = \mathbf{X}\mathbf{X}^+$ and $(\mathbf{X}^+\mathbf{X})^\mathsf{T} = \mathbf{X}^+\mathbf{X}$.

To understand the final property, consider the linear system

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$$

Let's multiply both sizes by the matrix $\mathbf{X}^\mathsf{T}$.

$$\mathbf{X}^\mathsf{T}\mathbf{y} = \mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\beta}$$

We know that the matrix $\mathbf{X}$ is not square; however, the matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$ is always square. Since $\mathbf{X}^\mathsf{T}\mathbf{X}$ is square, it is invertible provided it is full rank, which is guaranteed if $\mathbf{X}$ has full column rank (see §7.1.1). Assuming $(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}$ exists, let's multiply both sides of our equation by it.

If matrix $\mathbf{X}$ has m rows and n columns, the matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$ has n rows and n columns.

$$(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\beta}$$

Look carefully at the righthand side. We have the matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$ multiplied by its inverse, $(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}$. This is equal to the identity matrix, so

$$(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y} = \boldsymbol{\beta}$$

We have solved the system $\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$ for the vector $\boldsymbol{\beta}$, so the quantity $(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}$ on the lefthand side must be the pseudoinverse of the matrix $\mathbf{X}$.

Let's use pseudoinversion to solve our example model:

$$\begin{pmatrix} -0.05 \\ 0.40 \\ 0.66 \\ 0.65 \\ 1.12 \end{pmatrix} = \begin{pmatrix} 1 & 0.07 \\ 1 & 0.16 \\ 1 & 0.48 \\ 1 & 0.68 \\ 1 & 0.83 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \end{pmatrix}$$

Using MATLAB's `pinv` function we can calculate the pseudoinverse of the model matrix

$$\mathbf{X}^+ = \begin{pmatrix} 0.59 & 0.50 & 0.16 & -0.046 & -0.20 \\ -0.88 & -0.67 & 0.084 & 0.55 & 0.91 \end{pmatrix}$$

and find the least squares estimates for the parameters

$$\boldsymbol{\beta} = \mathbf{X}^+\mathbf{y} = \begin{pmatrix} 0.59 & 0.50 & 0.16 & -0.046 & -0.20 \\ -0.88 & -0.67 & 0.084 & 0.55 & 0.91 \end{pmatrix} \begin{pmatrix} -0.05 \\ 0.40 \\ 0.66 \\ 0.65 \\ 1.12 \end{pmatrix} = \begin{pmatrix} 0.020 \\ 1.21 \end{pmatrix}$$

Again, we see that $\beta_0 = 0.020$ and $\beta_1 = 1.21$.

8.5.1 Calculating the Pseudoinverse

Our new formula for the pseudoinverse ($\mathbf{X}^+ = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}$) gives us intuition about solving nonsquare linear systems — we are actually solving a related system involving the special matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$. This form of the pseudoinverse requires calculating a matrix inverse, which we have all sworn never to do except in dire situations. The function `pinv` in MATLAB uses a matrix decomposition method to find pseudoinverses, which we will discuss in section 19.3.2. Matrix decomposition methods also work on rank deficient matrices since they do not require a true matrix inversion.

8.6 Dimensions of the Model Matrix

The pseudoinverse of $\mathbf{X}$ is part of the least squares solution for the linear model $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$. Each row in $\mathbf{X}$ is an observation and each column corresponds to an unknown parameter. If $\mathbf{X}$ is square, we have one observation per parameter. We know that the pseudoinverse of a square, invertible matrix is identical to the ordinary inverse. We also know that linear systems with square coefficient matrices have a unique solution. There is no room to find a solution that minimizes the loss when there is only a single unique solution. We can fit all of the parameters, but as we will see later, we have no information about how well we did minimizing the loss.

If we have more observations than parameters ($\mathbf{X}$ has more rows than columns), the extra information in the observations can be used to estimate how well our solution minimizes the quadratic loss. The extra degrees of freedom can quantify our confidence in the model.

Finally, our system is *underdetermined* if we have fewer observations than parameters. The search space for parameters is simply too large, and we often cannot find a meaningful solution. Fitting these models requires special tools that we will discuss in Chapter 14.

Chapter 9

Building Regression Models

In Chapter 8 we outlined a framework for fitting linear models to data. Linear models are enormously flexible and can be applied to many problems in science and engineering. In this chapter we discuss how to formulate, solve, and interpret several types of linear models. The accompanying Matlab workbook describes how to build and solve linear models using matrices and a formula-based interface.

A few notes on notation before we begin. Linear models can be expressed using either standard algebra or vector algebra. Consider a model with two predictor variables (inputs). We can write this model as

$$y_i = \beta_1 x_{1,i} + \beta_2 x_{2,i} + \epsilon_i$$

or using vector notation

$$\mathbf{y} = \beta_1 \mathbf{x}_1 + \beta_2 \mathbf{x}_2 + \boldsymbol{\epsilon}.$$

We prefer the latter form since it is simpler and it emphasizes that each predictor variable is a vector. These vectors are collected into the model matrix, which is pseudoinverted to solve the linear model. Notice how we've dropped the "pred" and "true" labels from our equations. In this chapter it should be clear that the model is always fit using the true, measured values.

9.1 The Intercept

The model's intercept is the only parameter not associated with an input. For the linear model

$$\mathbf{y} = \beta_0 + \beta_1 \mathbf{x}_1 + \beta_2 \mathbf{x}_2 + \boldsymbol{\epsilon}$$

the coefficient β_i is associated with input $\mathbf{x}_i$, so by convention we call the intercept β_0 since it has no associated input. The above model is a slight abuse of notation. The lefthand side ($\mathbf{y}$) is a vector, and all the non-intercept terms on the righthand side ($\beta_1\mathbf{x}_1$, $\beta_2\mathbf{x}_2$, and $\boldsymbol{\epsilon}$) are also vectors. The intercept β_0 is a scalar, and we have not defined an addition operator between scalars and vectors. When writing a linear model, we assume that the intercept multiplies an implicit vector of ones, so the real model is

$$\mathbf{y} = \beta_0\mathbf{1} + \beta_1\mathbf{x}_1 + \beta_2\mathbf{x}_2 + \boldsymbol{\epsilon}.$$

Although we usually omit the vector $\mathbf{1}$, it is a helpful reminder that models with an intercept have a column of ones in their model matrix. Rewriting the above equation in vector notation gives

Be careful to distinguish between $\mathbf{1}$, a vector of ones, and $\mathbf{I}$, the identity matrix.

$$\mathbf{y} = \begin{pmatrix}\mathbf{1} & \mathbf{x}_1 & \mathbf{x}_2\end{pmatrix}\begin{pmatrix}\beta_0\\ \beta_1\\ \beta_2\end{pmatrix} + \boldsymbol{\epsilon}.$$

The model matrix for this model is $\mathbf{X} = \begin{pmatrix}\mathbf{1} & \mathbf{x}_1 & \mathbf{x}_2\end{pmatrix}$. If we wanted to fit a model without an intercept, we would write

$$\mathbf{y} = \begin{pmatrix}\mathbf{x}_1 & \mathbf{x}_2\end{pmatrix}\begin{pmatrix}\beta_1\\ \beta_2\end{pmatrix} + \boldsymbol{\epsilon}$$

and the model matrix would be $\mathbf{X} = \begin{pmatrix}\mathbf{x}_1 & \mathbf{x}_2\end{pmatrix}$.

When all of the input variables $\mathbf{x}_i$ are zero, the model's output is the value of the intercept ($y^{\text{pred}} = \beta_0$). Thus, whether or not to include an intercept in your model depends on the output you would expect when all inputs are zero. For example, imagine you are building a model that predicts the height of a plant based on the number of hours of sunlight it receives. A plant that never sees the sun should not grow, so it would be reasonable to exclude an intercept from the model. If instead you build a model that relates plant height to amount of fertilizer added, you would include an intercept since a plant that receives no fertilizer could still grow with only the nutrients in the soil. Most times your models will include an intercept, and software packages like MATLAB add them by default. However, you should consider if the intercept is needed and if it is reasonable for the output of the model to be nonzero when all of the input variables are zero.

9.2 Analyzing Models

There are two reasons to build models. The first is *prediction* — estimating a new output for inputs that were not included in the training set. For example, a set of clinical measures (blood pressure, resting pulse, white cell count) could be used to train a model that predicts a person's risk of heart attack. After training is finished, we can use the model for prediction by using a new person's clinical measures as inputs.

The second reason to build a model is *inference*. Inference focuses on how a model makes its predictions. For a heart attack model, we can ask which of the clinical measures are important for determining risk, or how much changing an input would raise or lower the model's prediction. It may surprise you how often we build models that are only used for inference and not for prediction. You may have seen studies that link coffee consumption to blood pressure. Such studies are analyzed by building a model that predicts blood pressure based on the number of cups of coffee consumed. This model has low predictive value; few people would need to predict what their blood pressure will be if they drank a certain amount of coffee. But the model has inferential value. Examining the parameters of the model can tell us how large of an effect coffee has on blood pressure and if this effect is statistically significant.

9.2.1 Prediction Intervals

The outputs of a model are only predictions. They are never exactly correct, and we would like some estimate of their accuracy. We can use our training data to assess our model's predictive power. First, we fit the model by finding parameter estimates that minimize the loss function. Model fitting uses the observed inputs and outputs ($\mathbf{x}^{\text{true}}$ and $\mathbf{y}^{\text{true}}$). Next, we feed the training inputs $\mathbf{x}^{\text{true}}$ back into the model to calculate the predicted output $\mathbf{y}^{\text{pred}}$. If the model fit the training data exactly, the predicted outputs $\mathbf{y}^{\text{pred}}$ would perfectly match the true outputs $\mathbf{y}^{\text{true}}$. In almost all cases, the predicted and true outputs will disagree, and we can use the discrepancy to estimate the model's accuracy.

A common measure of accuracy is the *root-mean-squared-error*, or RMSE. If the training data included n observations, then the model's RMSE is

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i^{\text{pred}} - y_i^{\text{true}}\right)^2}$$

The RMSE formula is best understood from the inside out. The squared error of a single prediction is $(y_i^{\text{pred}} - y_i^{\text{true}})^2$. Summing these errors and dividing by n gives the mean squared error. Unfortunately, the mean squared error is difficult to interpret, in part because the units of this error are the output's units squared. It is easiest if we transform the mean squared error back into output's units by taking

We prefer the term "loss" rather than "error" when training models, but error makes sense when measuring a model's accuracy.

the square root. Hence, the RMSE is the square root of the mean of the squared errors.

The RMSE is an estimate of the standard deviation of the prediction errors. When reporting a prediction when can include the RMSE to give the reader a feel for the model's accuracy. If a model predicts a patient's resting pulse rate as 68 bpm and the model's RMSE is 12 bpm, then we should report the prediction as 68 ± 12 bpm. It is often assumed that the model's prediction errors are normally distributed, so the RMSE can be used to estimate a *prediction interval*. The 95% prediction interval spans twice the RMSE on either side of the prediction. For our pulse example, the 95% prediction interval is $[68-2\times12,\ 68+2\times12] = [44, 92]$ bpm. Remember that if we transformed the output of a model (see §9.4), we need to undo the transformation for both the model prediction and the RMSE to get back to the original units.

9.2.2 Effect Sizes and Significance

Let's switch gears from prediction to inference. The goal of inference is to understand how the inputs relate to the output. You can think of model building as a multivariable hypothesis test. After collecting data, we construct a predictive model based on a set of inputs. We might discover that some of the inputs may not be useful for predicting the output, so model fitting in essence tests the strength of the relationship between each input and the output.

Model inference involves asking two questions about each of the models inputs.

1. How large of an effect does this input have on the output?
2. How confident are we in our estimate of this effect?

The first question concerns the *effect size* of the input. The effect size is another name for the coefficient that we estimate for the input. Consider the two input model

$$\mathbf{y} = 1.2 - 3.6\mathbf{x}_1 + 0.8\mathbf{x}_2 + \epsilon.$$

The effect size for input $\mathbf{x}_1$ is -3.6, and the effect size of input $\mathbf{x}_2$ is 0.8. The effect size quantifies the sensitivity of the output with respect to the input. Increasing $\mathbf{x}_1$ by one will *decrease* the output $\mathbf{y}$ by 3.6 (since the effect size is negative). Increasing $\mathbf{x}_2$ by one will *increase* the output by 0.8.

Increasing a variable by one is commonly called a "unit increase". This terminology is unrelated to the actual units of the variable, e.g. kilograms, seconds, etc.

Effect sizes have units. If the previous model predicted pulse rate in bpm and the input $\mathbf{x}_1$ was a person's age in years, the effect size would be -3.6 bpm/year. It is important to include any units when reporting effect sizes.

The effect size of an input answers the first inferential question ("how large of an effect does an input have on the output?"). Models estimate effects using inherently noisy data, so our estimates are never exact. Most statistical modeling packages will report a standard error and a p-value for each effect size. The standard error can be used to construct a confidence interval for the effect size. For example, the 95% confidence interval spans two standard errors on each size of the effect size. Often we want to know if an effect size is significantly different from zero. If an effect size is indistinguishable from zero (i.e. if the confidence interval includes zero), then we cannot reject the idea that the observed effect size is simply an artifact of the uncertainty in our data. Said another way, if an effect size is indistinguishable from zero, we should not be surprised if we refit the data with a new set of observations (of the same size) and find that the effect has "disappeared".

The t-test is a common method for testing if an effect size can be distinguished from zero. The p-value reported for an effect size is the p-value from a t-test. A threshold of $p < 0.05$ is frequently used to separate "significant" from "not significant" effects. If the p-value exceeds our threshold, we are not confident that the effect size is nonzero.

Be careful to distinguish between an effect size and its significance. The p-value is only related to the precision of our estimate; it has no bearing on the magnitude of the effect. A very small p-value indicates that an effect size is statistically distinguishable from zero, but the *practical significance* of the effect could be small. As an interesting example, consider a study released by the online dating site eHarmony (Cacioppo, et al., *Proc Nat Acad Sci*, 2013). The study reports a statistically significant increase in marital satisfaction among couple who met online vs. by other venues. While the results were statistically significant, the actual increase in marital satisfaction was only 2.9%. Even though the means of the two groups differed by less than 3%, the enormous sample size (19,131 couples) made the result statistically — but not practically — significant.

9.2.3 Degrees of Freedom

A model's parameters must be estimated from the observed data. We know from Chapter 7 that a linear system requires the information from one observation to estimate each unknown. Let's assume we use m observations to fit a model with n unknown parameters. If $m = n$, we have just enough information to estimate each parameter; however, we have no information left over to assess the accuracy of our predictions or the effect sizes. If we have more observations than unknown parameters ($m > n$), then we can use the remaining observations to build confidence intervals. We call the number of "extra" observations the *degrees of freedom*. A model with n parameters fit to m observations has $m - n$ degrees of freedom. Note that the intercept is an unknown parameter, so it should be included the parameter count. We need at least one degree of freedom to estimate prediction intervals or the significance of effect sizes. As the degrees of freedom increase, the RMSE and the standard errors of the effect sizes decrease.

9.3 Curvilinear Models

So far we've focused on models that are linear combinations of a set of inputs. It is important to remember that the parameters, not the inputs, are the unknowns in our models. We can apply any transformation we want to the inputs and retain a linear model. Consider the cubic polynomial model

$$\mathbf{y} = \beta_0 + \beta_1\mathbf{x} + \beta_2\mathbf{x}^2 + \beta_3\mathbf{x}^3 + \boldsymbol{\epsilon}.$$

We define vector exponentiation as an elementwise operation, so

$$\mathbf{x}^2 = \begin{pmatrix} x_1^2 \\ x_2^2 \\ \vdots \\ x_n^2 \end{pmatrix}.$$

This model is still linear with respect to its parameters, as we see when we rewrite it in matrix form.

$$\mathbf{y} = \begin{pmatrix} \mathbf{1} & \mathbf{x} & \mathbf{x}^2 & \mathbf{x}^3 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} + \boldsymbol{\epsilon}$$

This model has only a single input — the vector $\mathbf{x}$. We've created additional *features* for the model by squaring and cubing the entries in $\mathbf{x}$. Each of the transformed features appears as a separate column in the model matrix.

We can fit any polynomial using linear regression by transforming the input. These transformations are usually hidden "under the hood" in spreadsheet programs that let you perform polynomial regression. Spreadsheets allow their users to specify the degree of the polynomial. When a user clicks to increase the degree, the spreadsheet adds a new column to the model matrix and refits the model.

Any transformation, not just exponentiation, can be applied to the inputs of a model. Logarithms, square roots, and mean-centering (subtracting the mean of $\mathbf{x}$ from every entry) are common transformations. The outputs $\mathbf{y}$ can also be transformed. A model that includes transformed inputs or outputs is said to be *curvilinear* since it remains linear with respect to the parameters but the input/output relationship is no longer "straight".

9.4 Linearizing Models

We are able to fit curvilinear models using linear algebra because the models remained linear with respect to the parameters. Some models do not appear to be linear but can be made linear by transformation. For example, bacteria grow exponentially, so their growth can be describe by the model

$$N(t) = N_0 e^{\mu t}$$

where $N(t)$ is the number of bacteria at time t, N_0 is the initial number of bacteria, and μ is the growth rate. The parameters in this model are N_0 and μ, and the exponential growth model is not linear with respect to μ. Let's transform the model by taking the natural logarithm of both sides.

$$\begin{aligned}\log(N(t)) &= \log(N_0 e^{\mu t}) \\ &= \log(N_0) + \mu t\end{aligned}$$

Now let's make a few substitutions. Our output variable is $y = \log(N(t))$. The values $N(t)$ are all known, so we simply transform them with the natural logarithm before fitting the model. We will also set $\beta_0 = \log(N_0)$ and $\beta_1 = \mu$, making our final linear model

$$\mathbf{y} = \beta_0 + \beta_1 \mathbf{t} + \boldsymbol{\epsilon}.$$

We can use linear regression to estimate the parameters β_0 and β_1. These parameters can be transformed back into estimates of N_0 and μ:

$$\beta_0 = \log(N_0) \Rightarrow N_0 = e^{\beta_0}$$

$$\mu = \beta_1$$

The Michaelis-Menten equation is another nonlinear function that can be linearized. Recall that the velocity v of a reaction is a function of substrate concentration $[S]$ and two parameters, $V_{\max}$ and K_m.

$$v = \frac{V_{\max}[S]}{K_m + [S]}$$

This equation is nonlinear with respect to the parameters $V_{\max}$ and K_m, but it can be linearized by inverting both sides.

This linearization of the Michaelis-Menten equations is called Lineweaver-Burk or double reciprocal method.

$$\begin{aligned}\frac{1}{v} &= \frac{K_m + [S]}{V_{\max}[S]} \\ &= \frac{K_m}{V_{\max}[S]} + \frac{[S]}{V_{\max}[S]} \\ &= \frac{K_m}{V_{\max}}\frac{1}{[S]} + \frac{1}{V_{\max}}\end{aligned}$$

Using $1/[S]$ as the input $\mathbf{x}$ and $1/v$ as the output $\mathbf{y}$, the two parameters of the linear model are $\beta_0 = 1/V_{\max}$ and $\beta_1 = K_m/V_{\max}$.

Not every nonlinear model can be transformed into a linear one. Logarithmic transformations sometimes work for multiplicative models and models with parameters as exponents. Linearizing models allows us to find parameter values using linear regression, but there is a downside. The linearization can skew the distribution of our dataset and amplify measurement errors. For the linearized Michaelis-Menten equation, the output variable is $1/v$, the inverse of the reaction velocity. When the velocity v is very small, the transformed variable $1/v$ becomes very large. Any uncertainty in our velocity measurements will be amplified when v is small. An alternative approach is to avoid linearization and fit parameters directly to the nonlinear model. We will discuss nonlinear fitting methods in Part II.

9.5 Interactions

So far the inputs to our models are additive. Consider again the two input linear model

$$\mathbf{y} = 1.2 - 3.6\mathbf{x}_1 + 0.8\mathbf{x}_2 + \boldsymbol{\epsilon}.$$

For every unit increase in $\mathbf{x}_1$, the output $\mathbf{y}$ decreases by 3.6, regardless of the value of the other input $\mathbf{x}_2$. In modeling terms, we say there is no *interaction* between inputs $\mathbf{x}_1$ and $\mathbf{x}_2$. If we believe that the inputs do interact, that is, if the effect of changing one input depends on the value of the other input, then we can add a term to describe this interaction. The term we add is the product of the two inputs, and this term receives its own parameter. For example, a standard two-input linear model (without interaction) is

$$\mathbf{y} = \beta_0 + \beta_1\mathbf{x}_1 + \beta_2\mathbf{x}_2 + \boldsymbol{\epsilon}.$$

A similar model with an interaction term between $\mathbf{x}_1$ and $\mathbf{x}_2$ is

$$\mathbf{y} = \beta_0 + \beta_1\mathbf{x}_1 + \beta_2\mathbf{x}_2 + \beta_{12}\mathbf{x}_1{:}\mathbf{x}_2 + \boldsymbol{\epsilon}.$$

By convention we call the interaction parameter β_{12} since it measures the interaction effect between inputs $\mathbf{x}_1$ and $\mathbf{x}_2$. (It is read "1-2", not as the number "12".) We've used the colon to represent elementwise multiplication between the vectors $\mathbf{x}_1$ and $\mathbf{x}_2$, as this is the syntax used to specify linear models in many software packages. Adding interaction terms does not violate the linearity of the model. Remember that the inputs $\mathbf{x}_i$ are known, so multiplying them together yields yet another known quantity.

The elementwise product of two vectors **a** and **b** is

$$\mathbf{a}{:}\mathbf{b} = \begin{pmatrix} a_1 b_1 \\ a_2 b_2 \\ \vdots \\ a_n b_n \end{pmatrix}.$$

The interaction effect β_{12} quantifies the effects of the inputs that cannot be explained by either input alone. Notice that our interaction model retains the independent terms $\beta_1\mathbf{x}_1$ and $\beta_2\mathbf{x}_2$. The interaction term $\beta_{12}\mathbf{x}_1{:}\mathbf{x}_2$ only describes the "above and beyond" effects. If the inputs $\mathbf{x}_1$ and $\mathbf{x}_2$ are strictly additive, then the estimate of the interaction effect β_{12} will be zero. In biomedical terms, you can think of the interaction as the "synergy" between the inputs.

You might be wondering why the interaction between two variables is measured by their product, rather than some other function. There are two explanations, and you are free to choose the one you like best.

1. The interaction depends on both inputs, so it should not have an effect when either input is missing (zero). The term $\beta_{12}\mathbf{x}_1{:}\mathbf{x}_2$ is the simplest expression that is zero when either $\mathbf{x}_1$ or $\mathbf{x}_2$ is zero.

2. Alternatively, we could assume that the effect size of input $\mathbf{x}_1$ depends on the value of $\mathbf{x}_2$. We could write a model

   $$y = (\beta_1 + \beta_{12}\mathbf{x}_2){:}\mathbf{x}_1 + \beta_2\mathbf{x}_2$$

   where the effect size of x_1 is not a single parameter but instead an expression that depends on $\mathbf{x}_2$. The coefficient β_{12} adjusts the coefficient of $\mathbf{x}_1$ for each unit change in $\mathbf{x}_2$. If we distribute $\mathbf{x}_1$ into its coefficient, we see that this model is identical to our standard interaction model $y = \beta_1\mathbf{x}_1 + \beta_2\mathbf{x}_2 + \beta_{12}\mathbf{x}_1{:}\mathbf{x}_2$.

Linear models can have higher-order interactions, like the following model with all possible two- and three-way interactions.

$$\begin{aligned}\mathbf{y} = \beta_0 &+ \beta_1\mathbf{x}_1 + \beta_2\mathbf{x}_2 + \beta_3\mathbf{x}_3\\ &+ \beta_{12}\mathbf{x}_1{:}\mathbf{x}_2 + \beta_{13}\mathbf{x}_1{:}\mathbf{x}_3 + \beta_{23}\mathbf{x}_2{:}\mathbf{x}_3\\ &+ \beta_{123}\mathbf{x}_1{:}\mathbf{x}_2{:}\mathbf{x}_3 + \boldsymbol{\epsilon}\end{aligned}$$

A model with n inputs has 2^n possible parameters (including the intercept and main effects) if all interactions are considered. Fortunately, there is rarely a need for higher-order interactions. A three-way interaction term measures the effects of the three inputs that cannot be explained by the main effects or the two-way interactions among the three inputs. It is rare to see three inputs interact in a way that cannot be explained by pairwise interactions. In the statistical literature, the rarity of significant higher-order interactions is called the *hierarchical ordering principle*.

Part II

Nonlinear Systems

Part I of this book described methods for solving linear systems. These methods are definite — the solvability theorems tell us if a system is solvable and exactly how many solutions exist. The tools for linear systems are also constructive. If solutions exist, we have deterministic methods to find them.

In Part II we turn our attention to nonlinear systems. The nonlinearities remove the luxuries we encountered with linear systems. We will not know if a nonlinear system is solvable or how many solutions exist. Even when a solution exists, we will be forced to rely on iterative or stochastic methods to search for it.

There are nonlinear systems that are exempt from the above problems. The linear least-squares problems of Chapter 8 are nonlinear (quadratic), but we used the pseudoinverse to find a unique solution. In Chapter 11 we will see that the linear least-squares problem belongs to a special class of nonlinear problems because it is convex. Convex problems can be solved with relative ease, and learning to identify and exploit convexity is a powerful tool for nonlinear systems.

We will focus on two nonlinear problems. The first is the *root finding* problem: For a system of nonlinear equations $\mathbf{g}(\mathbf{x})$, what are the values of the vector $\mathbf{x}$ such that $\mathbf{g}(\mathbf{x}) = \mathbf{0}$? The second problem is the *optimization* problem: Find a vector $\mathbf{x}$ that minimizes the scalar function $f(\mathbf{x})$. These two problems are related, and algorithms that solve one problem can be used to solve the other. For example, consider a continuously differentiable function $f(\mathbf{x})$. If f has a minimum, it occurs when the gradient of f is equal to the zero vector. Minimizing the function f is equivalent to find a vector $\mathbf{x}$ such that $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ when $\mathbf{g}$ is the gradient of f. Similarly, imagine we want to find a zero of the nonlinear function $\mathbf{g}(\mathbf{x})$. If we define $f(\mathbf{x}) = \|\mathbf{g}(\mathbf{x})\|$, then the zero of the function $\mathbf{g}$ corresponds to the point at the minimum of the function f.

Solving nonlinear systems is more of an “art” than solving linear systems. We will learn several strategies that work well for some problems but not for others. There is no single best method for solving nonlinear problems, and we will focus on the strengths and weaknesses of each technique. In practice, you will learn to try methods that are inspired by the features of each problem.

Chapter 10

Root Finding

We've seen multiple methods for solving linear systems of equations. In this chapter we develop a method to solve nonlinear systems of equations using linear algebra. We begin with Newton's method for finding the roots of a single nonlinear equation. Then we generalize the method to systems of equations using a matrix formalism.

10.1 Nonlinear Functions

A nonlinear function is, simply put, a function that fails the tests for linearity. You might have been surprised that the affine function $g(x) = ax + b$ was nonlinear. The functions $g(x) = \cos x$, $g(x) = x^2$, and $g(x) = \log x$ are all nonlinear with respect to the independent variable x.

By convention we write nonlinear functions in the form

$$g(x) = 0$$

This convention is not a limitation, as any nonlinear function with a nonzero right hand side can be rewritten by moving the right hand terms to the left side. Writing nonlinear functions in this way lets us solve the function by identifying values where the function equals zero, i.e. by finding the *roots* of the function. For example, the equation

$$(x-1)^3 = 8$$

has a unique solution when $x = 3$. We can rewrite this equation as the function

$$g(x) = (x-1)^3 - 8 = 0$$

Notice that the function $g(x)$ has a root when $x = 3$, which is also the solution to the equation $(x-1)^3 = 8$.

Linear systems have exactly zero, one, or infinitely many solutions. By contrast, nonlinear systems can have any number of solutions. The function $g(x) = x^2 - 4$ has two roots: $x = 2$ and $x = -2$. Unlike linear systems, there is no grand solvability theorem for nonlinear systems. Except in special cases (for example, polynomials), we cannot tell *a priori* how many unique solutions exist for a nonlinear equation. Even when we know a solution exists, we do not have a general procedure like Gaussian elimination for finding solutions to nonlinear equations. Instead, we often rely on numerical techniques to find *some* of the roots of nonlinear functions.

10.2 Newton's Method

Given a function $g(x)$, how do we find its roots? One powerful method builds on an observation regarding the tangent lines of $g(x)$ near its roots. Imagine we are at a point x_0 that is near a root. The tangent line of $g(x)$ at the point x_0 will itself have a root that is closer to the root of $g(x)$. Let's call this new point x_1.

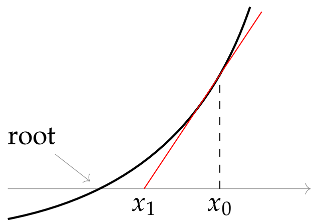


Figure 10.1: If a point x_0 is close to the root of a function (black), the tangent line (red) intersects the horizontal axis at a point x_1 that is closer to the root.

If we draw another tangent line for g at x_1, we see that the root of the tangent line is again closer to the root of g. We can repeat this procedure again and again, each time moving closer to the root of g. Rather than solve the nonlinear function g, we only need to solve a series of affine equations describing the tangent line at each iteration.

Let's formalize the above procedure. The starting point x_0, the values of g and its derivative g', and the root x_1 of the tangent line are related by

$$g'(x_0) = \frac{g(x_0)}{x_0 - x_1}$$

You can interpret this formula as "the slope of the tangent line at x_0 ($g'(x_0)$) is equal to the height of the function at x_0 ($g(x_0)$) divided by the distance between x_0 and x_1." Rearranging, we can find the root of the tangent line based on values at our current point.

In other words, the slope of the tangent line $g'(x_0)$ is its rise $g(x_0)$ divided by its run $(x_0 - x_1)$.

$$x_1 = x_0 - \frac{g(x_0)}{g'(x_0)}$$

Now we know the location of x_1, a point closer to the root of the original function g. We can apply the same procedure starting at x_1 to find a closer point x_2, and so on.

Newton published a very limited version of the method that bears his name. British mathematician Thomas Simpson was the first to apply the technique to general systems of nonlinear equations. He also noted connections between nonlinear systems and optimization.

$$
\begin{aligned}
x_2 &= x_1 - \frac{g(x_1)}{g'(x_1)} \\
x_3 &= x_2 - \frac{g(x_2)}{g'(x_2)} \\
&\vdots \\
x_{n+1} &= x_n - \frac{g(x_n)}{g'(x_n)}
\end{aligned}
$$

10.3 Convergence of Newton's Method

Let's find a root for the equation

$$g(x) = (x-4)^3 - 2x$$

By plotting the function, we see there is a root somewhere between $x = 6$ and $x = 6.5$. We can use Newton's Method to find a more precise estimate of the root. We first calculate the derivative

$$g'(x) = 3(x-4)^2 - 2$$

Let's choose our initial guess to be $x_0 = 6.0$. We're ready to calculate x_1.

$$
\begin{aligned}
x_1 &= x_0 - \frac{g(x_0)}{g'(x_0)} \\
&= x_0 - \frac{(x_0-4)^3 - 2x_0}{3(x_0-4)^2 - 2} \\
&= 6.0 - \frac{(6.0-4)^3 - 2(6.0)}{3(6.0-4)^2 - 2} \\
&= 6.4
\end{aligned}
$$

We can check if we've found a root by evaluating $g(x_1)$. If x_1 is a root, $g(x_1)$ should equal zero.

$$g(x_1) = g(6.4) = 1.024 \neq 0$$

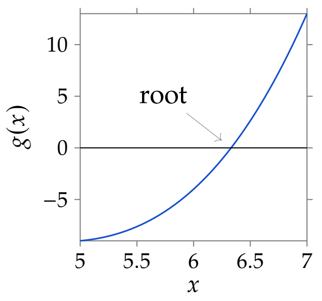


Figure 10.2: The function $g(x) = (x-4)^3 - 2x$ has a root between $x = 6$ and $x = 6.5$.

We haven't arrived at a root yet. Let's try another iteration of Newton's method to find a second guess (x_2) using x_1.

$$\begin{aligned} x_2 &= x_1 - \frac{g(x_1)}{g'(x_1)} \\ &= x_1 - \frac{(x_1-4)^3 - 2x_1}{3(x_1-4)^2 - 2} \\ &= 6.4 - \frac{(6.4-4)^3 - 2(6.4)}{3(6.4-4)^2 - 2} \\ &= 6.332984293 \end{aligned}$$

The new value x_2 is closer to being a root: $g(6.332984293) = 0.03203498$. We can always move closer using more iterations as shown in the following table.

When studying numerical methods we will extend our answers far beyond the number of significant figures. As engineers we later trim or *truncate* our answers to an appropriate number of significant figures based on the uncertainty in the system.

i	x_i	$g(x_i)$
0	6	-4
1	6.4	1.024
2	6.332984293	0.032034981
3	6.330748532	0.000034974
4	6.330746086	4.18421×10^{-11}

Newton's method converges quadratically once the x_i are close to the actual root. "Close" is not well defined and varies with each function. If an initial guess is far from the true root, Newton's method can either 1.) converge slowly until it becomes close enough for quadratic converge to kick in, or 2.) not converge at all. If Newton's method is converging slowly or diverges, you should try a different initial guess.

The quadratic convergence stems from our use of a linear approximation for the function, leaving a residual bounded by the quadratic terms.

10.4 Multivariable Functions

Newton's method works well for nonlinear functions of a single variable. We use a variant of Newton's method to solve multivariable functions. Multivariable functions accept a vector of inputs and produce a vector of outputs. We write the names of multivariable functions using bold, non-italicized font — $\mathbf{g}(\mathbf{x})$ — to remind us that a multivariable functions return a vector of outputs.

Multivariable functions are also called *multivariate* or *vector-valued* functions.

We're already familiar with linear multivariable functions like $\mathbf{g}(\mathbf{x}) = \mathbf{A}\mathbf{x}$. This function accepts a vector of inputs ($\mathbf{x}$) and returns another vector of outputs ($\mathbf{A}\mathbf{x}$). We can also define nonlinear multivariable functions. An example with three inputs and three outputs is

$$\mathbf{g}(\mathbf{x}) = \begin{pmatrix} x_1 - x_3 \\ x_3^2 + 2x_2 \\ \cos x_1 \end{pmatrix}$$

If $\mathbf{x} = \begin{pmatrix} 0 \\ -1 \\ 2 \end{pmatrix}$, then

$$\mathbf{g}(\mathbf{x}) = \begin{pmatrix} 0 - 2 \\ 2^2 + 2(-1) \\ \cos 0 \end{pmatrix} = \begin{pmatrix} -2 \\ 2 \\ 1 \end{pmatrix}$$

It's sometimes convenient to talk individually about the entries in the nonlinear function. We can write a multivariable function using the following notation

We use lowercase and italicized font (g_i) when referencing individual entries in a multivariable function since each entry produces only a single output.

$$\mathbf{g}(\mathbf{x}) = \begin{pmatrix} g_1(x_1, x_2, \ldots, x_n) \\ g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ g_n(x_1, x_2, \ldots, x_n) \end{pmatrix}$$

For the example above, $g_1 = x_1 - x_3$; $g_2 = x_3^2 + 2x_2$; and $g_3 = \cos x_1$.

10.5 The Jacobian Matrix

For functions of a single variable, Netwon's method uses the derivative to construct a linear approximation. The multivariable analog of the derivative is matrix of partial derivatives called the *Jacobian*, which we write as $\mathbf{J}(\mathbf{x})$.

The Jacobian is named after German mathematician Carl Gustav Jacob Jacobi. I assume it is based on his last name, or possibly his second-to-last name.

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_n}{\partial x_1} & \frac{\partial g_n}{\partial x_2} & \cdots & \frac{\partial g_n}{\partial x_n} \end{pmatrix}$$

The (i,j)th entry in the Jacobian is the partial derivative of the ith function with respect to the jth variable. If a multivariable function has n inputs and n outputs, its Jacobian is a square $n \times n$ matrix.

Let's compute the Jacobian for the function $\mathbf{g}(\mathbf{x}) = \begin{pmatrix} x_1 - x_3 \\ x_3^2 + 2x_2 \\ \cos x_1 \end{pmatrix}$.

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} \frac{\partial}{\partial x_1}(x_1 - x_3) & \frac{\partial}{\partial x_2}(x_1 - x_3) & \frac{\partial}{\partial x_3}(x_1 - x_3) \\ \frac{\partial}{\partial x_1}\left(x_3^2 + 2x_2\right) & \frac{\partial}{\partial x_2}\left(x_3^2 + 2x_2\right) & \frac{\partial}{\partial x_3}\left(x_3^2 + 2x_2\right) \\ \frac{\partial}{\partial x_1}(\cos x_1) & \frac{\partial}{\partial x_2}(\cos x_1) & \frac{\partial}{\partial x_3}(\cos x_1) \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & -1 \\ 0 & 2 & 2x_3 \\ -\sin x_1 & 0 & 0 \end{pmatrix}$$

10.6 Multivariable Newton's Method

For functions of a single variable, Newton's method iterates with the formula

$$x_{i+1} = x_i - \frac{g(x_i)}{g'(x_i)}$$

Using a multivariable linear approximation, we can define the multivariable analogue of Newton's method.

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{J}^{-1}(\mathbf{x}_i)\mathbf{g}(\mathbf{x}_i)$$

As an example, let's find a root of the function

$$\mathbf{g} = \begin{pmatrix} x_1x_2 - 2 \\ -x_1 + 3x_2 + 1 \end{pmatrix}$$

First we calculate the Jacobian matrix.

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} x_2 & x_1 \\ -1 & 3 \end{pmatrix}$$

Using an initial guess of $\mathbf{x}_0 = \begin{pmatrix} -1 \\ -1 \end{pmatrix}$ we begin iterating.

$$\begin{aligned}
\mathbf{x}_1 &= \mathbf{x}_0 - \mathbf{J}^{-1}(\mathbf{x}_0)\mathbf{g}(\mathbf{x}_0) \\
&= \begin{pmatrix} -1 \\ -1 \end{pmatrix} - \begin{pmatrix} -1 & -1 \\ -1 & 3 \end{pmatrix}^{-1} \begin{pmatrix} (-1)(-1) - 2 \\ -(-1) + 3(-1) + 1 \end{pmatrix} \\
&= \begin{pmatrix} -2 \\ -1 \end{pmatrix}
\end{aligned}$$

Now we use $\mathbf{x}_1$ to find the next guess $\mathbf{x}_2$.

$$\begin{aligned}
\mathbf{x}_2 &= \mathbf{x}_1 - \mathbf{J}^{-1}(\mathbf{x}_1)\mathbf{g}(\mathbf{x}_1) \\
&= \begin{pmatrix} -2 \\ -1 \end{pmatrix} - \begin{pmatrix} -1 & -2 \\ -1 & 3 \end{pmatrix}^{-1} \begin{pmatrix} (-2)(-1) - 2 \\ -(-2) + 3(-1) + 1 \end{pmatrix} \\
&= \begin{pmatrix} -2 \\ -1 \end{pmatrix}
\end{aligned}$$

Our guess $\mathbf{x}_2$ is exactly equal to the previous guess $\mathbf{x}_1$. Since our guess didn't change we are probably at a root. We can check by evaluating $\mathbf{g}(\mathbf{x}_2)$.

$$\mathbf{g}(\mathbf{x}_2) = \begin{pmatrix} (-2)(-1) - 2 \\ -(-2) + 3(-1) + 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Indeed, the vector $\begin{pmatrix} -2 \\ -1 \end{pmatrix}$ is a solution to our equation.

Nonlinear systems often have many solutions. Newton's method converges to the solution nearest the initial guess. If we chose the point $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ as our initial guess, Newton's method on the same function would converge to the root $\mathbf{x} = \begin{pmatrix} 3 \\ 2/3 \end{pmatrix}$ after four iterations.

"Nearest" in the topological sense, i.e. the solution that is down the gradient of the function at the initial guess.

10.7 * Gauss-Newton Method

The multivariate Newton's method assumes that the inputs and outputs of the function $\mathbf{g}$ have the same dimension. If the dimensions disagree, the Jacobian matrix will not be square and its inverse will not be defined. In some cases, we can apply a related method — the Gauss-Newton Method — that uses the pseudoinverse of the nonsquare Jacobian.

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{J}^{+}(\mathbf{x}_i)\mathbf{g}(\mathbf{x}_i)$$

For convergence, we require that the function $\mathbf{g}$ accepts an n-dimensional input vector and outputs an m-dimensional vector, where $m > n$.

10.8 Root Finding with Finite Differences

In all of our examples we have been able to calculate the derivative of the function g (or the Jacobian of multivariate function $\mathbf{g}$) using calculus. This is not always possible. Sometimes the function g is unknown to us or is very complicated. Sometimes g is a simulation that includes random numbers, like a traffic simulator that models random arrivals and departures of cars. In this case we cannot calculate the derivative without knowing what random numbers will appear when the function is later evaluated.

An alternative is to use finite differences to approximate the derivative. Recall from Chapter 5 that the derivative $g'(x)$ can be approximated by

$$g'(x) \approx \frac{g(x + \Delta x) - g(x)}{\Delta x}$$

for some small value Δx. Let's return to an example from earlier in this chapter:

$$g(x) = (x - 4)^3 - 2x$$

We can approximate the derivative at $x = 1$ with $\Delta x = 0.1$.

$$\begin{aligned} g'(1) &\approx \frac{g(1.1) - g(1)}{0.05} \\ &= \frac{(1.1 - 4)^3 - 2(1.1) - (1 - 4)^3 + 2(1)}{0.1} \\ &= 24.0 \end{aligned}$$

The actual value of the derivative at $x = 1$ is

$$\begin{aligned} g'(1) &= 3(1 - 4)^2 - 2 \\ &= 25 \end{aligned}$$

The accuracy of our approximation depends on both the size of the perturbation Δx and on the nonlinearity of the function. Using $\Delta x = 0.01$ puts us closer to the correct value of the derivative ($g'(x) \approx 25$), while a perturbation of $\Delta x = 0.2$ makes the approximation worse ($g'(x) \approx 23$). This is a big problem when the function g is *stochastic*, meaning it depends on random values. Stochastic functions are "noisy" and their outputs always include error. Making the perturbation smaller can amplify the effects of this error, but large perturbations will lead to a poor approximation of the derivative. One solution is to construct our approximation of the derivative using multiple finite difference measurements. Multiple measurements can average out the error, but they require more computation.

In addition to numerical issues, a finite difference approximation can be expensive to evaluate for multivariate functions. To approximate a partial derivative we perturb the vector $\mathbf{x}$ along a single dimension. We can write the perturbation using the Cartesian unit vectors

$$\frac{\partial g}{\partial \mathbf{x}_i} \approx \frac{g(\mathbf{x} + \Delta \hat{\mathbf{e}}_i) - g(\mathbf{x})}{\Delta}$$

where the scalar Δ is the perturbation size. Every entry in the Jacobian must be approximated using a function evaluation with a perturbed input. The Jacobian of an n-dimensional function has n^2 entries, so a finite difference approximation requires n^2 function evaluations.

A more recent solution is a technique called *automatic differentiation*, also known as "autodiff" or "autograd" (which is short for automatic gradient). Automatic differentiation uses specialized software to compute derivatives by tracking the mathematical operations in a function and applying the chain rule. Automatic differentiation is available in many state-of-the-art machine learning packages. It can calculate the true derivative of a function when applied correctly. Many implementations include an option to check the automatic differentiation results using finite differences.

10.9 Practical Considerations

Solving nonlinear equations is an art. Here are some tips.

- Nonlinear equations rarely have a single solution. Solvers try many (hundreds or thousands) of initial guesses to find several solutions. There is no general method for determining the total number of roots for a nonlinear system.

- Software packages like MATLAB's `fsolve` function can find roots with a variety of algorithms. Many techniques find points near roots and use Newton's method to finish the search.

- Software packages often allow users to provide both the function and the Jacobian. Knowing the Jacobian explicitly almost always improves speed and numerical stability. If the user doesn't provide a Jacobian, the software will estimate the Jacobian at every iteration using finite differences.
- Single variable Newton's method requires the function be continuously differentiable. Multivariable functions require the Jacobian be invertible. So-called "gradient free" algorithms are available for functions with poorly behaving, computationally expensive, or discontinuous derivatives.
- The multivariable Newton's method involves inverting the Jacobian, which is computationally expensive. Instead, numerical solvers rearrange the iteration equation:

$$\begin{aligned} \mathbf{x}_{i+1} &= \mathbf{x}_i - \mathbf{J}^{-1}(\mathbf{x}_i)\mathbf{g}(\mathbf{x}_i) \\ \mathbf{J}(\mathbf{x}_i)\mathbf{x}_{i+1} &= \mathbf{J}(\mathbf{x}_i)\mathbf{x}_i - \mathbf{J}(\mathbf{x}_i)\mathbf{J}^{-1}(\mathbf{x}_i)\mathbf{g}(\mathbf{x}_i) \\ \mathbf{J}(\mathbf{x}_i)\left(\mathbf{x}_{i+1} - \mathbf{x}_i\right) &= -\mathbf{g}(\mathbf{x}_i) \end{aligned}$$

  In this form, the solver can use Gaussian elimination on the augmented matrix $[\mathbf{J}(\mathbf{x}_i) \;\; -\mathbf{g}(\mathbf{x}_i)]$ to solve for $\mathbf{x}_{i+1} - \mathbf{x}_i$; adding $\mathbf{x}_i$ gives the new estimate for $\mathbf{x}_{i+1}$.

Chapter 11

Optimization and Convexity

We formulated the least squares method and linear regression as optimization problems. Our goal was to minimize the sum of the squared errors by choosing parameters for the linear model. Optimization problems have enormous utility in data science, and most model fitting techniques are cast as optimizations. In this chapter, we will develop a general framework for describing and solving several classes of optimization problems. We begin by reviewing the fundamentals of optimization. Next, we discuss convexity, a property that greatly simplifies the search for optimal solutions. Finally we derive vector expressions for common geometric constructs and show how linear systems give rise to convex problems.

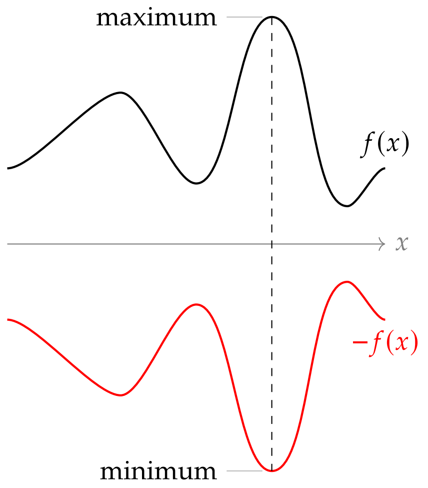


Figure 11.1: The maximum of a function $f(x)$ can be found by minimizing $-f(x)$.

11.1 Optimization

Optimization is the process of minimizing or maximizing a function by selecting values for a set of variables or parameters (called *decision variables*). If we are free to choose any values for the decision variables, the optimization problem is *unconstrained*. If our solutions must obey a set of constraints, the problem is a *constrained optimization*. In constrained optimization, any set of values for the decision variables that satisfies the constraints is called a *feasible solution*. The goal of constrained optimization is to select the "best" feasible solution.

Optimization problems are formulated as either minimizations or maximizations. We don't need to discuss minimization and maximization separately, since minimizing $f(x)$ is equivalent to maximizing $-f(x)$. Any algorithm for minimizing can be used for maximizing by multiplying the objective by -1, and vice versa. For the rest of this chapter, we'll talk about minimizing functions. Keep in mind that everything we discuss can be applied to maximization problems by switching the sign of the objective.

During optimization we search for minima. A minimum can either be *locally* or *globally* minimal. A global minimum is has the smallest objective value of any feasible solution. A local minimum has the smallest objective value for any of the feasible solutions in the surrounding area. The input to a function that yields the minimum is called the *argmin*, since it is the argument to the function that gives the minimum. Similarly, the *argmax* of a function is the input that gives the function's maximum. Consider the function $f(x) = 3 + (x-2)^2$. This function has a single minimum, f(2) = 3. The minimum is 3, while the argmin is $x = 2$, the value of the decision variable at which the minimum occurs. For optimization problems, the minimum (or maximum) is called the *optimal objective value*. The argmin (or argmax) is called the *optimal solution*.

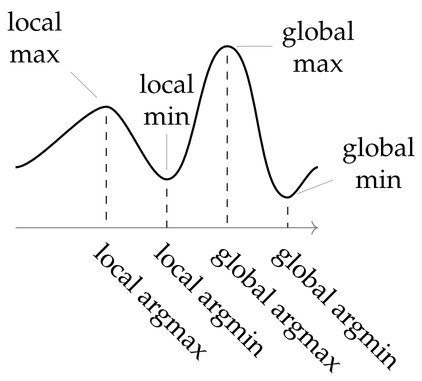


Figure 11.2: Minima and maxima of a function can be local or global.

11.1.1 Unconstrained Optimization

You already know how to solve unconstrained optimization problems in a single variable: set the derivative to the function equal to zero and solve. This method of solution relies on the observation that both maxima and minima occur when the slope of a function is zero. However, it is important to remember that not all roots of the derivative are maxima or minima. Inflection points (where the derivative changes sign) also have derivatives equal to zero. (Any point where the derivative of a function equals zero is called an *extreme point* or *extremum*. Setting the derivative of a function equal to zero and solving for the extrema is called *extremizing* a function.) You must always remember to test the root of the derivative to see if you've found a minimum, maximum, or inflection point. The easiest test involves the sign of the second derivative. If the second derivative at the point is positive, you've found a minimum. If it's negative, you've found a maximum. If the second derivative is zero, you've found an inflection point.

A similar approach works for optimizing multivariate functions. In this case one solves for points where the gradient is equal to zero, checking that you've not found an inflection point (called "saddle points" in higher dimensions).

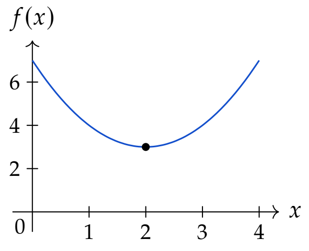


Figure 11.3: The function $f(x) = 3 + (x-2)^2$ has a minimum of $f = 3$ at argmin $x = 2$.

11.1.2 Constrained Optimization

Constrained optimization problems cannot be solved by finding roots of the derivatives of the objective. Why? It is possible that the minima or maxima of the unconstrained problem lie outside the feasible region of the constrained problem. Consider our previous example of $f(x) = 3 + (x-2)^2$, which we know has an

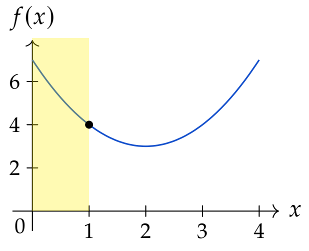


Figure 11.4: The yellow region is the feasible space ($x \leq 1$). The global argmin occurs at $x = 1$. The derivative of the function is not zero at this point.

argmin at $x = 2$. Say we want to solve the constrained problem

$$\min f(x) = 3 + (x-2)^2 \quad \text{s.t.} \quad x \le 1$$

The root of the derivative of f is still at $x = 2$, but values of x greater than one are not feasible. From the graph we can see that the minimum feasible value occurs at $x = 1$. The value of the derivative at $x = 1$ is -2, not zero.

In general, constrained optimization is a challenging field. Finding global optima for constrained problems is an unsolved area or research, one which is beyond the scope of this course. However, there are classes of problems that we can solve to optimality using the tools of linear algebra. These problems form the basis of many advanced techniques in data science.

11.2 Convexity

Many "solvable" optimization problems rely on a property called *convexity*. Both sets and functions can be convex.

11.2.1 Convex sets

A set of points is *convex* if given any two points in the set, the line segment connecting these points lies entirely in the set. You can move from any point in the set to any other point in the set without leaving the set. Circles, spheres, and regular polygons are examples of convex sets.

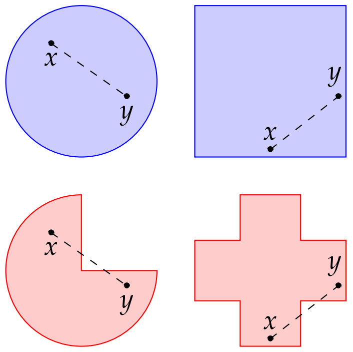


Figure 11.5: The blue shapes are convex. The red shapes are not convex.

To formally define convexity, we construct the line segment between any two points in the set.

Definition. *A set S is convex if and only if given any $\mathbf{x} \in S$ and $\mathbf{y} \in S$ the points $\lambda\mathbf{x} + (1-\lambda)\mathbf{y}$ are also in S for all scalars $\lambda \in [0,1]$.*

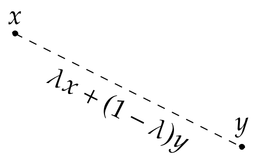


Figure 11.6: The segment connecting x and y can be defined as $\lambda x + (1-\lambda)y$ for $\lambda \in [0,1]$.

The expression $\lambda\mathbf{x} + (1-\lambda)\mathbf{y}$ is called a *convex combination* of $\mathbf{x}$ and $\mathbf{y}$. A convex combination of two points contains all points on the line segment between the two points. To see why, consider the 1-dimensional line segment between points 3 and 4.

$$\lambda(3) + (1-\lambda)(4) = 4 - \lambda, \quad \lambda \in [0,1]$$

When $\lambda = 0$, the value of the combination is 4. As λ moves from 0 to 1, the value of the combination moves from 4 to 3, covering all values in between.

Convex combinations work in higher dimensions as well. The convex combination of the vectors $\begin{pmatrix}1\\0\end{pmatrix}$ and $\begin{pmatrix}0\\1\end{pmatrix}$ is

$$\lambda\begin{pmatrix}1\\0\end{pmatrix} + (1-\lambda)\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}\lambda\\1-\lambda\end{pmatrix}$$

The combination goes from the first point $\begin{pmatrix}0\\1\end{pmatrix}$ when $\lambda = 0$ to the second point $\begin{pmatrix}1\\0\end{pmatrix}$ when $\lambda = 1$. Halfway in between, $\lambda = 1/2$ and the combination is $\begin{pmatrix}1/2\\1/2\end{pmatrix}$, which is midway along the line connecting $\begin{pmatrix}1\\0\end{pmatrix}$ and $\begin{pmatrix}0\\1\end{pmatrix}$. Sometimes it is helpful to think of a convex combination as a weighted sum of $\mathbf{x}$ and $\mathbf{y}$. The weighting (provided by λ) moves the combination linearly from $\mathbf{y}$ to $\mathbf{x}$ as λ goes from 0 to 1.

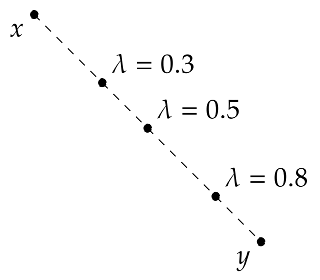


Figure 11.7: A convex combination in 2D: $\lambda x + (1-\lambda)y$.

11.2.2 Convex functions

There is a related definition for *convex functions*. This definition formalizes our visual idea of convexity (lines that curve upward) and concavity (lines that curve downward).

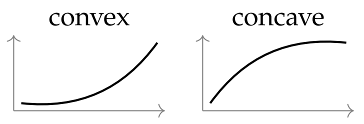
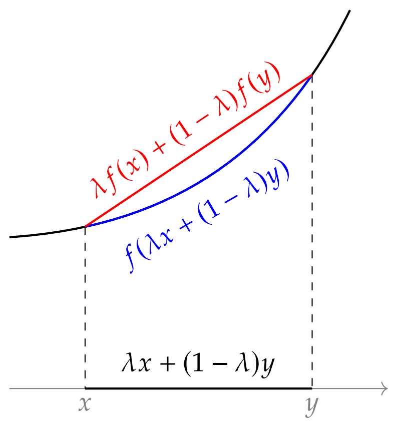


Figure 11.8: Convex functions curve upward. Concave functions curve downward.

Definition 1. *A function f is convex if and only if*

$$f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \le \lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y}), \quad \lambda \in [0,1]$$

This definition looks complicated, but the intuition is simple. If we plot a convex (upward curving) function, any chord – a segment drawn between two points on the line – should lie above the line. We can define the chord between any two points on the line, say $f(\mathbf{x})$ and $f(\mathbf{y})$ as a convex combination of these points, i.e. $\lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y})$. This is the right hand side of the above definition. For convex functions, we expect this cord to be greater than or equal to the function itself over the same interval. The interval is the segment from $\mathbf{x}$ to $\mathbf{y}$, or the convex combination $\lambda\mathbf{x}+(1-\lambda)\mathbf{y}$. The values of the function over this interval are therefore $f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y})$, which is the left hand side of the definition.



$\lambda f(x) + (1-\lambda)f(y)$
$f(\lambda x + (1-\lambda)y)$
$\lambda x + (1-\lambda)y$
x
y

Figure 11.9: The chord connecting any two points of a convex function (red) lies above the function (blue).

11.2.3 Convexity in Optimization

Why do we care about convexity? In general, finding local optima during optimization is easy; just pick a feasible point and move downward (during minimization) until you arrive at a local minimum. The truly hard part of optimization is finding global optima. How can you be assured that your local optimum is a global optimum unless you try out all points in the feasible space?

Fortunately, convexity solves the local vs. global challenge for many important problems, as we see with the following theorem.

Theorem. *When minimizing a convex function over a convex set, all local minima are global minima.*

Convex functions defined over convex sets must have a special shape where no *strictly* local minima exist. There can be multiple local minima, but all of these local minima must have the same value (which is the global minimum).

All local minima are *less than or equal to* the global minimum. Strictly local minima must be *less than* the global minimum.

Let's prove that all local minima are global minima when minimizing a convex function over a convex set.

Proof. Suppose the convex function f has a local minimum at $\mathbf{x}'$ that is not the global minimum (which is at $\mathbf{x}^*$). By the convexity of f,

$$f(\lambda\mathbf{x}' + (1-\lambda)\mathbf{x}^*) \leq \lambda f(\mathbf{x}') + (1-\lambda)f(\mathbf{x}^*)$$

Since $\mathbf{x}'$ is at a local, but not global, minimum, we know that $f(\mathbf{x}') > f(\mathbf{x}^*)$. If we replace $f(\mathbf{x}^*)$ on the right hand side by the larger quantity $f(\mathbf{x}')$, the inequality ($\leq$) becomes a strict inequality ($<$). (Even if both sides were equal, adding a small amount to the right hand side would still make it larger.) We now have

$$f(\lambda\mathbf{x}' + (1-\lambda)\mathbf{x}^*) < \lambda f(\mathbf{x}') + (1-\lambda)f(\mathbf{x}')$$

which, by simplifying the right hand side, becomes

$$f(\lambda\mathbf{x}' + (1-\lambda)\mathbf{x}^*) < f(\mathbf{x}')$$

This statement says that the value of the function f on any point on the line segment from $\mathbf{x}'$ to $\mathbf{x}^*$ is less than the value of the function at $\mathbf{x}'$. If this is true, we can find a point arbitrarily close to $\mathbf{x}'$ that is below our supposed local minimum $f(\mathbf{x}')$. Clearly, $f(\mathbf{x}')$ cannot be a local minimum if we can find a lower point arbitrarily closer to it. Our conclusion contradicts our original supposition. No local minimum can exist that are not equal to the global minimum. □

For a simpler, yet less intuitive argument, let $\lambda = 1$. Then the inequality becomes $f(\mathbf{x}') < f(\mathbf{x}')$, which is nonsense.

The previous proof seemed to rely only on the convexity of the objective function, not on the convexity of the solution set. The role of convexity of the set is hidden. When we make an argument about a line drawn from the local to the global minimum, we assume that all the points on the line are feasible. Otherwise, it does not matter if they have a lower objective than the local minimum, since they would not be allowed. By assuming the solution set is convex, we are assured that any point on this line is also feasible.

11.2.4 Convexity of Linear Systems

This course focuses on linear functions and systems of linear equations. It would be enormously helpful if linear functions and the solution set of linear systems were convex. Then we can look for local optima during optimization and know that we've found global optima.

Let's first prove the convexity of linear functions. For a function to be convex, we require that a line segment connecting any two points in the line lie above or one the line. For linear functions, this is intuitively true. The line segment connecting any two points is the line itself, so it always lies on the line. As a more formal argument, we describe a linear function as the product between a vector of coefficients $\mathbf{c}$ and $\mathbf{x}$, i.e. $f(\mathbf{x}) = \mathbf{c}^\mathsf{T}\mathbf{x}$. Let's start with the values of the function over the range spanned by arbitrary points $\mathbf{x}$ and $\mathbf{y}$. The segment of the domain corresponds to the convex combination $\lambda\mathbf{x} + (1-\lambda)\mathbf{y}$. The values of the function over this interval are

By convention, all vectors are column vectors, including $\mathbf{c}$; this requires a transposition to be conformable for multiplication by $\mathbf{x}$.

$$\begin{aligned} f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) &= \mathbf{c}^\mathsf{T}(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \\ &= \mathbf{c}^\mathsf{T}\lambda\mathbf{x} + \mathbf{c}^\mathsf{T}(1-\lambda)\mathbf{y} \\ &= \lambda\mathbf{c}^\mathsf{T}\mathbf{x} + (1-\lambda)\mathbf{c}^\mathsf{T}\mathbf{y} \\ &= \lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y}) \end{aligned}$$

which satisfies the definition of convexity: $f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \le \lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y})$.

Now let's turn to a linear system $\mathbf{Ax} = \mathbf{b}$. We want to show that the set of all solutions for this system (the *solution space*) is convex. Let's assume we have two points in the solution space, $\mathbf{x}$ and $\mathbf{y}$. Since $\mathbf{x}$ and $\mathbf{y}$ are solutions, we know that $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{Ay} = \mathbf{b}$. If the solution set is convex, any point in the convex combination of $\mathbf{x}$ and $\mathbf{y}$ is also a solution.

Following the conventions of the optimization field, we call the right hand side of linear systems the column vector $\mathbf{b}$, not $\mathbf{y}$ as we have said previously.

$$\begin{aligned} \mathbf{A}(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) &= \mathbf{A}\lambda\mathbf{x} = \mathbf{A}(1-\lambda)\mathbf{y} \\ &= \lambda\mathbf{Ax} + (1-\lambda)\mathbf{Ay} \\ &= \lambda\mathbf{b} + (1-\lambda)\mathbf{b} \\ &= \mathbf{b} \end{aligned}$$

Since $\mathbf{A}(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) = \mathbf{b}$, we know that all points on the line between $\mathbf{x}$ and $\mathbf{y}$ are solutions, so the solution set is convex.

Chapter 12

Gradient Descent

It's time to formalize the walking downhill method of optimization. This section introduces the *gradient descent* method, an iterative technique that takes steps downhill until a local minimum is found. As we will see in the coming chapters, gradient descent is not a single algorithm but instead a family of related algorithms. The defining feature of gradient descent is the use of local curvature of the objective function — the gradient — to identify the downhill direction.

12.1 Optimization by Gradient Descent

Let's begin with some notation. Our goal is to solve the problem

$$\min_{\mathbf{x}} f(\mathbf{x}),$$

which is read "minimize, by choice of $\mathbf{x}$, the function $f(\mathbf{x})$". We are searching for an input vector $\mathbf{x}$ that minimizes the scalar-valued function f. Optimization problem require that the objective function f be scalar-valued.

If the objective is a multivariate function $\mathbf{f}$, we can minimize $\|\mathbf{f}\|$ instead.

Gradient descent is an iterative technique. We begin with an initial guess $\mathbf{x}^{(0)}$, we find a sequence of better guesses

$$\mathbf{x}^{(0)} \to \mathbf{x}^{(1)} \to \mathbf{x}^{(2)} \to \cdots \to \mathbf{x}^{(k-1)} \to \mathbf{x}^{(k)}$$

so that each guess decreases the objective function:

$$f(\mathbf{x}^{(0)}) > f(\mathbf{x}^{(1)}) > f(\mathbf{x}^{(2)}) > \cdots > f(\mathbf{x}^{(k-1)}) > f(\mathbf{x}^{(k)}).$$

We keep iterating with gradient descent until there is no downhill direction, at which point we are by definition at a local minimum. The final guess $\mathbf{x}^{(k)}$ will be the local argmin.

The key to gradient descent is the *update rule*, a formula that tells us how to pick a next guess from the current guess. Imagine we are in the middle of gradient descent at guess $\mathbf{x}^{(k)}$. The update rules says that the next guess $\mathbf{x}^{(k+1)}$ will be our current guess plus a step in the downhill direction, or

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \left(\text{downhill step}\right).$$

Whatever this "downhill step" is, we can already see that it must be a vector. The current guess $\mathbf{x}^{(k)}$ is a vector, and addition is only defined if the downhill step is a vector of the same size. Each entry in the downhill step vector is a downhill step for the corresponding entry in our guess $\mathbf{x}^{(k)}$.

It helps to break the downhill step vector into two parts: a vector that points in the downhill direction, and a scalar that represents the size of the step we'll take. We can rewrite the update rule as a product of these two parts:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \left(\text{step size}\right)\left(\text{downhill direction}\right).$$

The step size is simply a scalar, so let's call it α and forget about it for a while. The step size is a *hyperparameter* of gradient descent. A hyperparameter is a variable in a training algorithm that is not a parameter of the model. Hyperparameters affect how a model is trained but are not used to make predictions once the training is complete. We'll have much more to say about the step size hyperparameter later in the chapter. For now, our update rule is

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\left(\text{downhill direction}\right).$$

What direction is downhill? We can find a direction that decreases the objective function f using its gradient. Let's define the function $\mathbf{g}(\mathbf{x})$ to be the gradient of the function f. Notice how the function $\mathbf{g}$ is vector-valued (and therefore written in bold font). The gradient is a vector of partial derivatives, one for every input:

$$\mathbf{g}(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}.$$

Importantly, **the gradient points uphill, not downhill**, as shown in Figure 12.1. The downhill direction is the negative of the gradient: $-\mathbf{g}(\mathbf{x})$.

Putting everything together, our final update rule is

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha\,\mathbf{g}(\mathbf{x}^{(k)}) \tag{12.1}$$

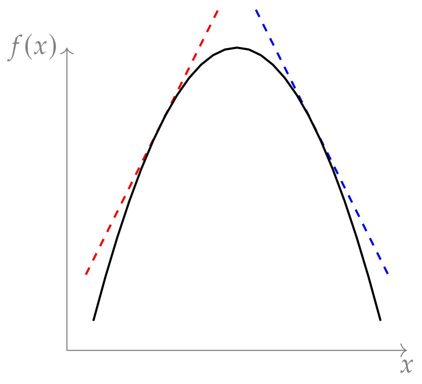


Figure 12.1: The gradient points in the uphill direction. The red tangent line has a positive gradient (slope), but the downhill direction is in the negative direction. The blue tangent line has a negative gradient but the downhill direction points toward $+x$.

where $\mathbf{g}(\mathbf{x}^{(k)})$ is the gradient of the objective function evaluated at the current iterate $\mathbf{x}^{(k)}$. We'll see this update rule repeatedly, and we must remember the origin of the minus sign. The new iterate $\mathbf{x}^{(k+1)}$ is the previous iterate $\mathbf{x}^{(k)}$ plus a step in the downhill direction; however, the downhill direction is $-\mathbf{g}(\mathbf{x}^{(k)})$, and the negative sign on the gradient can mislead us into thinking that we're subtracting, rather than adding, a step in the downhill direction. If you find yourself forgetting why there's a minus sign, just remember that the update rule can also be written $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\,(-\mathbf{g}(\mathbf{x}^{(k)}))$.

In multivariable calculus you may have written the gradient of a function $f(\mathbf{x})$ as $\nabla f(\mathbf{x})$. We avoid this notation in favor of $\mathbf{g}(\mathbf{x})$ for three reasons.

1. The symbol $\mathbf{g}(\mathbf{x})$ is simpler and can be bolded to remind us that the gradient is a vector.

2. It emphasizes the connection between optimization and root finding using the gradient ($\mathbf{g}(\mathbf{x}) = \mathbf{0}$).

3. Some gradient descent algorithms do not use the true gradient of the objective function, instead relying on an approximation or estimate of the gradient. We can think of the function $\mathbf{g}(\mathbf{x})$ as any "gradient-like" thing and still use the update rule in equation (12.1).

Let's stop for some examples. The first example is a the one-dimensional polynomial

$$f(x) = x^4 - 2x^3 - 23x^2 + 24x + 147.$$

We can use gradient descent to find a local minimum beginning with the guess $x^{(0)} = 2$. In one dimension, the gradient of $f(x)$ is the ordinary derivative

$$g(x) = \frac{df}{dx} = 4x^3 - 6x^2 - 46x + 24.$$

Let's assume we're given a step size $\alpha = 0.01$; we'll play around with the step size later. We begin iterating with our update rule.

$$\begin{aligned} x^{(1)} &= x^{(0)} - \alpha\, g(x^{(0)}) \\ &= 2 - 0.01\, g(2) \\ &= 2.6 \end{aligned}$$

The initial iterate $x^{(0)}$ had an objective value of $f(x^{(0)}) = 103$. After one round of gradient descent, the next iterate ($x^{(1)} = 2.6$) decreased the objective function to $f(x^{(1)}) = 64.4656$. The following tables shows the results of the first eight iterations.

Iteration	x	$f(x)$
0	2	103
1	2.6	64.46560
2	3.25856	24.53280
3	3.77059	5.41262
4	3.97379	3.03341
5	3.99919	3.00003
6	3.99998	3.00000
7	4.00000	3.00000
8	4.00000	3.00000

12.2 Linear Least-squares with Gradient Descent

As a second example, let's fit a linear model using gradient descent. As we learned in Chapter 8, the linear regression problem $\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$ can be solved by pseudoinverting the design matrix $\mathbf{X}$ to find the parameter estimates $\boldsymbol{\beta} = \mathbf{X}^+\mathbf{y}$. Pseudoinversion gives the least-squares estimates for the parameters $\boldsymbol{\beta}$, and the same solution can be obtained by minimizing the loss function

$$L(\boldsymbol{\beta}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i^{\text{pred}} - y_i^{\text{true}}\right)^2$$

Let's solve the linear regression problem using gradient descent on the loss function. In the univariate case with an intercept, our linear model takes the form $y^{\text{pred}} = \beta_0 + \beta_1 x$. Using the five data points from the table on page 59, our loss function is

$$L(\boldsymbol{\beta}) = \frac{1}{2}\sum_{i=1}^{5}\left(\beta_0 + \beta_1 x_i - y_i^{\text{true}}\right)^2$$

Be careful with the notation here. The function we are minimizing is the loss L (not f as before), and we are searching for a parameter vector $\boldsymbol{\beta}$ to minimize the loss. As for all linear regression problems, the pairs of data (x_i, y_i) are known.

Our first step is to calculate the gradient of the loss function

$$\mathbf{g}(\boldsymbol{\beta}) = \begin{pmatrix} \frac{\partial L}{\partial \beta_0} \\ \frac{\partial L}{\partial \beta_1} \end{pmatrix}.$$

Following the procedure in Section 8.3 for taking derivatives of sums, the entries in the gradient function are

$$\frac{\partial L}{\partial \beta_0} = \sum_{i=1}^{5} \left(\beta_0 + \beta_1 x_i - y_i^{\text{true}}\right)$$

and

$$\frac{\partial L}{\partial \beta_1} = \sum_{i=1}^{5} \left(\beta_0 + \beta_1 x_i - y_i^{\text{true}}\right) x_i.$$

The update rule for gradient descent is

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} - \alpha\, \mathbf{g}(\boldsymbol{\beta}^{(k)}),$$

remembering again that we are iterating over the parameters $\boldsymbol{\beta}$, not $\mathbf{x}$. We need an initial guess for the parameters, and lacking any insight from the problem we will choose the zero vector: $\boldsymbol{\beta}^{(0)} = \mathbf{0}$. Using a step size $\alpha = 0.1$, we can begin iterating.

Iteration	Loss	β_0	β_1
0	1.1375	0.0	0.0
50	0.0599557	0.0979449	1.04396
100	0.0544082	0.0332741	1.17936
150	0.0542539	0.0224895	1.20194
200	0.0542496	0.0206910	1.20571
250	0.0542495	0.0203911	1.20634
300	0.0542495	0.0203411	1.20644
350	0.0542495	0.0203327	1.20646
400	0.0542495	0.0203313	1.20646
450	0.0542495	0.0203311	1.20646
500	0.0542495	0.0203311	1.20646

Gradient descent found the same parameters as pseudo-inversion for our linear regression example. This is expected since the least-squares problem is convex and has a unique solution. Gradient descent terminates at a local minimum, and all local minima are global minima for convex problems. If gradient descent works so well, why don't we use it on linear least-squares problems in practice? There are two reasons:

1. Statistics of the parameters (p-values, confidence intervals, etc.) are computed from matrices that are also used to find the pseudoinverse. If we used

gradient descent on a linear regression problem, we would need to perform most of the pseudoinverse calculation anyway.

2. Gradient descent requires an initial parameter guess and a value for the step size hyperparameter. Both of these can be avoided by pseudoinversion.

Still, gradient descent plays an important role in regression. In the coming chapters we will introduce two powerful extensions to linear models — logistic regression and regularized regression — that cannot be fit using pseudoinversion. We will use gradient descent to parameterize these models.

12.3 Termination Conditions

Using the update rule (equation (12.1)) we can always find a next estimate of the input that is closer to a local argmin. In both of the previous examples, however, the iterates became so close to the local argmin that iterates stopped moving. Gradient descent conveniently self-terminates once we find a local minimum. To see why, consider the update rule $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha\,\mathbf{g}(\mathbf{x}^{(k)})$. Imagine if $\mathbf{x}^{(k)}$ is exactly a local argmin. The gradient is flat in all direction at a local minimum, so $\mathbf{g}(\mathbf{x}^{(k)}) = \mathbf{0}$. Therefore, the update rule says that

$$\begin{aligned}\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \alpha\,\mathbf{g}(\mathbf{x}^{(k)})\\ &= \mathbf{x}^{(k)} - 0\\ &= \mathbf{x}^{(k)}\end{aligned}$$

and the next iterate remains at the local argmin.

Self-termination is convenient, but it is rarely practical. Gradient descent terminates only when the gradient is exactly zero, so termination requires we land exactly on the a local argmin (or at least land within the precision of the computer). Usually we're not so lucky. Also, self-termination requires the gradient to shrink nicely to zero in a small region around the local argmin. As we will see in Chapter 14, there are many important optimization problems where such continuity is not guaranteed.

Alternatively, we can terminate gradient descent when we are satisfactorily near a local minimum. Two criteria are commonly used to halt gradient descent:

1. **Iterate convergence**. As we approach a local minimum and the gradient shrinks, the steps between iterates should also decrease. One strategy is to terminate gradient descent if the iterate $\mathbf{x}^{(k+1)}$ is very close to $\mathbf{x}^{(k)}$. Formally, we define some small value ϵ and stop iterating if $\left\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\right\| < \epsilon$. Any

norm (1-norm or 2-norm) will work, but a common choice is the max-norm (∞-norm) that measures the maximum distance between any corresponding elements in $\mathbf{x}^{(k+1)}$ and $\mathbf{x}^{(k)}$. The max-norm will keep gradient descent running if at least one dimension is still moving.

2. **Objective convergence**. The value of the objective function should also stop changing as gradient descent approaches a local minimum. A second termination strategy is to stop iterating when

$$\left| f(\mathbf{x}^{(k+1)}) - f(\mathbf{x}^{(k)}) \right| < \epsilon$$

for some small number ϵ. Terminating based on the objective value avoids the need to choose a norm since the objective function is always scalar-valued. However, a large change in the input to a function may lead to a small change in the objective value, so the objective convergence method may cause gradient descent to terminate while the iterates are still changing.

Both iterate and objective convergence have weaknesses. For example, objective convergence should not be used if the objective function is very flat, and iterate convergence can fail to terminate if the gradient is discontinuous near the local minimum. Some software packages apply both tests and terminate if either the iterates or objective values converge.

If the gradient is zero at a local minimum, why can't we use the magnitude of the gradient as a termination test? There are at least two reasons to avoid testing the gradient. First, it is common to use gradient descent with only an estimate of the gradient, and this estimate may not vanish completely at the local minimum. Second, terminating based on the gradient assumes the gradient is continuous and defined near the local argmin. As we'll see later, we can still solve optimization problems with gradient descent even if none of these conditions hold.

12.4 * Step Sizes

Walking downhill via gradient descent will bring us closer to a local minimum, but we also need to stop walking once we reach the bottom. The process of stopping at a local minimum is called *convergence*. We need some assurance that gradient descent will converge. We want our steps to become smaller as we approach the local minimum and disappear completely if we happen to arrive exactly at the local minimum. Conversely, if we are far away from the local minimum, we want to take large steps so we reach the local minimum in a reasonable amount of time. Think about finding a parking space for your car. You drive relatively quickly up and down the lanes of the parking lot until you close in on an open spot; then you slow down considerably to avoid hitting adjacent cars when entering the spot.

The step taken during each iteration of gradient descent is the product of two parts: a step size α and the direction $\mathbf{g}$. We have two methods of altering the length of the step at each iteration: 1.) decrease the step size α as the iterations increase, or 2.) decrease the magnitude of the gradient. Let's consider each method, starting with the step size.

12.4.1 * Step Size Scheduling

Gradient descent moves us closer to a local minimum at each iteration, so decreasing the step size α at each iteration will force us to take smaller steps as we get closer to the local minimum. The problem is timing these two events. We cannot say in advance how many iterations we need to get close to the local minimum. Decreasing the step size early on will slow convergence, but decreasing it too late can make us overshoot or zigzag around the local minimum.

Changing the step size requires the creation of a *step size schedule*. The schedule is simply a method that tells the gradient descent algorithm what step size to use at each iteration. A step size that follows a schedule requires a slight change in notation for the update rule at each iteration:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(x_k).$$

Rather than have a single, constant value α for all iterations, the step size at iteration k is α_k. The value of α_k is determined by the step size schedule.

There are many methods for constructing step size schedules, and the optimal schedule depends heavily on the function f to be minimized. While there is no universally best schedule, there are some properties of schedules that guarantee convergence. Remember that gradient descent goes on forever, moving us ever closer to a local minimum but never exactly there. We usually terminate gradient descent after a finite number of iterations, but we could let it run forever using infinitely many step sizes from the schedule. One method for ensuring convergence constrains the sums of all the step sizes in the schedule. The two constraints are

$$\sum_{k=0}^{\infty} \alpha_k = \infty \tag{12.2}$$

and

$$\sum_{k=0}^{\infty} \alpha_k^2 < \infty. \tag{12.3}$$

Equation (12.2) forces our step sizes to be large enough so that gradient descent does not stop prematurely before we are near the local minimum. The infinite sum of all the step sizes in the schedule must diverge to infinity.

It's easy to find a sequence of step sizes that sum to infinity — a constant step size would satisfy equation (12.2). We also need the step sizes to decrease as each iteration moves us closer to the local minimum. One method is to have the step size approach zero, and equation (12.3) ensures the decrease is rapid enough for convergence. Taken together, equations (12.2) and (12.3) define a "sweet spot" for step sizes schedules. The step sizes must be large enough so their sum diverges, but small enough so their squared sum converges. One step size schedule that satisfies the convergence criteria is $\alpha_k = 1/k$ (with $\alpha_0 \equiv 1$ to avoid dividing by zero). The first seven step sizes from this schedule are shown in the table below.

iteration	0	1	2	3	4	5	6
step size (α_k)	1.000	1.000	0.500	0.333	0.250	0.200	0.167

The series $\alpha_k = 1/k$ is the harmonic series, and the sum of this series diverges. The sum of the squares of the harmonic series converges, although the exact value it converges to is not important.

You might have noticed that if the sum of the step sizes is infinite but the sum of the squared step sizes is finite, then $\alpha_k^2 < \alpha_k$, at least when k is large. This implies that eventually $\alpha_k < 1$, and in practice it is rare to start with step sizes larger than one.

Chapter 13

Logistic Regression

Let's return to the problem of binary classification where a feature vector $\mathbf{x}$ is used to predict the class y of a sample. We've already used the Support Vector Machine to solve the binary classification problem. Recall that the SVM uses optimization to find a hyperplane $\mathbf{a} \cdot \mathbf{x} = b$ that separates the two classes. Although the SVM works well, it is difficult to understand how the algorithm predicts the class of new data. We could try to examine the support vectors that lie nearest the separating hyperplane, but in general we cannot directly interpret SVM models.

By contrast, we've seen how straightforward it is to interpret linear statistical models. We are able to assign meaningful interpretations to the fitted coefficients, and the relative importance of the predictor variables is quantified by the statistical outputs of the `fitlm` function. Ideally we would use linear models for the binary classification problem. However, there are two problems:

- The predictions of a classification algorithm are binary, while linear models make continuous predictions.
- Even if we force a linear model to make discrete predictions, we must also force the outputs of a linear model to stay within the set of classes.

In this chapter we develop a variant of linear regression called *logistic regression*. Logistic regression uses a linear model to predict binary outcomes. Rather than predict the class of a sample directly, a logistic regression model predicts the probability that the sample is in each class. We will show how a *link function* can be used to map the output of a linear model into a bounded range, like the interval $[0, 1]$ for probabilities.

13.1 Predicting Odds

You're probably familiar with probabilities as the long-run expectation of an uncertain process. Logistic regression uses a related concept called the *odds*. You may have used the term "odds" interchangeably with "probability," but they are not the same. Let's assume that a random variable y has two possible outcomes, 0 and 1. The odds of y is the ratio of the probability that y equals 1 to the probability that y equals 0, or

Pun intended.

$$\text{odds}(y) = \frac{P(y=1)}{P(y=0)}.$$

For example, if $\text{odds}(y) = 2$ then the probability that $y = 1$ is twice as large as the probability that $y = 0$. We can convert between probabilities and odds by remembering that probabilities sum to one, or P(y=0) + P(y=1) = 1. Then

Odds are usually expressed as a proportion, so an odds of 2 is written as 2:1, or "two to one".

$$\text{odds}(y) = \frac{P(y=1)}{P(y=0)} = \frac{P(y=1)}{1-P(y=1)} \Rightarrow P(y=1) = \frac{\text{odds}(y)}{1+\text{odds}(y)}.$$

The odds function lives interval $[0,\infty)$. The odds of y become infinite as the probability that $y = 1$ increases. The odds of y go to zero as the probability that $y = 0$ increases. This means that the logarithm of the odds, or the "log odds" is a continuous value in the interval $(-\infty,\infty)$, which is the same range as the predictions of a linear model. We can build a binary classifier by using a linear model to predict the log odds of the response variable y, i.e.

Some people go further and refer to the log odds as the "lods".

$$\log(\text{odds}(y)) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

The function $\log(\text{odds}(y))$ is called the *logit* function. Because it links the response variable to the linear models, we refer to the logistic (and other similar functions) as *link functions*.

To summarize:

$$\begin{aligned} y &\in 0 \text{ or } 1 \\ P(y=1) &\in [0,1] \\ \text{odds}(y) &\in [0,\infty) \\ \log(\text{odds}(y)) &\in (-\infty,\infty) \end{aligned}$$

13.2 From Odds to Probabilities

Log odds can be predicted using linear models, but it is difficult for most people to interpret the odds, much less their logarithm. Ideally we would have our logistic regression model predict probabilities. The logistic regression model from above was

$$\log(\text{odds}(y)) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

Exponentiating both sides to gives

$$\text{odds}(y) = e^{\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p} \equiv e^t$$

where the placeholder t equals the output of the linear model. We can solve for the probability that y equals 1 using the relationship between probabilities and odds.

$$\begin{aligned} P(y=1) &= \frac{\text{odds}(y)}{1+\text{odds}(y)} \\ &= \frac{e^t}{1+e^t} \\ &= \frac{1}{1+e^{-t}} \end{aligned}$$

For the last step we divided both the numerator and denominator by e^t.

This function is called the *logistic* or *sigmoid* function, and its shape is shown in Figure 13.1. The output of the function is restricted to the interval $[0,1]$ even though the inputs are unbounded. The bounded output makes it possible to interpret the outputs of the logistic function as probabilities.

Making predictions with logistic models is a two-step process. First, we use a linear model to predict the placeholder value t. The value of t is used to calculate the probability that the response is equal to one. If we were interested in classifying the response, we would say that $y=1$ if $P(y=1)>0.5$ and choose $y=0$ otherwise. Note that the point $P(y=1)=0.5$ occurs when $t=0$. When classifying with a logistic regression model, our response prediction switches from class 0 to class 1 when the output of the linear model $t=\beta_0+\beta_1x_1+\cdots+\beta_px_p$ switches from negative to positive.

Logistic regression is used for binary classification, so the model should alternate between predicting class 0 and class 1. The sigmoid shape is a compromise; it is smooth and continuous but still transitions rapidly from 0 to 1. The smoothness of the logistic function (and its convenient derivative) makes it easier to fit logistic regression models by gradient descent.

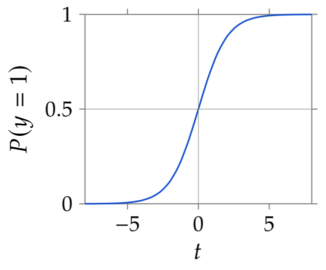


Figure 13.1: The logistic function.

13.3 Example: Predicting the risk of Huntington's Disease

Huntington's Disease is an inherited genetic condition caused by repeated CAG sequences in the Huntingtin (*HTT*) gene. Too many CAG repeats create a "glutamine knot" in the protein, causing toxic protein aggregates in neurons. Symptoms of Huntington's appear later life, and an individual's risk for developing the disease correlates with the number of CAG repeats.

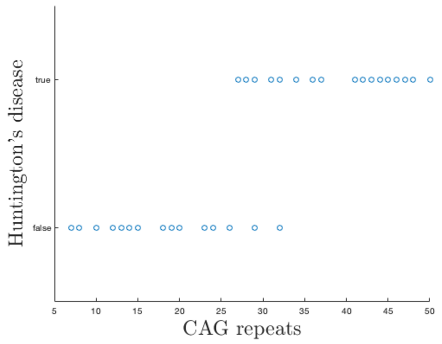
Source: Walker FO. Huntington's disease. *The Lancet*. 2007: **369**, (9557), 218–228.

# of CAG Repeats	Disease Outcome
< 28	Not affected.
28–35	Increases risk.
36–40	Affected; some offspring affected.
> 40	Affected; all offspring affected.

Let's build a model to predict the probability of developing Huntington's based on the number of CAG repeats. The response variable is binary (Huntington's disease or not) and the predictor variable is continuous (the number of CAG repeats in the *HTT* gene). To train the model we counted the number of CAG repeats in 50 individuals with and without the disease.

MATLAB code

```
load huntington.mat
scatter(hunt.CAGs,categorical(hunt.disease))
xlabel('CAG repeats',axargs{:})
ylabel("Huntington's disease",axargs{:})
```



We see from these data that predicting disease status with low (<25) or high (>35) CAG repeats is straightforward. However, there is a region between 25 and 35 CAG repeats where disease status is ambiguous. Let's build a logistic regression model to predict Huntington's status. We use the MATLAB function `fitglm`, for "fit generalized linear model". The `fitglm` function is similar to `fitglm`; the first argument is a table of data, and the second argument is a formula describing the model. However, `fitglm` can use a wide range of link functions and datatypes when fitting linear models. For logistic regression using binary responses we need to specify the logit link function and a binomial distribution.

MATLAB code

```
model = fitglm(hunt,'disease ~ CAGs','link','logit', ...
         'Distribution','binomial')
```

MATLAB output

```
model =
Generalized linear regression model:
  logit(disease) ~ 1 + CAGs
  Distribution = Binomial

Estimated Coefficients:
            Estimate      SE       tStat      pValue
            ________    _______   _______   ________
(Intercept)  -14.032     5.7832   -2.4263   0.015252
CAGs.        0.50558    0.20395    2.4789   0.013179

50 observations, 48 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 55, p-value = 1.18e-13
```

Remember that the model we're fitting is

$$\log(\text{odds}(\text{disease})) = \beta_0 + \beta_1[\text{CAGs}].$$

We know the best fit values of β_0 and β_1 from the output of the `fitglm` model:

$$\log(\text{odds}(\text{disease})) = -14.032 + 0.50558[\text{CAGs}].$$

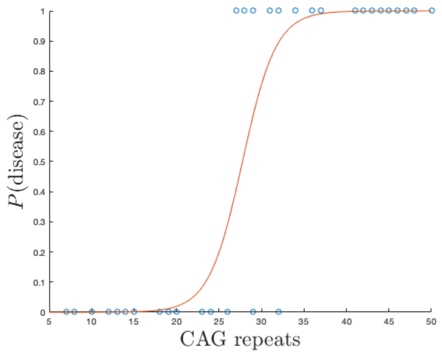
We can also rewrite this model to predict the probability of having Huntington's disease

$$P(\text{disease}) = \frac{1}{1 + e^{-14.032+0.50558[\text{CAGs}]}},$$

which we plot below along with the training data.

MATLAB code

```
scatter(hunt.CAGs,hunt.disease)
hold on
cag_range = linspace(5,50,100);
beta = model.Coefficients.Estimate;
plot(cag_range, 1./(1+exp(-(beta(1)+beta(2)*cag_range))))
hold off
xlabel('CAG repeats',axargs{:});
ylabel('$$P(\mathrm{disease})$$',axargs{:});
```



We are often interested in the point where $P(\text{disease}) = 0.5$, as this is the threshold number of CAG repeats where a person is equally likely to have or not have Huntington's. The logistic function reaches its midpoint when the linear model moves from negative to positive. Thus we can simply solve for when $\beta_0 + \beta_1[\text{CAGs}] = 0$.

$$\begin{aligned} -14.03 + 0.51[\text{CAGs}] = 0 \Rightarrow [\text{CAGs}] &= 14.03/0.51 \\ &\approx 28 \text{ CAG repeats} \end{aligned}$$

When the output of the linear model is zero,

$$P(y = 1) = \frac{1}{1 + e^0} = \frac{1}{2}$$

.

13.4 Interpreting coefficients as odds ratios

The coefficients of the linear part of a logistic regression equation are not directly interpretable. The coefficients describe how the linear model changes given a unit change in the input variables, but the outputs of the linear model undergo a nonlinear transformation before becoming a probability. Instead, we interpret logistic regression models by calculating the change in odds that accompany a unit change in an input variable. This change is odds is called the *odds ratio*. For example, we can define the odds ratio that corresponds to increasing variable x_i by 1 as

$$\text{odds ratio}(x_i) = \frac{\text{odds}(x_i + 1)}{\text{odds}(x_i)}.$$

Let's calculate the odds ratio for Huntington's disease that accompanies an increase of one CAG repeat.

$$\begin{aligned}
\text{odds ratio}([\text{CAGs}]) &= \frac{\text{odds}([\text{CAGs}]+1)}{\text{odds}([\text{CAGs}])} \\
&= \frac{e^{\beta_0+\beta_1([\text{CAGs}]+1)}}{e^{\beta_0+\beta_1[\text{CAGs}]}} \\
&= \frac{e^{\beta_0}e^{\beta_1[\text{CAGs}]}e^{\beta_1}}{e^{\beta_0}e^{\beta_1[\text{CAGs}]}} \\
&= e^{\beta_1}
\end{aligned}$$

Since $\beta_1 = 0.51$ in out model, having one more CAG repeat increases the odds of developing Huntington's disease by $e^{0.51} = 1.67$-fold. For any logistic regression model, the odds ratio for variable x_i is the exponential of the corresponding coefficient β_i.

$$\text{odds ratio}(x_i) = \frac{\text{odds}(x_i+1)}{\text{odds}(x_i)} = e^{\beta_i}$$

If β_i is negative the odds ratio e^{β_i} will be less than one and the odds will decrease.

You may have heard news reports that "doing X increases your risk of Y". Researchers performing this type of study often use logistic regression models to predict the odds of developing condition Y based on input variable X. The reported increase in risk is simply the odds ratio associated with the coefficient of X.

13.5 Fitting Logistic Regression Models

We fit a logistic regression model using a set of n training point $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i$ is a vector of input features and y_i is a binary output variable (either 0 or 1). The output of the logistic regression model is $P(y_i)$, the probability that $y_i = 1$ given an input $\mathbf{x}_i$. A perfect model would predict that

$$\begin{aligned}
P(y_i) &= 1 \quad \text{when } y_i \text{ is } 1 \\
P(y_i) &= 0 \quad \text{when } y_i \text{ is } 0
\end{aligned}$$

A common loss function for logistic regression is

$$L(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=1}^{n}\left[-y_i \log P(y_i) - (1-y_i)\log(1-P(y_i))\right] \tag{13.1}$$

where $P(y_i)$ is the output of the logistic function

$$P(y_i) = \frac{1}{1+e^{-t}}, \quad t = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

Let's examine the summand of the loss function

$$-y_i \log P(y_i) - (1-y_i)\log(1-P(y_i)).$$

When $y_i = 0$, this expression reduces to $-\log(1-P(y_i))$, which reaches a minimum of zero only when $P(y_i) = 0$. (When $P(y_i) \neq 0$, $1 - P(y_i)$ is less than one, so $-\log(1-P(y_i))$ is a positive, nonzero value). The other option is that $y_i = 1$, in which case the loss summand becomes $-\log P(y_i)$. This expression is minimized when $P(y_i) = 1$. Both cases are what we want in a loss function — to minimize the loss we set $P(y_i) = 0$ when $y_i = 0$ and $P(y_i) = 1$ when $y_i = 1$.

Remember that $P(y_i)$ is a probability, so it must lie in the range $[0,1]$. Also, the logarithm of 1 is 0, the logarithm of a number smaller than 1 is negative; and the logarithm of 0 approaches negative infinity.

The loss function in equation (13.1) isn't the only loss function that would work for logistic regression. The function $P(y_i)^{1-y_i}(1-P(y_i))^{y_i}$ is also minimized when the probability $P(y_i)$ matches the value of y_i. However, this loss function also has a maximum value of one for each training point. We prefer that our loss functions be unbounded above so that no matter how terrible our model is, making it worse will always increase the loss. Said another way, we always want to distinguish between "bad" solutions and "very bad" solutions; otherwise, if we started training at a very bad solution, there would be little change in the loss by improving to a merely bad solution. Since the derivative of the loss function drives our training updates, any plateau in our loss function will decrease the training rate.

The loss function in equation (13.1) did not appear out of thin air. Minimizing (13.1) is equivalent to maximizing the likelihood of the model predicting the training values. Actually, it maximizes the log-likelihood since taking the logarithm of the likelihood doesn't change the argmax but makes the function easier to compute.

13.5.1 Gradient Descent

Our loss function is nonlinear and must be minimized using gradient descent or another iterative approach. We first need to compute the gradient of the loss with respect to each parameter in $\boldsymbol{\beta}$.

$$\mathbf{g}(\boldsymbol{\beta}) = \begin{pmatrix} \frac{\partial L}{\partial \beta_0} \\ \vdots \\ \frac{\partial L}{\partial \beta_p} \end{pmatrix}$$

The loss function L depends on the probabilities P, which depend on the output t of the linear model, which depends on the parameters $\boldsymbol{\beta}$. This nested structure is a perfect opportunity to use the chain rule to compute the entries of the gradient. For a single parameter β_j

$$\frac{\partial L}{\partial \beta_j} = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial L_i}{\partial P_i}\frac{\partial P_i}{\partial t_i}\frac{\partial t_i}{\partial \beta_j}.$$

Let's compute each of the righthand side derivative in turn. For the loss function, it is simpler to compute the derivative separately when $y_i = 0$ and $y_i = 1$.

$$\frac{\partial L_i}{\partial P_i} = \begin{cases} \frac{1}{1-P(y_i)}, & \text{when } y_i = 0 \\ -\frac{1}{P(y_i)}, & \text{when } y_i = 1 \end{cases}$$

The probability P is the output of the logistic function, which has a convenient derivative.

$$\begin{aligned} \frac{\partial P_i}{\partial t_i} &= P(t_i)(1 - P(t_i)) \\ &= \frac{1}{1+e^{-t_i}}\frac{1}{1+e^{t_i}} \end{aligned}$$

Finally, we compute the derivative of the linear model $t = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$, being careful to handle the special case of the intercept ($i = 0$).

$$\frac{\partial t_i}{\partial \beta_j} = \begin{cases} 1, & \text{when } j = 0 \\ x_{ij}, & \text{when } j > 0 \end{cases}$$

The notation x_{ij} requires an explanation. Remember that we have n pairs of training data $(\mathbf{x}_i, y_i)$. The value x_{ij} is the jth entry in the ith feature vector of the training set.

With the derivative in hand, all we need to begin gradient descent is an initial guess for the parameter vector $\boldsymbol{\beta}$. A convenient guess is $\boldsymbol{\beta} = \mathbf{0}$. Setting all parameters equal to zero makes the output of the linear model $t = 0$ for every training point. The logistic function takes the value 0.5 when $t = 0$, so a zero initial guess begins right in the middle with a prediction that y_i is equally likely to be 0 or 1 for every training point. Unless we have some prior information that says otherwise, guessing 0 or 1 with equal probability is a fine place to start.

An accompanying MATLAB workbook implements gradient descent to fit the Huntington's model from earlier in this chapter.

Chapter 14

Bias, Variance, and Regularization

14.1 Learning vs. Memorizing

Just because we trained a model does not mean it learned anything useful from the data. We need to *test* the model to assess its accuracy. We test a model using data that were not used for training so we can distinguish between *learning* and *memorizing*. Memorizing occurs when a model simply remembers the correct outputs for each of the training inputs. When shown a training input again, the model can produce the correct result. However, if the model is given an input that was not included in the training set (i.e. an input that it has not memorized), the model cannot predict the correct value. By contrast, learning occurs when a model can predict correctly without memorizing. Models learn by finding relationships in the input features that hold information about the correct output. A model that learns well can usually make good predictions on new data since the feature relationships are still valid. Models that predict accurately on data that were not part of the training data are said to *generalize*.

Models that cannot generalize well are not useful. Such models have only memorize the correct answers for the training data, but we already know the answers to the training data! Testing our models with our training data cannot distinguish between models that memorize and models that learn. We need separate data that have not been shown to the model. Models that memorize will perform poorly on these data, but models that learned will do better.

All models more accurately predict the results of their training data compared to the testing data, so do not be alarmed if the testing accuracy is lower than the training accuracy. The reduced performance on testing data is called the *generalization gap*, and it affects all models even if they are not memorizing. In this chapter we will discuss why the generalization gap occurs and how to reduce it. We will also discuss methods to assess both the training and testing accuracy of our models.

14.2 Holdout

The simplest approach for validating a model is called *holdout*. Holdout removes a minority of the training data and sets it aside for testing. The holdout set can be used for validation since it was not used during training.

There are no rules for how much of the training data should be removed for holdout. The number of holdout observations, not the fraction of the entire dataset, is most important. For example, imagine if we only included two points in our holdout set when training a binary classifier (like logistic regression). There are only three possible outcomes when testing our model: 0/2, 1/2, or 2/2, making our accuracy 0, 0.5, or 1.0. This is clearly a crude assessment of our model's accuracy. If the holdout dataset includes n observations, the resolution of our accuracy estimate is $1/n$. A holdout set with 10 observations can only estimate the accuracy to within 0.1, and this is an upper bound. Small holdout sets are hampered by stochasticity. Testing points that happen to be similar to a training data are easier for a model to predict correctly. A few "good" or "bad" points can have a big effect if the holdout set is small.

It is common in machine learning to refer to accuracy as a fractional value, not as a percentage.

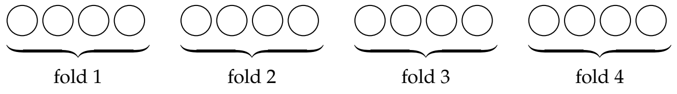
Perhaps counterintuitively, larger training sets require a smaller *fraction* of data be reserved for holdout. A training set with 20 observations could require 50% or more the data be set aside for holdout, and even then the validation accuracy would have a precision of no less than 0.1. By contrast, the Netflix Prize contest, a community-based competition to predict personalized ratings for movies, used a training dataset with over 100 million observations. The final validation set included only 1.36% of these points to award a $1,000,000 prize to the winning team.

Although the Netflix Prize holdout set contained over one million observations, the top two teams tied for accuracy. The tiebreaker went to the team that submitted 20 minutes before the other.

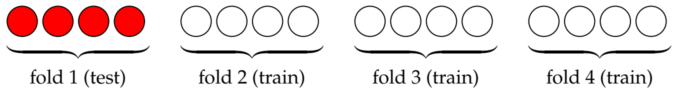
Holdout works well for large datasets where only a small fraction of the data need to be excluded from training. In small datasets a large fraction of data need to be removed for validation. **After validation, the holdout data can be added back to the training set before training a final model.** Thus the holdout data are not "lost", but for small datasets with large holdout the validation accuracy will be a poor estimate of the accuracy of the final model trained with the entire dataset. For large datasets, the expense of retraining the model often outweighs the gain in accuracy from including the small fraction of holdout points.

14.3 Cross Validation

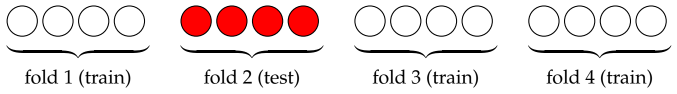
Cross validation is an alternative to holdout. In cross validation, all points in the dataset are used for training and testing, but never at the same time. Cross validation begins by splitting the dataset into a set of k groups of roughly equal size. Each group of data is called a *fold*, and points are randomly assigned to the folds. For example, at dataset with 16 points could be divided into $k = 4$ folds.



To begin cross validation, one of the folds is set aside for validation, similar to holdout. A model is trained using the remaining $k - 1$ folds and tested against the holdout fold.



Next we put fold 1 back into the training set and set aside fold 2 for testing. Then we re-train our model using folds 1, 3, and 4 and validate with fold 2.



This process continues $k = 4$ times, with the final model trained on folds 1–3 and validated with the data in fold 4. The final step is to average the accuracies across all k folds. This average is reported as the final accuracy, and a full model can be trained using all of the data.

The advantage of k-fold cross validation is that every point in the dataset is used for testing, so the method is not sensitive to which data are selected for holdout. However, the method is still stochastic as the accuracy of each model depends on how the data are randomly assigned to the folds. A k-fold cross validation requires training k separate models in addition to the final model with all of the data. This might be costly for very large datasets, so cross validation is more common in small- to medium-sized problems.

14.3.1 Leave-one-out Cross Validation

There is no rule for determining the number of folds (k) for a cross validation. Smaller datasets benefit from higher values of k since fewer points are held out at a time and training multiple models is not computationally prohibitive. One extreme of k-fold cross validation occurs when k equals the number of points in the dataset. In this case each fold contains only a single point, hence the name *leave-one-out* cross validation. Leave-one-out is the most computationally demanding strategy for cross validation, as a new model must be trained for each point in the dataset. However, leave-one-out provides the best estimate of the accuracy of the final model trained with all the data. Each of the validation sub-models is trained with all but one point, so these models closely resemble the performance of a model trained with all of the data. Since each fold contains one point, there is no randomness to the validation procedure if the training algorithm is deterministic.

Leave-one-out is abbreviated "L1O". Assigning two points per fold is called leave-two-out (L2O) cross validation.

14.4 Bias vs. Variance

Cross validation measures the accuracy of our models. We need to understand why our models are inaccurate before we can develop strategies to improve them. As a reminder, we care most about generalization accuracy — the ability to predict results that were not included in the training set. The fundamental source of all model inaccuracies is limited data. Take, for example, the data in Figure 14.1. The center panel shows a continuous function that we are trying to learn using six data points. The other eight panels show six randomly sampled points from the center function. It would be difficult to estimate the original function using any of these subsets alone. Any model fit to a subset of data would not generalize well to parts of the function that were not we represented in the training data.

Most datasets contain more than six training points, but remember that the function in Figure 14.1 is one-dimensional. Many of our machine learning methods are applied to high-dimensional data, so the *density* of training data may be lower than the sampling shown in the Figure 14.1. Data acquisition is enormously expensive, so we are often left with far less training data than we would like.

A model's error can be divided into two sources. The first source is *bias*. Bias appears when the model underfits the data because the model lacks the flexibility to match the underlying system. Imagine fitting a model that predicts how many text messages a person sends per day based on their age. Even if we had millions of training data, we could not possibly predict everyone's texting patterns using such a simple model. A teenager with a phone might text a lot, but not all teenagers have phones. Our model's predictions will have high bias since adding more data or switching to a new sample will not improve the predictions. Bias is robust to subsampling, meaning we will fit similar models to different datasets.

You can see high-bias models in the first column of Figure 14.2. Each model is a two-parameter linear model ($y = \beta_0 + \beta_1 x$) fit to six points sampled from the

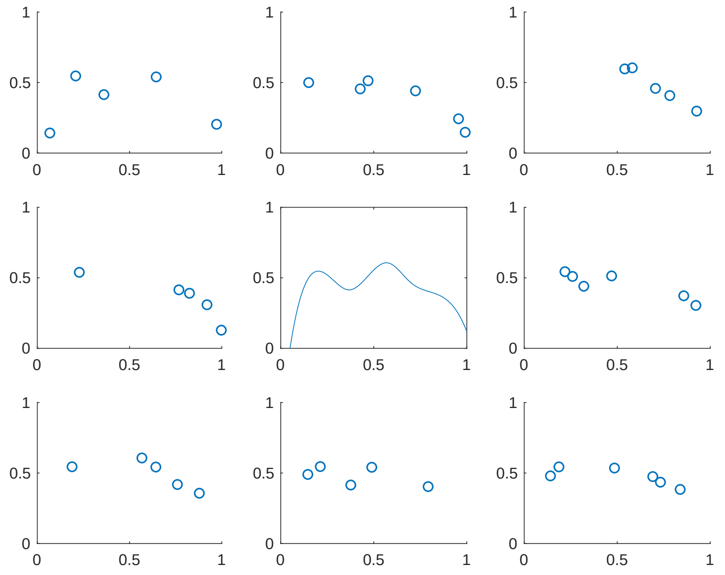


Figure 14.1: Samples of six points vary in their approximations of the function in the center panel.

true function. Although the six-point samples vary widely, the fit models and their generalization error are similar. Although the models are not sensitive to changes in the dataset, they do not approximate the function well. After all, we cannot expect a purely linear model to reproduce the nonlinear function shown at the top of the figure.

The other source of generalization error is model *variance*. Models with high variance are overfit to the data. High variance can been seen in right column of Figure 14.2. These are six-parameter curvilinear models ($y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_5 x^5$). Notice how well these models predict the training data (blue circles). In fact, since there are six parameters and six data points, these models predict the training data exactly! However, you should also notice how poorly some of these models would generalize. The top model predicts a huge spike between 0.5 and 1, and the middle model predicts an increase, not decrease, near 1.0. Neither of these model match the true function at the top of the figure. Models with high variance have two characteristics: 1.) they are good at memorizing training data, and 2.) they are highly sensitive to the training data. The two-parameter models on the left are similar for all three datasets, but the six-parameter models have very

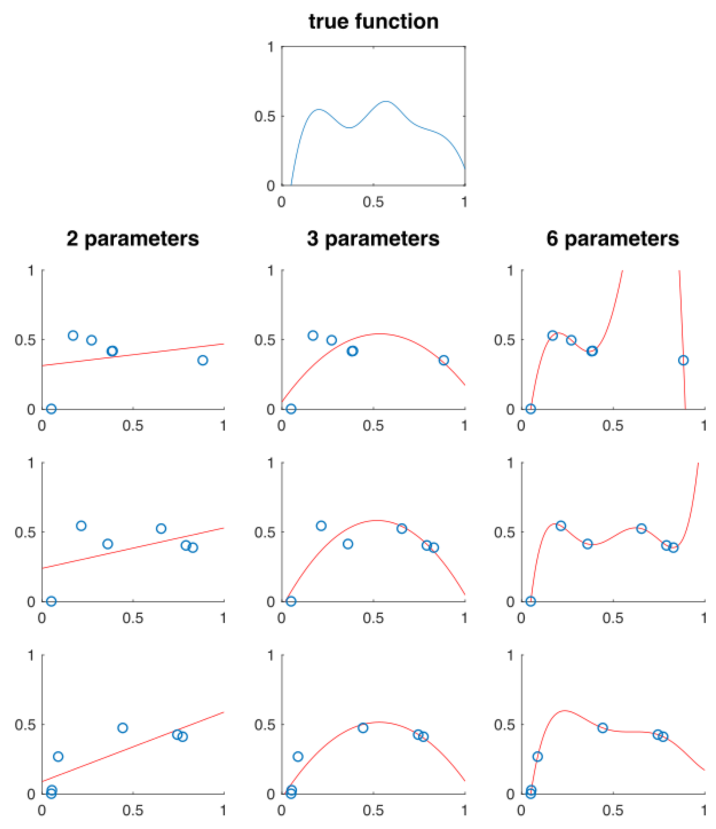


Figure 14.2: A model's bias and variance depend on the number of parameters.

different shapes.

The key to fitting models is to balance bias and variance. We want models that are flexible enough to match the true system (low bias) but not overly sensitive to the specifics of our training data (low variance). A three-parameter quadratic model ($y = \beta_0 + \beta_1 x + \beta_2 x^2$) achieves this balance for the function in Figure 14.2 (center column). The models are not exact, but each model resembles the true function regardless of which points are included in the training set. This example shows overly simple and overly complex models can both lead to poor generalization.

14.5 Regularization

Figure 14.2 is a toy example. We knew the true function and were able to collect multiple subsamples to test our model's accuracy. This allowed us to adjust the number of parameters until we found a model that balanced bias and variance. In reality, we won't be able to tune and retrain our model, partly because we don't know the true function we are trying to replicate.

A more general strategy is to start with a model that has more parameters than necessary and try to minimize overfitting. This approach is called *regularization*, and it relies on an observation that overfit models tend to have many parameters with large magnitudes. Keeping parameters small during training tends to produce models that generalize better. We regularize an algorithm by penalizing it whenever the model's parameters get too big. Formally, this is accomplished by adding a regularization term to the objective function.

14.5.1 The LASSO

Let's use regularization to prevent overfitting of a linear model. Linear models are trained using a quadratic loss function

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} \left(y_i^{\text{pred}} - y_i^{\text{true}} \right)^2 .$$

Here $\boldsymbol{\beta}$ is a vector of parameters. If the linear model has input features $\mathbf{x}$, the output $y^{\text{pred}} = \mathbf{x} \cdot \boldsymbol{\beta}$. We can substitute this model into our loss function to make it clear that we are minimizing the loss by selecting a set of parameters $\boldsymbol{\beta}$.

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} \left(\mathbf{x} \cdot \boldsymbol{\beta} - y_i^{\text{true}} \right)^2$$

Now we can add regularization. Remember that the goal of regularization is to keep the parameters in $\boldsymbol{\beta}$ from becoming too large and overfitting the model. We can add a term to our objective function so our algorithms tries to minimize both the total loss and the magnitudes of the parameters.

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} \left(\mathbf{x}\cdot\boldsymbol{\beta} - y_i^{\text{true}}\right)^2 + \lambda \sum_{i=1}^{p} |\beta_i| \tag{14.1}$$

The regularization term adds up the magnitudes of the parameters. This sum is weighted by a hyperparameter λ that balances the two objectives: minimize loss or minimize the parameter magnitudes. The hyperparamter λ can be any non-negative value. Setting $\lambda = 0$ eliminates all regularization, making Equation (14.1) equivalent to normal least-squares regression. Setting λ to a large value will focus most of the algorithm's attention on keeping the parameters small. If λ is large enough, the algorithm will ignore the loss entirely and set all of the parameters to zero. There is no definite rule for selecting a value for λ. Like all hyperparameters, it must be tuned for each problem to maximize performance. Cross validation can be used to ensure the value of λ promotes generalization of the trained model.

The regularized form of linear regression in Equation (14.1) is called the "least absolute shrinkage and selection operator", or LASSO. The effects of regularization can be seen in Figure 14.3. When $\lambda = 0$, the LASSO is equivalent to standard linear regression, so a six-parameter model overfits random samples of six data points. Adding regularization ($\lambda = 0.001$) decreases the variance of the models, as seen by the similar shapes of the models in the center column. Increasing the regularization ($\lambda = 0.01$) further reduces the variance, and all the models in the right column have the same overall shape. However, we are starting to see the effects of over-regularizing the model. Regularization penalizes large parameters, so the models are beginning to underpredict the data. Said another way, high regularization reduces the variance so much that we begin to see increased model bias.

The goal of regularization is to improve generalization, but this goal causes decreased accuracy on the training data. The first column of Figure 14.3 fits the training data exactly, but the curves are clearly overfit and do not resemble the true function. The regularized models in columns two and three do not predict the training exactly, although they will generalize better since they more reliably predict data outside the training set. Do not be alarmed if adding regularization decreases the training accuracy of your model. You are reducing your model's ability to memorize in hopes that it will generalize better.

The example in Figure 14.3 uses regularization to keep a polynomial model from overfitting data. This is only one application of the LASSO. As discussed in

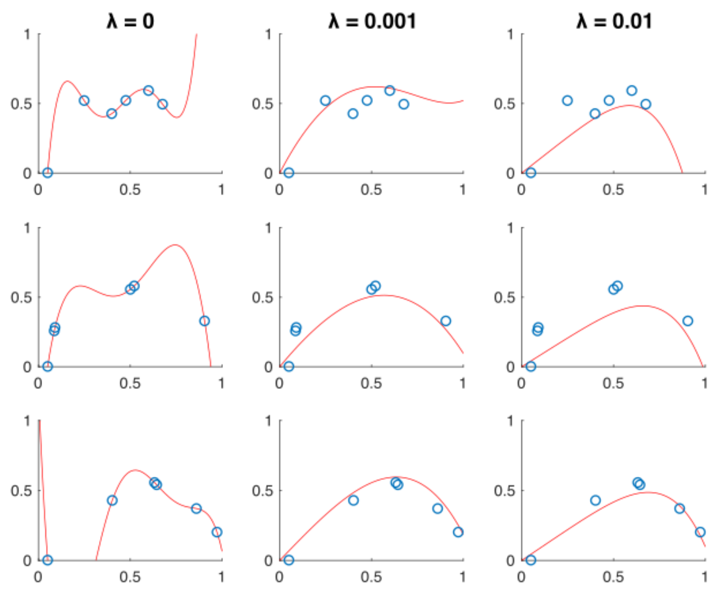


Figure 14.3: Increasing values of λ decrease the variance of linear models trained by the LASSO.

section 14.5.2, the absolute value in the LASSO leads to solutions where many of the parameters are set to zero. This *selection* property is useful when we have far too many input features for our model. For example, many genome-wide association studies use logistic regression to compute the risk of a disease given mutations (SNPs) in a genome. Every humans has thousands of SNPs, so any regression model trained with all SNPs would be vastly overfit. A properly regularized logistic regression model will only assign nonzero effect sizes to a small number of informative SNPs, essentially selecting the best SNPs for predicting the risk of the disease. Regularization not only improves generalization; it also helps refine datasets by selecting interesting features.

14.5.2 Generalized Regularization

The LASSO combines regularization and linear regression, but regularization can be applied to any machine learning technique. In general, machine learning algorithms find parameter values that minimize a loss function, so they can be cast as optimization problems of the form

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} L(\boldsymbol{\beta})$$

where $\boldsymbol{\beta}$ is a vector of parameters and L is a loss function that depends on the parameters. Just as we did with the LASSO, we can add a regularization term to the objective function to give the regularized problem

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} L(\boldsymbol{\beta}) + \lambda \left\|\boldsymbol{\beta}\right\|_k .$$

The hyperparameter λ determines how badly we penalize the parameters based on the sum of their k-norms. The LASSO used the 1-norm (the absolute value) of each parameter, but we can use any norm to measure the size of each parameter. Each norm has a different effect on the regularization, as we explain in the following sections.

0-norm Regularization

The 0-norm measures the number of nonzero parameters. The 0-norm of any nonzero value is equal to 1, or

$$\left\|\beta\right\|_0 = \begin{cases} 0, & \beta = 0 \\ 1, & \beta \neq 0 \end{cases} .$$

The 0-norm is not a true norm since it violates some of the defining properties of norms. However, it is useful for "counting" nonzero values. Regularizing with the 0-norm penalizes the number of nonzero parameters, not their magnitudes. A 0-norm regularized model will minimize the loss function using the smallest number of parameters. This regularization strategy mimics what we accomplished in Figure 14.2 by changing the number of parameters in our curvilinear models. Changing the model from six to two parameters simplified the model but did not place any constraints on the values of the parameters.

The 0-norm would be excellent for regularization except for one problem — it is computationally intractable. Problems that include a 0-norm are essentially discrete problems where parameters are either "on" or "off". Such problems are combinatorially complex and their difficulty increases exponentially with the number of parameters. Large-scale machine learning problems would be impossible to solve if they included 0-norm regularization. Fortunately, the 1-norm approximates many of the features of the 0-norm in a computationally efficient way.

1-norm Regularization

As its name implies, the LASSO performs "shrinkage and selection" on the parameters of a linear model. Shrinkage penalizes parameters based on their magnitude, while selection encourages the model to have nonzero values for only a subset of the parameters. Selection is akin to 0-norm regularization, but the LASSO used only the 1-norm, or absolute value, when penalizing the parameters. It turns out that 1-norm regularization performs both operations. Minimizing the absolute value of parameters forces many of the parameters to zero. While there is no guarantee that minimizing the 1-norm will achieve the least number of nonzero parameters (as the 0-norm would), the 1-norm provides a decent approximation of the 0-norm. At the same time, the 1-norm also penalizes parameters based on their magnitudes, all while remaining computationally tractable!

The combination of shrinkage, selection, and efficiency make the 1-norm a popular choice for regularization. It is ideal for generating *sparse* solution, i.e. models with relatively few nonzero parameters. Despite its strengths, the 1-norm has two disadvantages. First, the derivative of the absolute value is discontinuous at zero, so 1-norm regularized problems can be difficult to solve by gradient descent. Alternative methods like coordinate descent or stochastic gradient descent (Chapter **??**) can be used instead. Second, the solutions to 1-norm regularized problems are not unique. Many different parameter sets can have the same 1-norm penalty.

2-norm Regularization

The 2-norm, like the 1-norm, penalizes based on the magnitude of the model's parameters; however, this is where the similarities end. The 2-norm penalty increases quadratically with the size of a parameter, so 2-norm regularization tries very hard to avoid any large parameters. Instead, the 2-norm encourages solutions with many nonzero parameters with small magnitudes. This is the opposite of the sparse solutions produced by 1-norm regularization. Problems with the 2-norm have unique solutions and continuous derivatives (provided the loss function is continuously differentiable). Both of these features are attractive for large-scale optimization.

Regularization with the 2-norm is also called least-squares or Tikhonov regularization.

The Elastic Net: 1-norm + 2-norm Regularization

Some algorithms try to combine the desirable features of both 1-norm and 2-norm regularization. Both regularization terms are added to the objective function.

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} L(\boldsymbol{\beta}) + \lambda_1 \left\|\boldsymbol{\beta}\right\|_1 + \lambda_2 \left\|\boldsymbol{\beta}\right\|_2$$

Each type of regularization is weighted by separate hyperparameter: λ_1 for the 1-norm and λ_2 for the 2-norm. The relative size of these hyperparameters determines the importance of each type of regularization. Setting $\lambda_1 = 0$ is equivalent to 2-norm regularization, and setting $\lambda_2 = 0$ performs 1-norm regularization. Combining both 1-norm and 2-norm regularization is sometimes called *Elastic Net* regularization.

Chapter 15

Geometry

15.1 Geometry of Linear Equations

Why do linear systems have convex solution spaces? Before answering, we should understand the shape of individual equations (rows) in the systems $\mathbf{A}\mathbf{x} = \mathbf{b}$. The equation corresponding to the i^{th} row is

$$\mathbf{A}(i, :) \cdot \mathbf{x} = b_i$$

which we will simplify by using the notation

$$\mathbf{a} \cdot \mathbf{x} = b$$

where $\mathbf{a}$ and $\mathbf{x}$ are vectors and the value b is a scalar. In two dimensions this expression defines a line

$$a_1 x_1 + a_2 x_2 = b.$$

The above representation of a line is the *standard form*, which differs from the *slope-intercept* form you remember from algebra ($y = mx + b$). It seems intuitive why the slope-intercept form is a line; a change in x produces a corresponding change $m\Delta x$ in y, with an intercept b when $x = 0$. What is the analogous reasoning for why $\mathbf{a} \cdot \mathbf{x} = b$ is a line?

First, we note that the vector $\mathbf{a}$ always point perpendicular, or *normal* to the line. For the horizontal line $y = 3$, the vector $\mathbf{a} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ points vertically. For the vertical line $x = 3$, the vector $\mathbf{a} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ point horizontal. For the line $x_1 + x_2 = 1$, $\mathbf{a} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, which is still perpendicular to the original line.

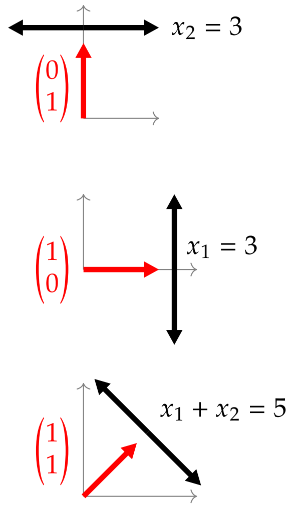

$x_2 = 3$
$\begin{pmatrix} 0 \\ 1 \end{pmatrix}$
$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$
$x_1 = 3$
$\begin{pmatrix} 1 \\ 1 \end{pmatrix}$
$x_1 + x_2 = 5$

Figure 15.1: The vector $\mathbf{a}$ is always normal (perpendicular) to the line $\mathbf{a} \cdot \mathbf{x} = b$.

To help visualize the equation $\mathbf{a} \cdot \mathbf{x} = b$, we need to know the length of $\mathbf{a}$. The easiest solution is to normalize $\mathbf{a}$ by dividing both sizes of the equation by the norm of $\mathbf{a}$.

$$\frac{1}{\|\mathbf{a}\|}\mathbf{a} \cdot \mathbf{x} = b / \|\mathbf{a}\| \, .$$

If we use our previous notation of $\hat{\mathbf{a}}$ for the normalized form of $\mathbf{a}$ and define scalar $d = b/\|\mathbf{a}\|$, we have

$$\hat{\mathbf{a}} \cdot \mathbf{x} = d.$$

The equation $\hat{\mathbf{a}} \cdot \mathbf{x} = d$ is called the *Hesse normal form* of a line, plane, or hyperplane. We know that $\hat{\mathbf{a}}$ is a unit vector normal to the line. What is the meaning of d? Let's compute the dot product $\hat{\mathbf{a}} \cdot \mathbf{x}$ using an arbitrary point $\mathbf{x}$ on the line.

$$d = \hat{\mathbf{a}} \cdot \mathbf{x} = \|\hat{\mathbf{a}}\| \, \|\mathbf{x}\| \cos\theta = \|\mathbf{x}\| \cos\theta.$$

Thus, d is the projection of the magnitude of $\mathbf{x}$ onto the normal vector. For any point on the line, this projection is always the same length – the distance between the origin and the nearest point on the line. Conversely, a line is the set of all vectors whose projection against a vector $\hat{\mathbf{a}}$ is a constant distance (d) from the origin.

The same interpretation follows in higher dimensions. In 3D, the expression $\hat{\mathbf{a}} \cdot \mathbf{x} = d$ defines a plane with normal vector $\hat{\mathbf{a}}$ at a distance d from the origin. This definition fits with the algebraic definition of a plane that you may have seen previously: $a_1x_1 + a_2x_2 + a_3x_3 = d$. In higher dimensions (four or more), this construct is called a *hyperplane*.

Remember that when analyzing an expression of the form $\hat{\mathbf{a}} \cdot \mathbf{x} = d$, the constant on the right hand side (d) is only equal to the distance between the line and the origin if the vector $\hat{\mathbf{a}}$ is normalized. For example, the line

$$3x_1 + 4x_2 = 7$$

has coefficient vector $\mathbf{a} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$, which is not normalized. To normalize $\mathbf{a}$, we divide both sides by $\|\mathbf{a}\| = \sqrt{3^2 + 4^2} = 5$, yielding

$$\frac{3}{5}x_1 + \frac{4}{5}x_2 = \frac{7}{5}.$$

Now we can say that the distance between this line and the origin is 7/5.

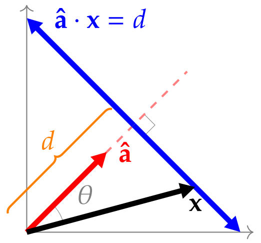


Figure 15.2: A line is the set of all points $\mathbf{x}$ whose projection onto $\hat{\mathbf{a}}$ is the distance d.

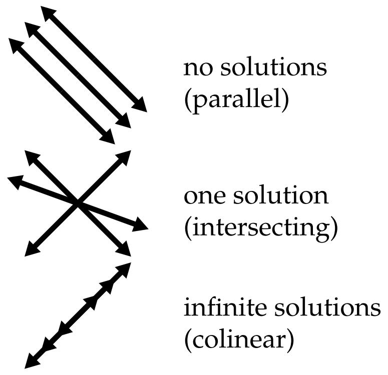


Figure 15.3: A system of linear equations can have zero, one, or infinitely many points of intersection.

15.2 Geometry of Linear Systems

The equation $\hat{\mathbf{a}} \cdot \mathbf{x} = d$ defines a hyperplane. It is also a single row in the linear system $\mathbf{Ax} = \mathbf{b}$. What does the entire system of equations look like? First, let's consider a set of three equations in two dimensions (so we can visualize them as lines). Solutions to $\mathbf{Ax} = \mathbf{b}$ are points of intersection of all three equations. If the lines are parallel, no solutions exist. If the lines all intersect at one point, there is a unique solution. If the lines are *colinear* (all fall upon the same line), infinitely many solutions exist. Note that these are the only options – zero, one, or infinitely many solutions, just as predicted by the grand solvability theorem. It is impossible to draw three straight lines that intersect in only two places.

If linear systems $\mathbf{Ax} = \mathbf{b}$ are a set of intersecting lines in 2D, what is do the inequalities $\mathbf{Ax} \le \mathbf{b}$ represent? Each inequality states that the projection of $\mathbf{x}$ onto the normal vector $\mathbf{a}$ must be less than d. These points form a *half-plane* – all the points on one side of a hyperplane. The system $\mathbf{Ax} \le \mathbf{b}$ has a solution space that is the overlap of multiple half-planes (one for each row in $\mathbf{A}$). As we proved earlier, this solution set is a convex set.

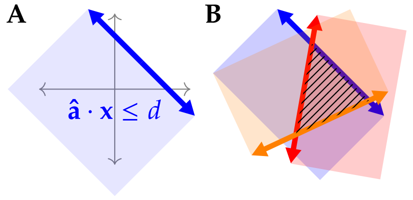


Figure 15.4: **A**. One inequality defines a half-plane. **B**. Multiple half-planes intersect to form a convex solution set for the system $\mathbf{Ax} \le \mathbf{b}$.

Chapter 16

Support Vector Machines

In this chapter we focus on *classification*, the problem of using *features* to predict which *class* an observation belongs to. We are particularly interested in distinguishing among two classes, a problem known as binary classification. We will introduce the Support Vector Machine, or SVM, a framework for solving classification problems using optimization and linear algebra.

"Machine" refers to an algorithm. We'll explain what a support vector is later in the chapter.

As an example, consider the following dataset that reports the blood pressure and cholesterol levels of 20 patients. Twelve of the patients have not experienced a heart attack, but the remaining eight have. Let's load and plot the data.

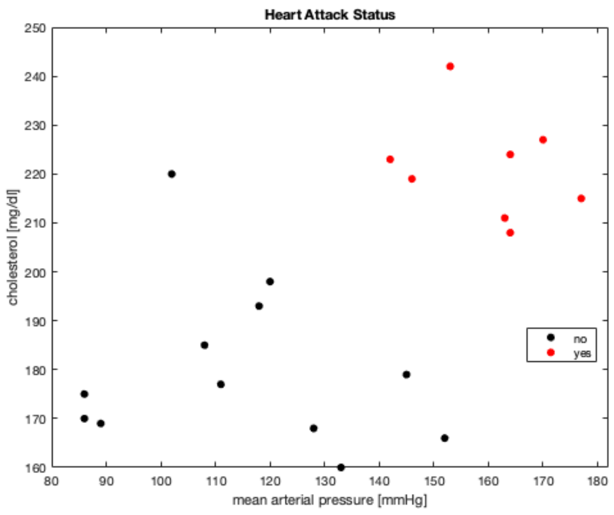
Matlab code

```
load HeartAttack.mat
hatk
```

	BloodPressure	Cholesterol	HeartAttack
1	133	160	'no'
2	152	166	'no'
3	128	168	'no'
4	89	169	'no'
5	86	170	'no'
6	86	175	'no'
7	111	177	'no'
8	145	179	'no'
9	108	185	'no'
10	118	193	'no'

Matlab code

```
gscatter(hatk.BloodPressure,hatk.Cholesterol, ...
         hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
title('Heart Attack Status')
hold off
```



It's clear that we can separate the patients who experienced a heart attack from the ones who did not. However, the separation requires knowledge of both blood pressure and cholesterol levels. There is no cholesterol level alone that separates the two classes, and the same is true for blood pressure. We want to identify a hyperplane that separates the classes so we can predict the heart attack risk for other patients.

For small datasets like this, it is possible to simply draw a line that separates the classes. For problems with only two features, classification is often trivial. However, classifying with thousands of features cannot be done *ad hoc*. Fortunately, everything we will learn in lower dimensions generalizes easily to higher dimensions.

16.1 Separating Hyperplanes

First we *code* the points by setting one group equal to +1 and the other group to −1. For our heart attack data, patients who experienced a heart attack are +1 and the rest are −1.

The term *code* in this context is unrelated to computer programming.

Matlab code

```
hatk.HeartAttack = [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
                     1  1  1  1  1  1  1  1]'
```

	BloodPressure	Cholesterol	HeartAttack
1	133	160	-1
2	152	166	-1
3	128	168	-1
4	89	169	-1
5	86	170	-1
6	86	175	-1
7	111	177	-1
8	145	179	-1
9	108	185	-1
10	118	193	-1

The +1 and −1 designations are arbitrary — it doesn't matter which group is which. Switching the +1 and −1 codings will give the same classifer. The resulting hyperplane will have the normal vector pointing the opposite way, but this does not affect performance of the classifier.

Our goal is to find a hyperplane that separates the +1 and −1 points. Recall that any hyperplane can be represented in the Hesse form as

This is the Hesse form, not the Hesse normal form since **a** has not been normalized.

$$\mathbf{a} \cdot \mathbf{x} = b$$

where **a** is a vector of coefficients and b is a scalar. Normally our goal is to find values for the vector **x**. For classification problems we know that values of **x** (the features) for each point. Our goal is to find the coefficients **a** and the scalar b that define the separating hyperplane.

We want to choose **a** and b such that all of the +1 points are on one side of the hyperplane and all of the −1 points lie on the other size. By convention, we will put the +1 points above the plane, which we enforce with the constraint

$$\mathbf{a} \cdot \mathbf{x} \geq b$$

for any values $\mathbf{x}$ that are coded +1. Similarly, we require the −1 points be below the hyperplane with the constraint

$$\mathbf{a} \cdot \mathbf{x} \leq b$$

for any values $\mathbf{x}$ that are coded −1. Note that there are usually infinitely many hyperplanes that can separate the +1 and −1 points. We want to find the hyperplane that maximizes the gap, or *margin*, between the +1 and −1 points. The hyperplane that maximizes this gap is called the *maximal margin hyperplane*.

To find the maximal margin hyperplane, we start with two parallel hyperplanes. We require all the +1 points be above the top plane and all −1 points be below the bottom plane. We push the two planes apart until the top plane hits the nearest +1 point and the bottom plane hits the nearest −1 point. When the gap between the two planes is maximized, we know that the maximal margin hyperplane will sit halfway in between the two planes.

Let's formalize this procedure. We define the top plane to be $\mathbf{a} \cdot \mathbf{x} = b + 1$ and the bottom plane to be $\mathbf{a} \cdot \mathbf{x} = b - 1$. Since both planes have the same normal vector $\mathbf{a}$ we know they are parallel. The ±1 terms are arbitrary since we aren't requiring the vector $\mathbf{a}$ to be a unit vector. How far apart are these planes? Let's convert the planes to the Hess normal form. Then the top plane is at a distance of $(b + 1)/\|\mathbf{a}\|$ from the origin and the bottom plane is at distance $(b - 1)/\|\mathbf{a}\|$. The distance between the planes is therefore

$$\frac{b+1}{\|\mathbf{a}\|} - \frac{b-1}{\|\mathbf{a}\|} = \frac{2}{\|\mathbf{a}\|}$$

The gap between the planes is inversely proportional to the magnitude of $\mathbf{a}$. Maximizing the separation between the planes is equivalent to minimizing the magnitude of $\mathbf{a}$. All together, the maximal margin hyperplane is the solution to the following constrained optimization problem:

$$\begin{aligned} \underset{\mathbf{a},b}{\text{minimize}} \quad & \|\mathbf{a}\| \\ \text{subject to} \quad & \mathbf{a} \cdot \mathbf{x} \geq b + 1 \quad \text{for the +1 points} \\ & \mathbf{a} \cdot \mathbf{x} \leq b - 1 \quad \text{for the −1 points} \end{aligned}$$

This might seem like a difficult optimization, but there is an important simplification. Remember the definition of the magnitude

$$\|\mathbf{a}\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$$

The square root function is *monotonically increasing*, meaning it always increases as its argument increases. Because of monotonicity, minimizing the square root of an

Functions like $\cos(x)$ are not monotonic as they both increase and decrease as x increases.

input is equivalent to minimizing the input itself. Rather than minimize $\|\mathbf{a}\|$ we can simply minimize the expression $a_1^2 + a_2^2 + \cdots + a_n^2$. The classification problem becomes

$$\begin{aligned} \underset{\mathbf{a},b}{\text{minimize}} \quad & a_1^2 + a_2^2 + \cdots + a_n^2 \\ \text{subject to} \quad & \mathbf{a} \cdot \mathbf{x} \geq b + 1 \qquad \text{for the +1 points} \\ & \mathbf{a} \cdot \mathbf{x} \leq b - 1 \qquad \text{for the −1 points} \end{aligned}$$

Since the objective is purely quadratic with positive coefficients, we know it is convex. We also know that the constraints are linear and therefore also convex. We are minimizing a convex function over a convex set, which is easily solved.

16.2 Setting up the SVM Quadratic Program

The SVM problem outlined above is a *quadratic program (QP)*, a term in optimization that means a problem with a quadratic objective function and a set of linear constraints. Let's set up a QP for four observations from our heart attack data:

Blood Pressure	Cholesterol	HeartAttack
133	160	−1
89	169	−1
164	224	+1
153	242	+1

1. Define the dimensions of the problem. We have two predictor variables: blood pressure and cholesterol level. Let's set x_1 = blood pressure and x_2 = cholesterol. We then know that **a** has two dimensions (a_1 and a_2).

2. Write out the objective function. The objective is to minimize the magnitude of **a**, or
$$\underset{a_1,a_2,b}{\text{minimize}}\; a_1^2 + a_2^2$$

3. Write out constraints for each point by substituting values for **x**. We have four points so we will have four constraints. The constraints for the −1 points are
$$\begin{aligned} 133a_1 + 160a_2 &\leq b - 1 \\ 89a_1 + 169a_2 &\leq b - 1 \end{aligned}$$

For the +1 points we have

$$\begin{aligned} 164a_1 + 224a_2 &\geq b + 1 \\ 153a_1 + 242a_2 &\geq b + 1 \end{aligned}$$

All together, the quadratic program for finding the SVM classifier for these data is

$$\begin{aligned} \underset{a_1, a_2, b}{\text{minimize}} \quad & a_1^2 + a_2^2 \\ \text{subject to} \quad & 133a_1 + 160a_2 \leq b - 1 \\ & 89a_1 + 169a_2 \leq b - 1 \\ & 164a_1 + 224a_2 \geq b + 1 \\ & 153a_1 + 242a_2 \geq b + 1 \end{aligned}$$

16.3 SVM in Matlab

Setting up an SVM problem by hand is informative but unwieldy for large datasets. There are several software libraries for efficiently formulating and solving SVM problems. We will use the `fitcsvm` function to fit an SVM classifier. The function takes two arguments: a matrix of features and a vector with class codings. Let's begin by putting our two features into a matrix.

The name `fitcsvm` stands for "fit classifier SVM".

Matlab code

```
features = [hatk.BloodPressure hatk.Cholesterol]
```

Matlab output

```
features = 20x2
   133   160
   152   166
   128   168
    89   169
    86   170
    86   175
   111   177
   145   179
   108   185
   118   193
```

Now we call `fitcsvm` and store the output in a variable that we'll call `model`.

Matlab code

```
model = fitcsvm(features, hatk.HeartAttack)
```

Matlab output

```
model =
  ClassificationSVM
             ResponseName: 'Y'
    CategoricalPredictors: []
               ClassNames: [-1 1]
           ScoreTransform: 'none'
          NumObservations: 20
                    Alpha: [3x1 double]
                     Bias: -16.4864
         KernelParameters: [1x1 struct]
           BoxConstraints: [20x1 double]
          ConvergenceInfo: [1x1 struct]
          IsSupportVector: [20x1 logical]
                   Solver: 'SMO'
```

The model object contains lots of information. Some important pieces are the values for **a** (`model.Beta`) and value of the scalar b (`model.Bias`)

Matlab code

```
model.Beta
```

Matlab output

```
ans = 2x1
    0.0465
    0.0488
```

Matlab code

```
model.Bias
```

Matlab output

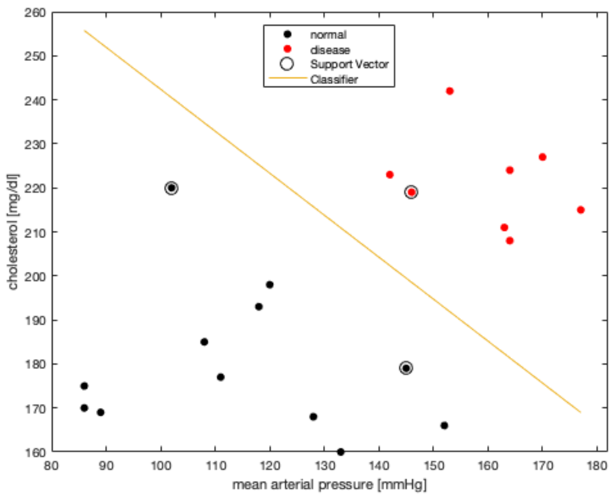
```
ans = -16.4864
```

We can use these values to plot the maximal margin hyperplane.

Matlab code

```
bp = hatk.BloodPressure; ch = hatk.Cholesterol;
gscatter(bp,ch,hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
plot(bp(model.IsSupportVector), ...
     ch(model.IsSupportVector), ...
     'ko', 'MarkerSize',10);
```

```
plot(bp, ...
     -model.Beta(1)/model.Beta(2)*bp ...
         - (model.Bias)/model.Beta(2))
legend('normal','disease','Support Vector','Classifier');
hold off
```



We've also identified the *support vectors* on the above plot. Remember that to find the maximal margin hyperplane we push two parallel planes outward until they hit the nearest +1 and −1 points. These nearest points are called the support vectors since they "support" the planes. Support vectors are what determine the location of the maximal margin hyperplane; their importance gives rise to the name Support Vector Machine.

So far we've trained an SVM model. We can also make predictions about new patients using the model object and the Matlab function `predict`. The `predict` function accepts a model object and a matrix of features for the unknown observations. It returns predictions (+1 or −1) for each observation. Let's make predictions for two new patients with the following data:

Blood Pressure	Cholesterol
153	230
99	132

Matlab code

```
predict(model, [153 230; 99 132])
```

Matlab output

```
ans = 2x1
     1
    -1
```

Our model predicts the first patient would have a history of heart attack while the second patient would not.

16.4 *k*-fold Cross Validation in Matlab

Performing a *k*-fold cross validation requires 1.) randomizing the folds, 2.) retraining the model, and 3.) classifying each fold. Fortunately, there is a Matlab function to perform *k*-fold cross validation. We can perform a 5-fold cross validation on our heart attack SVM model as follows

Matlab code

```
xval = crossval(model,'Kfold',5)
```

Matlab output

```
xval =
  classreg.learning.partition.ClassificationPartitionedModel
    CrossValidatedModel: 'SVM'
         PredictorNames: {'x1'  'x2'}
           ResponseName: 'Y'
        NumObservations: 20
                  KFold: 5
              Partition: [1x1 cvpartition]
             ClassNames: [-1 1]
         ScoreTransform: 'none'
```

The object returned by `crossval` contains information about how the folds were created and tested. The accuracy of the classifier is measured by the *loss*, with lower loss meaning a better model. We can find the loss by calling the `kfoldLoss` function on our `crossval` return object.

Note the capital "L" in `kfoldLoss`.

Matlab code

```
kfoldLoss(xval)
```

Matlab output

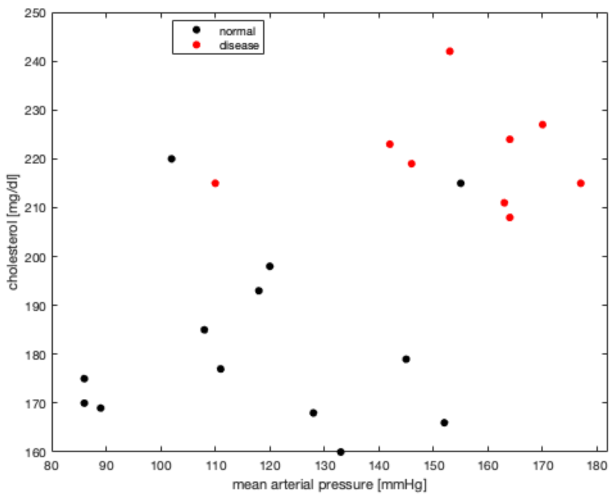
```
ans = 0
```

16.5 Soft Classifiers

Our heart attack data was perfectly classifiable since we could cleanly separate the +1 and −1 classes. This is not always the case, especially for biological datasets. Let's add two points to our dataset and replot the data.

MATLAB code

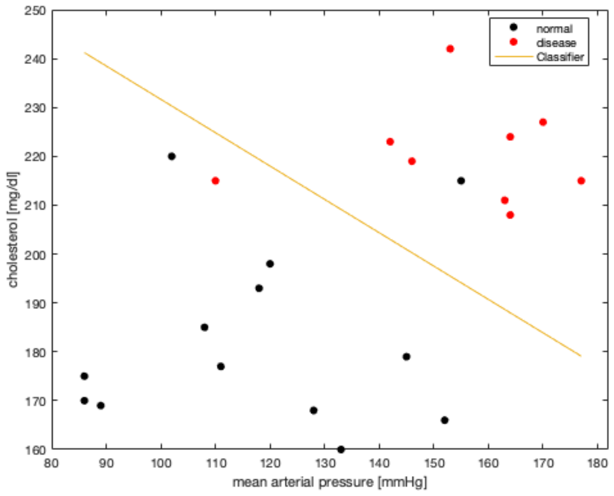
```
hatk(end+1,:) = {155,215,-1};
hatk(end+1,:) = {110,215, 1};
gscatter(hatk.BloodPressure,hatk.Cholesterol, ...
         hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
legend('normal','disease')
hold off
```



With the new data, it doesn't appear that we can perfectly separate the heart attacks from the rest. Let's try to refit our model.

MATLAB code

```
mdl = fitcsvm([hatk.BloodPressure hatk.Cholesterol], ...
              hatk.HeartAttack);
gscatter(hatk.BloodPressure,hatk.Cholesterol, ...
        hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
plot(hatk.BloodPressure, ...
     -mdl.Beta(1)/mdl.Beta(2)*hatk.BloodPressure ...
         - (mdl.Bias)/mdl.Beta(2))
legend('normal','disease','Classifier');
hold off
```



Now we have some points that sit on the wrong side of the classifier. Our accuracy should have decreased (i.e. out loss during cross validation should increase).

Matlab code

```
xval = crossval(mdl,'Kfold',5);
kfoldLoss(xval)
```

Matlab output

```
ans = 0.0909
```

The SVM formulation we described above is called a *hard classifier* since it requires that all points be on the correct side of the classifier. In practice, SVM software packages use a *soft classifier* where points can appear on the wrong side.

When solving SVM problems with soft classifiers, the goal is to both maximize the separation and minimize the number of incorrectly classified points. We will not discuss the mathematical formulation of soft classifiers in this book.

Part III

High-Dimensional Systems

Chapter 17

Vector Spaces, Span, and Basis

17.1 Vector Spaces

Vector spaces are collections of vectors. The most common spaces are $\mathbb{R}^2$, $\mathbb{R}^3$, and $\mathbb{R}^n$ — the spaces that include all 2-, 3-, and n-dimensional vectors. We can construct *subspaces* by specifying only a subset of the vectors in a space. For example, the set of all 3-dimensional vectors with only integer entries is a subspace of $\mathbb{R}^3$.

Remember that $\mathbb{R}^2$ is not a subspace of $\mathbb{R}^3$; they are completely separate, non-overlapping spaces.

17.2 Span

A set of m vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m$ is said to *span* a space V if any vector $\mathbf{u}$ in V can be written as a linear combination of the vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m$. This is equivalent to saying there exist scalars $a_1, a_2, \ldots, a_m$ such that

$$\mathbf{u} = a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_m\mathbf{v}_m.$$

Writing a vector $\mathbf{u}$ as a linear combination of $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m$ is called *decomposing* $\mathbf{u}$ over $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m$. If a set of vectors spans a space, they can be used to decompose any other vector in the space.

We've already seen vector composition using a special set of vectors $\hat{\mathbf{e}}_j$, the unit vectors with only one nonzero entry at element j. For example, the vector

$$\begin{pmatrix} -2 \\ 4 \\ 5 \end{pmatrix} = -2\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + 4\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + 5\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Thus the vectors $\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2, \hat{\mathbf{e}}_3$ spans $\mathbb{R}^3$. In general, the set of vectors $\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2, \ldots, \hat{\mathbf{e}}_n$ spans the space $\mathbb{R}^n$. Are these the only sets of vectors that span these spaces?

No, there are infinitely many sets of vectors that span each space. Consider the vectors $\begin{pmatrix}1\\1\end{pmatrix}$ and $\begin{pmatrix}-1\\1\end{pmatrix}$. We can show that these vectors span $\mathbb{R}^2$ by showing that any vector $\mathbf{u}$ in $\mathbb{R}^2$ can be written as a linear combination

$$\mathbf{u} = \begin{pmatrix}u_1\\u_2\end{pmatrix} = a_1\begin{pmatrix}1\\1\end{pmatrix} + a_2\begin{pmatrix}-1\\1\end{pmatrix}.$$

Finding the coefficients a_1 and a_2 is akin to solving the system of linear equations

$$\begin{aligned} a_1 - a_2 &= u_1 \\ a_1 + a_2 &= u_2 \end{aligned}$$

which has the unique solution

$$a_1 = \frac{u_1+u_2}{2}, \quad a_2 = \frac{u_2-u_1}{2}.$$

To demonstrate, let $\mathbf{u} = \begin{pmatrix}-2\\4\end{pmatrix}$. Then $a_1 = 1$ and $a_2 = 3$, so

$$a_1\begin{pmatrix}1\\1\end{pmatrix} + a_2\begin{pmatrix}-1\\1\end{pmatrix} = \begin{pmatrix}1\\1\end{pmatrix} + 3\begin{pmatrix}-1\\1\end{pmatrix} = \begin{pmatrix}1-3\\1+3\end{pmatrix} = \begin{pmatrix}-2\\4\end{pmatrix}.$$

We've shown that there are least two sets of vectors that span $\mathbb{R}^2$, $\left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\}$ and $\left\{\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}-1\\1\end{pmatrix}\right\}$. How can we say there are infinitely many? If vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ span a space V, then the vectors $k_1\mathbf{v}_1$ and $k_2\mathbf{v}_2$ also span V for any scalars k_1 and k_2. To prove this, remember that any vector $\mathbf{u}$ can be decomposed onto $\mathbf{v}_1$ and $\mathbf{v}_2$, i.e.

$$\begin{aligned} \mathbf{u} &= a_1\mathbf{v}_1 + a_2\mathbf{v}_2 \\ &= \frac{a_1}{k_1}(k_1\mathbf{v}_1) + \frac{a_2}{k_2}(k_2\mathbf{v}_2) \end{aligned}$$

Since (a_1/k_1) and (a_2/k_2) are simply scalars, we've shown that $\mathbf{u}$ can be decomposed onto the vectors $k_1\mathbf{v}_1$ and $k_2\mathbf{v}_2$. Therefore, $k_1\mathbf{v}_1$ and $k_2\mathbf{v}_2$ must also span $\mathbb{R}^2$. For example, the vectors $\begin{pmatrix}3\\0\end{pmatrix}$ and $\begin{pmatrix}0\\-1/2\end{pmatrix}$ are scalar multiples of $\begin{pmatrix}1\\0\end{pmatrix}$ and $\begin{pmatrix}0\\1\end{pmatrix}$. The

Since k_1 and k_2 are arbitrary, this allows us to generate infinitely many sets of vectors that span any space from a single spanning set.

former two vectors must therefore span $\mathbb{R}^2$, so we can decompose the vector $\begin{pmatrix} -2 \\ 4 \end{pmatrix}$ onto them:

$$\begin{pmatrix} -2 \\ 4 \end{pmatrix} = -\frac{2}{3}\begin{pmatrix} 3 \\ 0 \end{pmatrix} - 8\begin{pmatrix} 0 \\ 1/2 \end{pmatrix}.$$

Similarly, if $\mathbf{v}_1$ and $\mathbf{v}_2$ span a space V, the vectors $\mathbf{v}_1$ and $(\mathbf{v}_1 + \mathbf{v}_2)$ also span V:

$$\begin{aligned} \mathbf{u} &= a_1\mathbf{v}_1 + a_2\mathbf{v}_2 \\ &= a_1\mathbf{v}_1 + a_2(\mathbf{v}_1 + \mathbf{v}_2) - a_2\mathbf{v}_1 \\ &= (a_1 - a_2)\mathbf{v}_1 + a_2(\mathbf{v}_1 + \mathbf{v}_2) \end{aligned}$$

If scalars a_1 and a_2 decompose $\mathbf{u}$ over $\mathbf{v}_1$ and $\mathbf{v}_2$, then $(a_1 - a_2)$ and a_2 decompose $\mathbf{u}$ over $\mathbf{v}_1$ and $(\mathbf{v}_1 + \mathbf{v}_2)$.

17.3 Review: Linear Independence

We said before that vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ are *linearly independent* if and only if

$$a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n = \mathbf{0}$$

implies that all coefficients $a_1, a_2, \ldots, a_n$ are zero. No linear combination of a set of linearly independent vectors can be the zero vector except for the trivial case where all the coefficients are zero. We often say that a set of vectors are linearly dependent if one of the vectors can be written as a linear combination of the others, i.e.

$$\mathbf{v}_i = a_1\mathbf{v}_1 + \cdots + a_{i-1}\mathbf{v}_{i-1} + a_{i+1}\mathbf{v}_{i+1} + \cdots + a_n\mathbf{v}_n.$$

Moving the vector $\mathbf{v}_i$ to the right hand side

$$\mathbf{0} = a_1\mathbf{v}_1 + \cdots + a_{i-1}\mathbf{v}_{i-1} - \mathbf{v}_i + a_{i+1}\mathbf{v}_{i+1} + \cdots + a_n\mathbf{v}_n$$

we see 1.) a linear combination of the vectors sums to the zero vector on the left, and 2.) at least one of the coefficients (the -1 in front of $\mathbf{v}_i$) is nonzero. This is consistent with are above definition of linear independence. We said these vectors were not linearly independent, so it is possible for a linear combination to sum to zero using at least one nonzero coefficient.

17.4 Basis

The concepts of span and linear independence are a powerful combination. Any linearly independent set of vectors that span a space V are called a *basis* for V.

Any vector in a space be decomposed over a set of vectors that span the space. **However, every vector in a space has a unique decomposition over an associated basis.** Said another way, for every vector in a space, there are only one set of coefficients $a_1, a_2, \ldots, a_n$ that decompose it over a basis $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$.

We can prove that a decomposition over a basis is unique by contradiction. Suppose there were two sets of coefficients – $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ – that decomposed a vector $\mathbf{u}$ over a basis $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$. Then

$$\mathbf{u} = a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n = b_1\mathbf{v}_1 + b_2\mathbf{v}_2 + \cdots + b_n\mathbf{v}_n.$$

We can move all the right hand size over to the left and group terms to give

$$(a_1 - b_1)\mathbf{v}_1 + (a_2 - b_2)\mathbf{v}_2 + \cdots + (a_n - b_n)\mathbf{v}_n = \mathbf{0}.$$

Remember that $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ is a basis, so the vectors must be linearly independent. By the definition of linear independence, the only way the above equation can be true is if all the coefficients are zero. This implies that $a_1 = b_1$, $a_2 = b_2$, and so on. Clearly this is a violation of our original statement that $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots b_n$ where different. Therefore, there can only be one way to decompose any vector onto a basis.

17.4.1 Testing if vectors form a basis

Every basis for a vector space has the same number of vectors. This number is called the *dimension* of the vector space. For standard vectors spaces like $\mathbb{R}^n$, the dimension is n. The dimension of $\mathbb{R}^2$ is 2, and the dimension of $\mathbb{R}^3$ is 3.

Most people think of dimension as the number of elements in a vector. While the true definition of dimension is the number of vectors in the basis, counting elements in a vector works for spaces like $\mathbb{R}^n$. To see why, remember that the Cartesian unit vectors $\hat{\mathbf{e}}_i$ form a basis for $\mathbb{R}^n$, but we need n of these vectors, one per element.

Any set of vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ is a basis for a space V if and only if:

1. The number of vectors (n) matches the dimension of V.
2. The vectors span V.
3. The vectors are linearly independent.

Proving any two of the above statements automatically implies the third is true.

We get to choose which two of the above three statements to prove when verifying that $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ is a basis. The first statement is usually trivial — does the number of vectors match the dimension? We almost always choose to prove the first statement. Proving that vectors are linearly independent is always easier that proving the vectors span the space. If we collect the vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ into a matrix, the rank of this matrix should be n if the vectors are linearly independent.

17.4.2 Decomposing onto a basis

How do we decompose a vector onto a basis? Remember that decomposing $\mathbf{u}$ over $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ is equivalent to finding a set of coefficients $a_1, a_2, \ldots, a_n$ such that

$$a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n = \mathbf{u}.$$

Let's collect the vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ into a matrix $\mathbf{V}$, where each vector $\mathbf{v}_i$ is the ith column in $\mathbf{V}$. Then

$$\begin{pmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_n \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \mathbf{V}\mathbf{a} = \mathbf{u}.$$

We see that finding the coefficients $a_1, a_2, \ldots, a_n$ that decompose a vector $\mathbf{u}$ onto a basis $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ is equivalent to solving the linear system $\mathbf{V}\mathbf{a} = \mathbf{u}$.

By formulating vector decomposition as a linear system, we can easily see why the decomposition of a vector over a basis is unique. In $\mathbb{R}^n$, the basis contains n vectors, each with n elements. So, the matrix $\mathbf{V}$ is a square, $n \times n$ matrix. Since the vectors in the basis (and therefore the columns in $\mathbf{V}$) are linearly independent, the matrix $\mathbf{V}$ has full rank. Thus, the solution to $\mathbf{V}\mathbf{a} = \mathbf{u}$ must be unique, implying that the decomposition of every vector onto a basis is unique.

Since $\mathbf{V}$ is square and full rank, its inverse ($\mathbf{V}^{-1}$) must exist. The system has a unique solution $\mathbf{a} = \mathbf{V}^{-1}\mathbf{u}$.

17.5 Orthogonal and Orthonormal Vectors

A set of vectors is an *orthogonal set* if every vector in the set is orthogonal to every other vector in the set. If every vector in an orthogonal set has been normalized, we say the vectors form an *orthonormal set*. Orthogonal and orthonormal sets are ideal candidates for basis vectors. Since there is no "overlap" among the vectors, we can easily decompose other vectors onto orthogonal basis vectors.

Imagine you have an orthogonal set of vectors you want to use as a basis. We'll assume you have the correct number of vectors (equal to the dimension of your space) for this to be possible. Based on the above requirements for basis vectors, we need only to show that these vectors are linearly independent. If so, they are a basis. For a set of n orthogonal vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$, linear independence requires that

$$a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n = \mathbf{0}$$

if and only if all the coefficients $a_1, a_2, \ldots, a_n$ are equal to zero. Let's take the dot product of both sides of the above equation with the vector $\mathbf{v}_1$:

$$(a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n) \cdot \mathbf{v}_1 = \mathbf{0} \cdot \mathbf{v}_1.$$

On the right hand size, we know that $\mathbf{0} \cdot \mathbf{v}_1 = 0$ for any vector $\mathbf{v}_1$. We also distribute the dot product on the left hand side to give

$$a_1\mathbf{v}_1 \cdot \mathbf{v}_1 + a_2\mathbf{v}_2 \cdot \mathbf{v}_1 + \cdots + a_n\mathbf{v}_n \cdot \mathbf{v}_1 = 0.$$

Since all the vectors are orthogonal, $\mathbf{v}_i \cdot \mathbf{v}_1$ is zero except when $i = 1$. Canceling out all the dot products equal to zero shows that

$$a_1\mathbf{v}_1 \cdot \mathbf{v}_1 = a_1 \|\mathbf{v}_1\|^2 = 0.$$

We know that $\|\mathbf{v}_1\|^2 \neq 0$, so the only way $a_1\mathbf{v}_1 \cdot \mathbf{v}_1$ can equal zero is if a_1 is zero. If we repeat this entire process by taking the dot product with $\mathbf{v}_2$ instead of $\mathbf{v}_1$, we will find that $a_2 = 0$. This continues with $\mathbf{v}_3, \ldots, \mathbf{v}_n$ until we can say that $a_1 = a_2 = \cdots = a_n = 0$. Therefore, if the vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ are an orthogonal (or orthonormal) set, they are linearly independent.

17.5.1 Decomposing onto orthonormal vectors

We saw previously that finding the coefficients to decompose a vector onto a basis requires solving a system of linear equations. For high-dimensional spaces, solving such a system can be computationally expensive. Fortunately, decomposing a vector onto an orthonormal basis is far more efficient.

Theorem. *The decomposition of a vector* $\mathbf{u}$ *onto an orthonormal basis* $\hat{\mathbf{v}}_1, \hat{\mathbf{v}}_2, \ldots, \hat{\mathbf{v}}_n$ *given by*

$$\mathbf{u} = a_1\hat{\mathbf{v}}_1 + a_2\hat{\mathbf{v}}_2 + \cdots + a_n\hat{\mathbf{v}}_n$$

has coefficients

$$\begin{aligned} a_1 &= \mathbf{u} \cdot \hat{\mathbf{v}}_1 \\ a_2 &= \mathbf{u} \cdot \hat{\mathbf{v}}_2 \\ &\vdots \\ a_n &= \mathbf{u} \cdot \hat{\mathbf{v}}_n \end{aligned}$$

We use the symbol $\hat{\mathbf{v}}_i$ for vectors in an orthonormal set to remind us that each vector has been normalized.

Proof. We use a similar strategy as when we proved the linear independence of orthogonal sets. Let's start with the formula for vector decomposition

$$\mathbf{u} = a_1\hat{\mathbf{v}}_1 + a_2\hat{\mathbf{v}}_2 + \cdots + a_n\hat{\mathbf{v}}_n.$$

Taking the dot product of both sides with the vector $\hat{\mathbf{v}}_1$ yields (after distributing the right hand side)

$$\mathbf{u} \cdot \hat{\mathbf{v}}_1 = a_1\hat{\mathbf{v}}_1 \cdot \hat{\mathbf{v}}_1 + a_2\hat{\mathbf{v}}_2 \cdot \hat{\mathbf{v}}_1 + \cdots + a_n\hat{\mathbf{v}}_n \cdot \hat{\mathbf{v}}_1.$$

Because all the vectors $\hat{\mathbf{v}}_1, \hat{\mathbf{v}}_2, \ldots, \hat{\mathbf{v}}_n$ are orthogonal, the only nonzero term on the right hand side is $a_1\hat{\mathbf{v}}_1 \cdot \hat{\mathbf{v}}_1$, so

$$\mathbf{u} \cdot \hat{\mathbf{v}}_1 = a_1\hat{\mathbf{v}}_1 \cdot \hat{\mathbf{v}}_1.$$

By definition of the dot product, $\hat{\mathbf{v}}_1 \cdot \hat{\mathbf{v}}_1 = \|\hat{\mathbf{v}}_1\|^2$. Since $\hat{\mathbf{v}}_1$ is a unit vector, $\|\hat{\mathbf{v}}_1\|^2 = 1$. Thus $a_1 = \mathbf{u} \cdot \hat{\mathbf{v}}_1$. By repeating the same procedure with $\hat{\mathbf{v}}_2$, we find that $a_2 = \mathbf{u} \cdot \hat{\mathbf{v}}_2$, and so on. □

Decomposing vectors over an orthonormal basis is efficient, requiring only a series of dot products to compute the coefficients. For example, we can decompose the vector $\mathbf{u} = \begin{pmatrix} 7 \\ -5 \\ 10 \end{pmatrix}$ over the orthonormal basis $\left\{ \hat{\mathbf{v}}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \hat{\mathbf{v}}_2 = \begin{pmatrix} 0 \\ 3/5 \\ 4/5 \end{pmatrix}, \hat{\mathbf{v}}_3 = \begin{pmatrix} 0 \\ 4/5 \\ -3/5 \end{pmatrix} \right\}$.

$$a_1 = \mathbf{u} \cdot \hat{\mathbf{v}}_1 = \begin{pmatrix} 7 \\ -5 \\ 10 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = 7 + 0 + 0 = 7$$

$$a_2 = \mathbf{u} \cdot \hat{\mathbf{v}}_2 = \begin{pmatrix} 7 \\ -5 \\ 10 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 3/5 \\ 4/5 \end{pmatrix} = 0 - 3 + 8 = 5$$

$$a_3 = \mathbf{u} \cdot \hat{\mathbf{v}}_3 = \begin{pmatrix} 7 \\ -5 \\ 10 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 4/5 \\ -3/5 \end{pmatrix} = 0 - 4 - 6 = -10$$

The decomposition of $\mathbf{u}$ is

$$\mathbf{u} = 7\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + 5\begin{pmatrix} 0 \\ 3/5 \\ 4/5 \end{pmatrix} - 10\begin{pmatrix} 0 \\ 4/5 \\ -3/5 \end{pmatrix} = \begin{pmatrix} 7 + 0 + 0 \\ 0 + 3 - 8 \\ 0 + 4 + 6 \end{pmatrix} = \begin{pmatrix} 7 \\ -5 \\ 10 \end{pmatrix}.$$

17.5.2 Checking an orthonormal set

Given a set of vectors $\hat{\mathbf{v}}_1, \hat{\mathbf{v}}_2, \ldots, \hat{\mathbf{v}}_n$, how can we verify that they are orthonormal? We need to test two things.

1. All vectors are normalized ($\|\hat{\mathbf{v}}_i\| = 1$ for all $\hat{\mathbf{v}}_i$).
2. All vectors are mutually orthogonal ($\hat{\mathbf{v}}_i \cdot \hat{\mathbf{v}}_j = 0$ for all $i \neq j$).

The first test is straightforward. The second can be a little cumbersome, as we need to test all $n^2 - n/2$ pairs of vectors for orthogonality. A simpler, albeit more sometimes more computationally intensive approach, is to collect the vectors into a matrix $\mathbf{V} = \begin{pmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_n \end{pmatrix}$. Then the set of vectors is orthonormal if an only if $\mathbf{V}^{-1} = \mathbf{V}^\mathsf{T}$. While inverting a matrix is "expensive," for small to medium size vector sets this method avoids the need to iterate over all pairs of vectors to test for orthonormality.

Interestingly, the proof of this method (not shown here) reveals that if the columns of $\mathbf{V}$ are an orthonormal set, the rows of $\mathbf{V}$ are also an orthonormal set!

17.5.3 Projections

Our next goal will be to create orthonormal sets of vectors from sets that are not orthogonal. Before introducing such an algorithm, we need to develop a geometric tool — the vector *projection*. The projection of vector $\mathbf{v}$ onto vector $\mathbf{u}$ is a vector that points along $\mathbf{u}$ with length equal to the "shadow" of $\mathbf{v}$ onto $\mathbf{u}$. Previously we used the dot product to calculate the magnitude of the projection of $\mathbf{v}$ onto $\mathbf{u}$, which was a scalar equal to $\|\mathbf{v}\| \cos\theta$, where θ is the angle between $\mathbf{v}$ and $\mathbf{u}$. To calculate the actual projection, we multiply the magnitude of the projection ($\|\mathbf{v}\| \cos\theta$) by a unit vector that points along $\mathbf{u}$. Thus the projection of $\mathbf{v}$ onto $\mathbf{u}$ is defined as

$$\mathrm{proj}_{\mathbf{u}}(\mathbf{v}) = (\|\mathbf{v}\| \cos\theta)\hat{\mathbf{u}}.$$

By definition, $\hat{\mathbf{u}} = \mathbf{u}/\|\mathbf{u}\|$. Also, we note that $\mathbf{v}\cdot\mathbf{u} = \|\mathbf{v}\|\,\|\mathbf{u}\|\cos\theta$, so the expression $\|\mathbf{u}\|\cos\theta$ can be written in terms of the dot product $(\mathbf{v}\cdot\mathbf{u})/\|\mathbf{u}\|$. We can rewrite our formula for the projection using only dot products:

$$\mathrm{proj}_{\mathbf{u}}(\mathbf{v}) = (\|\mathbf{v}\| \cos\theta)\hat{\mathbf{u}} = \frac{\mathbf{v}\cdot\mathbf{u}}{\|\mathbf{u}\|}\frac{\mathbf{u}}{\|\mathbf{u}\|} = \frac{\mathbf{v}\cdot\mathbf{u}}{\|\mathbf{u}\|^2}\mathbf{u} = \frac{\mathbf{v}\cdot\mathbf{u}}{\mathbf{u}\cdot\mathbf{u}}\mathbf{u}.$$

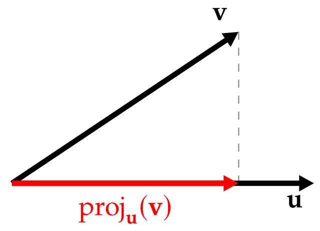


Figure 17.1: The projection is a vector "shadow" of one vector onto another.

We can use the projection to make any two vectors orthogonal, as demonstrated by the following theorem.

Theorem. *Given any vectors* $\mathbf{v}$ *and* $\mathbf{u}$, *the vector* $\mathbf{v}$- $\mathrm{proj}_{\mathbf{u}}(\mathbf{v})$ *is orthogonal to* $\mathbf{u}$.

Proof. If the vector $\mathbf{v} - \mathrm{proj}_{\mathbf{u}}(\mathbf{v})$ is orthogonal to $\mathbf{u}$, the dot product between these vectors must be zero.

$$\begin{aligned}
\left(\mathbf{v} - \mathrm{proj}_{\mathbf{u}}(\mathbf{v})\right)\cdot\mathbf{u} &= \left(\mathbf{v} - \frac{\mathbf{v}\cdot\mathbf{u}}{\mathbf{u}\cdot\mathbf{u}}\mathbf{u}\right)\cdot\mathbf{u} \\
&= \mathbf{v}\cdot\mathbf{u} - \frac{\mathbf{v}\cdot\mathbf{u}}{\mathbf{u}\cdot\mathbf{u}}\mathbf{u}\cdot\mathbf{u} \\
&= \mathbf{v}\cdot\mathbf{u} - \mathbf{v}\cdot\mathbf{u} \\
&= 0
\end{aligned}$$

□

Subtracting the projection of $\mathbf{v}$ onto $\mathbf{u}$ from the vector $\mathbf{v}$ "corrects" $\mathbf{v}$ by removing its overlap with $\mathbf{u}$. The resulting vector is a vector closest to $\mathbf{v}$ that is still orthogonal to $\mathbf{u}$.

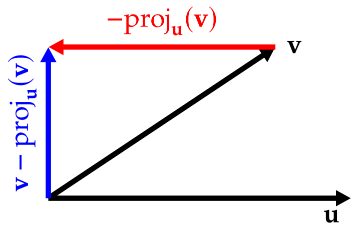

$-\mathrm{proj}_{\mathbf{u}}(\mathbf{v})$
$\mathbf{v}$
$\mathbf{v}-\mathrm{proj}_{\mathbf{u}}(\mathbf{v})$
$\mathbf{u}$

Figure 17.2: Subtracting the projection of $\mathbf{v}$ onto $\mathbf{u}$ from $\mathbf{v}$ makes the vectors orthogonal.

We must begin with linearly independent vectors. Otherwise, orthogonalization will turn one of the vectors into the zero vector, which is not allowed in a basis.

17.5.4 Creating orthonormal basis vectors

We can make any two vectors orthogonal by adjusting one based on its projection onto the other. We can apply these corrections iteratively to make any set of linearly independent vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ into an orthonormal basis set $\hat{\mathbf{u}}_1, \hat{\mathbf{u}}_2, \ldots, \hat{\mathbf{u}}_n$. First, we set

$$\mathbf{u}_1 = \mathbf{v}_1.$$

We leave this first vector unchanged. All other vectors will be made orthogonal to it (and each other). Next, we create $\mathbf{u}_2$ by making $\mathbf{v}_2$ orthogonal to $\mathbf{u}_1$:

$$\mathbf{u}_2 = \mathbf{v}_2 - \mathrm{proj}_{\mathbf{u}_1}(\mathbf{v}_2).$$

Now we have two orthogonal vectors, $\mathbf{u}_1$ and $\mathbf{u}_2$. We continue by creating $\mathbf{u}_3$ from $\mathbf{v}_3$, but this time we must make $\mathbf{v}_3$ orthogonal to both $\mathbf{u}_1$ and $\mathbf{u}_2$:

$$\mathbf{u}_3 = \mathbf{v}_3 - \mathrm{proj}_{\mathbf{u}_1}(\mathbf{v}_3) - \mathrm{proj}_{\mathbf{u}_2}(\mathbf{v}_3).$$

We continue this process for all n vectors, making each vector $\mathbf{v}_i$ orthogonal to all the newly created orthogonal vectors $\mathbf{u}_1, \ldots, \mathbf{u}_{i-1}$. This approach is called the Gram-Schmidt algorithm. Given a set of vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$, we create a set of orthogonal vectors

$$\begin{aligned}
\mathbf{u}_1 &= \mathbf{v}_1 \\
\mathbf{u}_2 &= \mathbf{v}_2 - \mathrm{proj}_{\mathbf{u}_1}(\mathbf{v}_2) \\
\mathbf{u}_3 &= \mathbf{v}_3 - \mathrm{proj}_{\mathbf{u}_1}(\mathbf{v}_3) - \mathrm{proj}_{\mathbf{u}_2}(\mathbf{v}_3) \\
&\vdots \\
\mathbf{u}_i &= \mathbf{v}_i - \mathrm{proj}_{\mathbf{u}_1}(\mathbf{v}_i) - \cdots - \mathrm{proj}_{\mathbf{u}_{i-1}}(\mathbf{v}_i) \\
&\vdots \\
\mathbf{u}_n &= \mathbf{v}_n - \mathrm{proj}_{\mathbf{u}_1}(\mathbf{v}_n) - \cdots - \mathrm{proj}_{\mathbf{u}_{n-1}}(\mathbf{v}_n)
\end{aligned}$$

Said more succinctly,

$$\mathbf{u}_i = \mathbf{v}_i - \sum_{k=1}^{i-1} \mathrm{proj}_{\mathbf{u}_k}(\mathbf{v}_i)$$

The Gram-Schmidt algorithm products an orthogonal set of vectors. To make the set orthonormal, we must subsequently normalize each vector.

Chapter 18

Eigenvectors and Eigenvalues

Consider the matrix

$$\mathbf{A} = \begin{pmatrix} 2 & 7 \\ -1 & -6 \end{pmatrix}.$$

Multiplying $\mathbf{A}$ by the vector $\mathbf{x}_1 = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$ gives an interesting result.

$$\mathbf{A}\mathbf{x}_1 = \begin{pmatrix} 2 & 7 \\ -1 & -6 \end{pmatrix} \begin{pmatrix} -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 5 \\ -5 \end{pmatrix} = -5 \begin{pmatrix} -1 \\ 1 \end{pmatrix} = -5\mathbf{x}_1.$$

Similarly, with $\mathbf{x}_2 = \begin{pmatrix} -7 \\ 1 \end{pmatrix}$:

$$\mathbf{A}\mathbf{x}_2 = \begin{pmatrix} 2 & 7 \\ -1 & -6 \end{pmatrix} \begin{pmatrix} -7 \\ 1 \end{pmatrix} = \begin{pmatrix} -7 \\ 1 \end{pmatrix} = \mathbf{x}_2.$$

In both cases, multiplication by $\mathbf{A}$ returned a scalar multiple of the vector (-5 for $\mathbf{x}_1$ and 1 for $\mathbf{x}_2$). This is not a property of solely the matrix $\mathbf{A}$, since the vector $\mathbf{x}_3 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ is not transformed by a single scalar.

$$\mathbf{A}\mathbf{x}_3 = \begin{pmatrix} 2 & 7 \\ -1 & -6 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 9 \\ 5 \end{pmatrix} \neq \lambda\mathbf{x}_2$$

Similarly, the results we are seeing are not properties of the vectors $\mathbf{x}_1$ and $\mathbf{x}_2$, since they do not become scalar multiples of themselves when multiplied by other matrices.

$$\mathbf{B} = \begin{pmatrix} 2 & 1 \\ -3 & 0 \end{pmatrix}$$

$$\mathbf{B}\mathbf{x}_1 = \begin{pmatrix} 2 & 1 \\ -3 & 0 \end{pmatrix}\begin{pmatrix} -1 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 3 \end{pmatrix} \neq \lambda \mathbf{x}_1$$
$$\mathbf{B}\mathbf{x}_2 = \begin{pmatrix} 2 & 1 \\ -3 & 0 \end{pmatrix}\begin{pmatrix} -7 \\ 1 \end{pmatrix} = \begin{pmatrix} -13 \\ 21 \end{pmatrix} \neq \lambda \mathbf{x}_2$$

The phenomena we're observing is a result of the paring between the matrix **A** and the vectors $\mathbf{x}_1$ and $\mathbf{x}_2$. In general, we see that multiplying a vector by a matrix returns a scalar multiple of the vector, or

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{x}.$$

Any vector **x** that obeys the above relationship is called an *eigenvector* of the matrix **A**. The scalar λ is called the *eigenvalue* associated with the eigenvector **x**. The vector **x** is an eigenvector of the matrix **A**; it is not generally an eigenvector of other matrices.

Eigenvectors were originally called *characteristic* vectors, as they describe the character of the matrix. German mathematicians dropped this nomenclature in favor of the German prefix "eigen-", which mean "own". An eigenvector can be viewed as one of a matrix's "own" vectors since it is not rotated when transformed by multiplication.

In the example above, the matrix $\mathbf{A} = \begin{pmatrix} 2 & 7 \\ -1 & -6 \end{pmatrix}$ has two eigenvectors, $\mathbf{v}_1 = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$ with eigenvalue $\lambda_1 = -5$, and $\mathbf{v}_2 = \begin{pmatrix} -7 \\ 1 \end{pmatrix}$ with eigenvector $\lambda_2 = 1$.

18.1 Properties of Eigenvectors and Eigenvalues

Only square matrices have eigenvectors and eigenvalues. To understand why the matrix must be square, remember that a non-square matrix with m rows and n columns transforms an n-dimensional vectors into an m-dimensional vector. Clearly, the m-dimensional output cannot be the n-dimensional input multiplied by a scalar!

An n by n matrix of real numbers can have up to n distinct eigenvectors. Each eigenvector is associated with an eigenvalue, although the eigenvalues can be duplicated. Said another way, two eigenvectors $\mathbf{v}_1$ and $\mathbf{v}_2$ of a matrix will never be the same, but the corresponding eigenvalues λ_1 and λ_2 can be identical.

Although the number of eigenvectors may vary, all eigenvectors for a matrix are linearly independent. Thus, if an n by n matrix has n eigenvectors, these vectors can be used as a basis (called an *eigenbasis*). If an eigenbasis exists for a matrix, decomposing vectors over this basis simplifies the process of matrix

An n by n matrix with n eigenvectors and n distinct eigenvalues is called a *perfect matrix*. As the name implies, perfect matrices are great to find, but somewhat uncommon.

multiplication. To illustrate, imagine we decompose the vector $\mathbf{x}$ over a set of eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$. Decomposing $\mathbf{x}$ means we can find coefficients $a_1, \ldots, a_n$ such that

$$\mathbf{x} = a_1\mathbf{v}_1 + \cdots + a_n\mathbf{v}_n.$$

Now let's compute the product $\mathbf{Ax}$. We multiply both sides of the decomposition by $\mathbf{A}$.

$$\mathbf{Ax} = \mathbf{A}\left(a_1\mathbf{v}_1 + \cdots + a_n\mathbf{v}_n\right)$$

We distribute the matrix $\mathbf{A}$ into the sum on the right hand side and note that the constants a_i can be moved in front of the matrix multiplication.

$$\mathbf{Ax} = a_1\mathbf{Av}_1 + \cdots + a_n\mathbf{Av}_n$$

Remember that $\mathbf{v}_1, \ldots, \mathbf{v}_n$ are eigenvectors of $\mathbf{A}$, so $\mathbf{Av}_i = \lambda_i\mathbf{v}_i$. We can simplify the previous expression to

$$\mathbf{Ax} = a_1\lambda_1\mathbf{v}_1 + \cdots + a_n\lambda_n\mathbf{v}_n.$$

We don't need to perform the multiplication at all! Instead, we can scale each eigenvector by the eigenvalue. Multiplying again by the matrix $\mathbf{A}$ multiplies each eigenvector by its eigenvalue.

We use the notation $\mathbf{A}^2$ to denote $\mathbf{AA}$, $\mathbf{A}^3$ for $\mathbf{AAA}$, and $\mathbf{A}^k$ for the product of k matrices $\mathbf{A}$.

$$\mathbf{A}^2\mathbf{x} = a_1\lambda_1^2\mathbf{v}_1 + \cdots + a_n\lambda_n^2\mathbf{v}_n$$

$$\mathbf{A}^k\mathbf{x} = a_1\lambda_1^k\mathbf{v}_1 + \cdots + a_n\lambda_n^k\mathbf{v}_n$$

18.2 Computing Eigenvectors and Eigenvalues

We can use the relationship between matrix multiplication and eigenvalues to find eigenvectors for any matrix. Our computational approach is based on the following theorem.

Theorem. *Given any (random) vector* $\mathbf{b}$*, repeated multiplication by the matrix* $\mathbf{A}$ *will converge to the eigenvector of* $\mathbf{A}$ *with the largest magnitude eigenvalue – provided the largest eigenvalue is unique. Said another way,*

$$\lim_{k\to\infty} \mathbf{A}^k\mathbf{b} = \mathbf{v}_{\max}.$$

Proof. We know that the product $\mathbf{Ax}$ can be expressed as a linear combination of the eigenvectors and eigenvalues of $\mathbf{A}$, i.e. $\mathbf{Ax} = a_1\lambda_1\mathbf{v}_1 + \cdots + a_n\lambda_n\mathbf{v}_n$. Thus

$$\lim_{k\to\infty} \mathbf{Ab} = \lim_{k\to\infty}\left(a_1\lambda_1^k\mathbf{v}_1 + \cdots + a_n\lambda_n^k\mathbf{v}_n\right).$$

As k increases, the values λ_i^k grow very large. However, the λ_i to not grow at the same rate. The largest eigenvalue will grow the fastest. At very large values of k, the term associated with the largest eigenvalue will dominate the entire sum, so the result will point in only the direction of the associated eigenvector. Note that convergence to a single eigenvector requires that the largest eigenvalue be distinct. If two eigenvectors have the same (largest) eigenvalue, both terms in the above sum would "blow up" at the same rate. Repeated multiplications by $\mathbf{A}$ would then converge to the sum of the two associated eigenvectors. $\square$

The above theorem allows us to find the eigenvector paired with the largest eigenvalue. While the direction of the eigenvector doesn't change, its magnitude grows as the number of multiplication of $\mathbf{A}$ increases. If convergence is slow, we might need to work with numbers before finding the eigenvector. To avoid numerical difficulties, we renormalize the vector after every multiplication by $\mathbf{A}$. This algorithm is called the Power Iteration method, which proceeds as follows:

1. Choose a random vector $\mathbf{b}_0$. For fastest convergence, it helps to choose a vector close to $\mathbf{v}_{\max}$ if possible. Normalize this vector to product $\hat{\mathbf{b}}_0 = \mathbf{b}_0/\|\mathbf{b}_0\|$.
2. Compute vector $\mathbf{b}_1 = \mathbf{A}\hat{\mathbf{b}}_0$. Normalize this vector to give $\hat{\mathbf{b}}_1 = \mathbf{b}_1/\|\mathbf{b}_1\|$.
3. Repeat step 2 to product $\hat{\mathbf{b}}_2, \hat{\mathbf{b}}_3, \ldots, \hat{\mathbf{b}}_k$. Stop when all entries of $\hat{\mathbf{b}}_k$ do not change from the entries in $\hat{\mathbf{b}}_{k-1}$. The vector $\hat{\mathbf{b}}_k$ is an eigenvector of $\mathbf{A}$.

Now that we have the eigenvector $\mathbf{v}_{\max}$, how do we find the associated eigenvalue $\lambda_{\max}$? We know that $\mathbf{v}_{\max}$ is an eigenvector of $\mathbf{A}$, to $\mathbf{A}\mathbf{v}_{\max} = \lambda_{\max}\mathbf{v}_{\max}$. The ith element in $\mathbf{A}\mathbf{v}_{\max}$ should be equal to $\lambda_{\max}$ times the ith element in $\mathbf{v}_{\max}$. However, since we only found a numerical approximation to the $\mathbf{v}_{\max}$, the estimate for $\lambda_{\max}$ from each element in $\mathbf{v}_{\max}$ might differ slightly. To "smooth out" these variations, compute the eigenvalue using the Rayleigh quotient:

The eigenvector associated with the largest magnitude eigenvalue is called the *leading eigenvector*.

$$\lambda_{\max} = \frac{\mathbf{v}_{\max} \cdot \mathbf{A}\mathbf{v}_{\max}}{\mathbf{v}_{\max} \cdot \mathbf{v}_{\max}}.$$

To see why the Raleigh quotient works, consider an eigenvector $\mathbf{v}$ for matrix $\mathbf{A}$ with associated eigenvalue λ. Then

$$\frac{\mathbf{v} \cdot \mathbf{A}\mathbf{v}}{\mathbf{v} \cdot \mathbf{v}} = \frac{\mathbf{v} \cdot (\lambda \mathbf{v})}{\mathbf{v} \cdot \mathbf{v}} = \lambda \frac{\mathbf{v} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}} = \lambda.$$

The dot product in the Rayleigh quotient averages out all of the small discrepancies between our estimate $\mathbf{v}_{\max}$ and the true largest eigenvector. The Rayleigh quotient provides a numerically stable estimate of the largest eigenvalue.

Now that we've found the first eigenvector, how do we find the others? If we start the Power Iteration method over again using the matrix $(\mathbf{A} - \lambda_{\max}\mathbf{I})$ instead of $\mathbf{A}$, the algorithm will converge to the eigenvector associated with the second largest eigenvalue. We can subtract this eigenvalue from $\mathbf{A}$ and repeat to find the third eigenvector, and so on. Proving that Power Iteration is able to find subsequent eigenvectors is beyond the scope of this book. However, as we'll see later, finding only the first eigenvector is sufficient for addressing a number of interesting problems.

18.2.1 Eigenvalues and Eigenvectors in Matlab

The Matlab function `eig` computes eigenvalues and eigenvectors. The statement `[V,L] = eig(A)` involving an n by n matrix `A` returns two n by n matrices:

- Each column of the matrix `V` is an eigenvector `A`.
- The matrix `L` is a diagonal matrix. The ith entry on the diagonal is the eigenvalue associated with the ith column in `V`.

Remember that any vector that points in the same direction as an eigenvector of a matrix is also an eigenvector of that matrix. If the eigenvectors returned by computational systems like Matlab are not what you expect, remember that they may be normalized or scaled – but still point along the same direction.

18.3 Applications

Eigenvalue and eigenvectors can be used to solve a number of interesting engineering and data science problems.

18.3.1 Solving Systems of ODEs

Consider the linear system of ODEs

$$\begin{aligned}\frac{dx_1}{dt} &= x_1 + 2x_2\\ \frac{dx_2}{dt} &= 3x_1 + 2x_2\end{aligned}$$

with initial conditions $x_1(0) = 0$ and $x_2(0) = -4$. We can write this system using vectors and matrices as

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x}, \quad \mathbf{x}(0) = \mathbf{x}_0$$

where for the example above

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad \mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix}, \quad \mathbf{x}_0 = \begin{pmatrix} 0 \\ -4 \end{pmatrix}.$$

If we know the eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ and eigenvalues $\lambda_1, \ldots, \lambda_n$ for the matrix $\mathbf{A}$, we can compute the solution as

$$\mathbf{x}(t) = c_1 \mathbf{v}_1 e^{\lambda_1 t} + c_2 \mathbf{v}_2 e^{\lambda_2 t} + \cdots + c_n \mathbf{v}_n e^{\lambda_n t}.$$

This solution requires the matrix $\mathbf{A}$ be perfect and therefore have a complete set of eigenvectors.

The scalars $c_1, \ldots, c_n$ are the constants of integration. To find these values, notice what happens to our solution at time $t = 0$:

$$\mathbf{x}(0) = \mathbf{x}_0 = c_1 \mathbf{v}_1 + c_2 \mathbf{v}_2 + \cdots + c_n \mathbf{v}_n.$$

The function $f(t) = e^{\lambda t}$ is an *eigenfunction* of the derivative operator, i.e.

$$\frac{d}{dt} f(t) = \lambda e^{\lambda t} = \lambda f(t)$$

. The solution of a system of linear ODEs is the product of the eigenvectors of $\mathbf{A}$ and the eigenfunctions of $\frac{d\mathbf{x}}{dt}$.

At $t = 0$, the right hand side is a decomposition of the initial conditions $\mathbf{x}_0$. If we collect the eigenvectors as columns of a matrix $\mathbf{V} = (\mathbf{v}_1 \mathbf{v}_2 \ldots \mathbf{v}_n)$, we can find the constants $c_1, \ldots, c_n$ by solving the linear system

$$\mathbf{V} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} = \mathbf{x}_0.$$

Returning to our original example, the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix}$$

has eigenvalue/eigenvector pairs

$$\lambda_1 = -1, \quad \mathbf{v}_1 = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \qquad \text{and} \qquad \lambda_2 = 4, \quad \mathbf{v}_2 = \begin{pmatrix} 2 \\ 3 \end{pmatrix}.$$

The integration constants c_1 and c_2 are defined by the system $\mathbf{Vc} = \mathbf{x}_0$, which for this example is

$$\begin{pmatrix} -1 & 2 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 0 \\ -4 \end{pmatrix}.$$

Solving the above equations reveals $c_1 = -8/5$ and $c_2 = -4/5$. The final solution to this systems of ODEs is

$$\mathbf{x}(t) = -\frac{8}{5} \begin{pmatrix} -1 \\ 1 \end{pmatrix} e^{-t} - \frac{4}{5} \begin{pmatrix} 2 \\ 3 \end{pmatrix} e^{4t}.$$

18.3.2 Stability of Linear ODEs

The eigenvalues of **A** are sufficient to tell if the system $\frac{d\mathbf{x}}{dt} = \mathbf{Ax}$ is stable. For a system of linear ODEs to be stable, all eigenvalues of **A** must be nonpositive. If the eigenvalues are all negative, each term $e^{\lambda_i t}$ goes to zero at long times, so all variables in the system to go zero. If any of the eigenvalues are zero, the system is still stable (provided all other eigenvalues are negative), but the system will go to a constant value $c_i\mathbf{v}_i$, where $\mathbf{v}_i$ is the eigenvector associated with the zero eigenvalue.

18.3.3 Positive Definite Matrices

A symmetric matrix **A** is *positive definite* (p.d.) if $\mathbf{x}^\mathsf{T}\mathbf{Ax} > 0$ for all nonzero vectors **x**. If a matrix **A** satisfies the slightly relaxed requirement that $\mathbf{x}^\mathsf{T}\mathbf{Ax} \geq 0$ for all nonzero **x**, we say that **A** is *positive semi-definite* (p.s.d.).

Remember that a matrix **A** is symmetric if $\mathbf{A} = \mathbf{A}^\mathsf{T}$.

Knowing that a matrix is positive (semi-)definite is useful for quadratic programming problems like the Support Vector Machine classifier. The quadratic function $f(\mathbf{x}) = \mathbf{x}^\mathsf{T}\mathbf{Qx}$ is convex if and only if the matrix **Q** is positive semi-definite. For optimization problems like quadratic programs, the convexity of the objective function has enormous implications. Convex quadratic programs must only have global optima, making them easy to solve using numerical algorithms.

If **Q** is positive definite (rather than just positive semi-definite) then $\mathbf{x}^\mathsf{T}\mathbf{Qx}$ is strictly convex.

Determining if a matrix is positive (semi-)definite can be difficult unless we use eigenvalues. Any matrix with only positive eigenvalues is positive definite, and any matrix with only nonnegative eigenvalues is positive semi-definite. For example, consider the 2×2 identity matrix

$$\mathbf{I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

The product $\mathbf{x}^\mathsf{T}\mathbf{Ix}$ is

$$\begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = x_1^2 + x_2^2.$$

Since $x_1^2 + x_2^2$ is greater than zero for all nonzero inputs x_1 and x_2, the matrix **I** is positive definite and all its eigenvalues should be positive. Indeed, the eigenvalues for the identity matrix are $\lambda_1 = \lambda_2 = 1$.

As another example, consider the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & -2 \\ -2 & 1 \end{pmatrix}.$$

The product $\mathbf{x}^\mathsf{T}\mathbf{Ax} = x_1^2 - 4x_1x_2 + x_2^2$, which is not always positive. When $x_1 = x_2 = 1$, we see that $x_1^2 - 4x_1x_2 + x_2^2 = -2$. We know that **A** is not positive definite (or even positive semi-definite), so **A** should have at least one negative eigenvalue. As expected, the eigenvalues for **A** are $\lambda_1 = 3$ and $\lambda_2 = -1$.

18.3.4 Network Centrality

Networks are represented by collections of *nodes* connected by *edges*. When analyzing a network, it is common to ask which node occupies the most important position in the network. For example, which airport would cause the most problems if closed due to weather? In biological networks, the importance or *centrality* of an enzyme is used to prioritize drug targets.

We can quantify the centrality of each node in a network using random walks. We start by choosing a random node in the network. Then we randomly choose one of the edges connected to the node and travel to a new node. This process repeats again and again as we randomly traverse nodes and edges. The centrality of each node is proportional to the number of times we visit the node during the random walk. In the airport analogy, randomly traveling to cities across the country means you will frequently visit major hub airports.

Measuring centrality by random walks is easy to understand but impractical for large networks. It could take millions of steps to repeatedly reach all the nodes in networks with only a few thousand of nodes. In practice, we use eigenvectors to calculate centrality for networks. We begin by constructing an *adjacency matrix* for the network. The adjacency matrix encodes the connections (edges) between the nodes. The adjacency matrix is square with rows and columns corresponding to nodes in the network. The (i,j) entry in the network is set to 1 if there is an edge connecting node i with node j. Otherwise, the entries are zero. Note that we only consider direct connections. If node 1 is connected to node 2 and node 2 is connected to node 3, we do not connect nodes 1 and 3 unless there is a direct edge between them. Also, no node is connected to itself, so the diagonal elements are always zero.

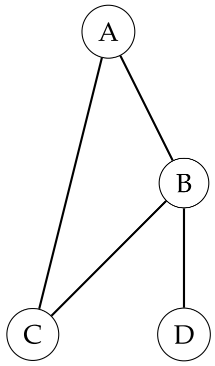


Figure 18.1: Sample network with four nodes and four edges.

Consider the four node network shown in Figure 18.1. The four node network has the following adjacency matrix.

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

To identify the most central node in the network, we find the eigenvector that corresponds to the largest eigenvalue ($\lambda_{\max}$). For the above adjacency matrix,

$\lambda_{\max} \approx 2.2$, and the associated eigenvector is

$$\mathbf{v}_{\max} = \begin{pmatrix} 0.52 \\ 0.61 \\ 0.52 \\ 0.28 \end{pmatrix}.$$

The entries in the eigenvector $\mathbf{v}_{\max}$ are called the *eigencentrality* scores. The largest entry corresponds to the most central node, and the smallest entry is associated with the least central node. We see that node B (entry 2) is most central in Figure 18.1 and node D (entry 4) is least central.

Centrality requires only the leading eigenvector of a network's adjacency matrix. Power Iteration (section 18.2) can find the leading eigenvector efficiently for large networks.

In simple networks like Figure 18.1, the most central node also has the most direct connections. This is not always the case. Eigencentrality considers not only the number of connections but also their importance. Each edge is weighted by the centrality of the nodes it connect. Connections from more central nodes are more important, just as flights between major hub cities usually have the highest passenger volumes. Eigencentrality has numerous applications including web searching. Google uses a modified version of centrality (called PageRank) to determine which results should be displayed first to users.

18.4 Geometric Interpretation of Eigenvalues

Consider a matrix $\mathbf{A} \in \mathbb{R}^2$ with eigenvalues λ_1 and λ_2 and corresponding eigenvectors $\mathbf{v}_1$ and $\mathbf{v}_2$. Let's take a vector $\mathbf{x}$ and decompose it over the eigenvectors.

$$\mathbf{x} = a_1\mathbf{v}_1 + a_2\mathbf{v}_2$$

We can represent the vector $\mathbf{x}$ by plotting it; however, instead of using the normal Cartesian unit vectors as axes, we will use the eigenvectors. The values of $\mathbf{x}$ along the "eigenaxes" are a_1 and a_2. What happens when we multiply $\mathbf{x}$ and $\mathbf{A}$?

$$\mathbf{A}\mathbf{x} = \mathbf{A}(a_1\mathbf{v}_1 + a_2\mathbf{v}_2) = \lambda_1 a_1\mathbf{v}_1 + \lambda_2 a_2\mathbf{v}_2$$

Visually, multiplying by $\mathbf{A}$ scales the values along the eigenvector axes. The scaling factor along each axis is the corresponding eigenvalue. To quantify the overall effect of multiplying by the matrix $\mathbf{A}$, we can compare the areas swept out by the vector $\mathbf{x}$ before and after multiplication. The area is simply the product of the values along each axis, so the ratio becomes

$$\frac{\text{area}(\mathbf{A}\mathbf{x})}{\text{area}(\mathbf{x})} = \frac{\lambda_1 a_1 \lambda_2 a_2}{a_1 a_2} = \lambda_1\lambda_2.$$

We can repeat the same calculation in three dimensions by looking at the ratio of the volume before and after multiplying by the matrix **A**.

$$\frac{\text{volume}(\mathbf{Ax})}{\text{volume}(\mathbf{x})} = \frac{\lambda_1 a_1 \lambda_2 a_2 \lambda_3 a_3}{a_1 a_2 a_3} = \lambda_1\lambda_2\lambda_3$$

In general, the product of the eigenvalues of a matrix describe to overall effect of multiplying a vector by the matrix. The product of the eigenvalues of a matrix **A** is called the *determinant* of **A**, or det(**A**).

$$\det(\mathbf{A}) = \lambda_1\lambda_2\cdots\lambda_n$$

If the determinant of a matrix is large, multiplying a vector by the matrix enlarges the volume swept out by the vector. If the determinant is small, the volume contracts.

Remember that the volume we're discussing here is the volume when a vector is plotted using the matrix's eigenvectors as axes.

18.5 Properties of the Determinant

The determinant is a powerful property of a matrix. Determinants can be easily calculated for a matrix and contain useful information about the matrix.

In MATLAB, the function `det` calculates the determinant of a matrix.

Let's say a vector $\mathbf{y} = \mathbf{Ax}$. We know the determinant of the matrix **A** is the ratio of the volumes of **x** and **y**.

$$\det(\mathbf{A}) = \frac{\text{volume}(\mathbf{Ax})}{\text{volume}(\mathbf{x})} = \frac{\text{volume}(\mathbf{y})}{\text{volume}(\mathbf{x})}$$

If the inverse of **A** exists, we know that $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$. Thus, the determinant of the inverse matrix $\mathbf{A}^{-1}$ is

$$\det(\mathbf{A}^{-1}) = \frac{\text{volume}(\mathbf{A}^{-1}\mathbf{y})}{\text{volume}(\mathbf{y})} = \frac{\text{volume}(\mathbf{x})}{\text{volume}(\mathbf{y})} = \frac{1}{\det(\mathbf{A})}.$$

The determinant of $\mathbf{A}^{-1}$ is the inverse of the determinant of **A**. If the determinant of a matrix is zero, this property indicates there is a problem with the inverse of the matrix.

$$\det(\mathbf{A}) = 0 \Rightarrow \det(\mathbf{A}^{-1}) = \frac{1}{\det(\mathbf{A})} = \frac{1}{0} \rightarrow \text{undefined}$$

Although we won't prove it in this course, **a matrix has an inverse if and only if the determinant of the matrix is nonzero**. The following statements are, in fact, equivalent for a square matrix **A**:

- $\mathbf{A}$ can be transformed into the identity matrix by elementary row operations.
- The system $\mathbf{Ax} = \mathbf{y}$ is solvable and has a unique solution.
- $\mathbf{A}$ is full rank.
- $\mathbf{A}^{-1}$ exists.
- $\det(\mathbf{A}) \neq 0$.

A matrix with a determinant equal to zero has a geometric interpretation. Remember that the determinant is the product of the eigenvalues. It is the product of the scaling factors of the matrix along each eigenvector. If one of the eigenvalues is zero, we are missing information about how the matrix scales along at least one eigenvector. Our knowledge of the transformation is incomplete, which is why we cannot find a unique solution for the corresponding linear system.

Using the determinant we can concisely state our last field axiom. Recall that for scalars we required a multiplicative inverse exist for any nonzero member of the field, i.e.

$$\text{For all scalars } a \neq 0 \text{ there exists } a^{-1} \text{ such that } aa^{-1} = 1.$$

For vector spaces, we have the following:

$$\begin{gathered}\text{For all square matrices } \mathbf{A} \text{ where } \det(\mathbf{A}) \neq 0, \\ \text{there exists } \mathbf{A}^{-1} \text{ such that } \mathbf{AA}^{-1} = \mathbf{I} = \mathbf{A}^{-1}\mathbf{A}.\end{gathered}$$

Chapter 19

Matrix Decompositions

19.1 Eigendecomposition

Let's discuss a square, $n \times n$ matrix $\mathbf{A}$. Provided $\mathbf{A}$ is not defective, it has n linearly independent eigenvectors which we will call $\mathbf{v}_1, \ldots, \mathbf{v}_n$. The eigenvectors are linearly independent and therefore form a basis for $\mathbb{R}^n$ (an *eigenbasis*). We said in the last chapter that any vector $\mathbf{x}$ can be decomposed onto the eigenbasis by finding coefficients $a_1, \ldots, a_n$ such that

$$\mathbf{x} = a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n.$$

Multiplying the vector $\mathbf{x}$ by the matrix $\mathbf{A}$ is equivalent to scaling each term in the decomposition by the corresponding eigenvalue (λ_i).

$$\mathbf{A}\mathbf{x} = a_1\lambda_1\mathbf{v}_1 + a_2\lambda_2\mathbf{v}_2 + \cdots + a_n\lambda_n\mathbf{v}_n$$

We can think of matrix multiplication as a transformation with three steps.

1. Decompose the input vector onto the eigenbasis of the matrix.
2. Scale each term in the decomposition by the appropriate eigenvalue.
3. Reassemble, or "un-decompose" the output vector.

Each of these steps can be represented by a matrix operation. First, we collect the n eigenvalue into a matrix $\mathbf{V}$.

$$\mathbf{V} = (\mathbf{v}_1\, \mathbf{v}_2\, \cdots\, \mathbf{v}_n)$$

Each column in the matrix $\mathbf{V}$ is an eigenvector of the matrix $\mathbf{A}$. To decompose the vector $\mathbf{x}$ onto the columns of $\mathbf{V}$ we find the coefficients $a_1, \dots, a_n$ by solving the linear system

$$\mathbf{Va} = \mathbf{x}$$

where $\mathbf{a}$ is a vector holding the coefficients $a_1, \dots, a_n$. The matrix $\mathbf{V}$ is square and has linearly independent columns (the eigenvectors of $\mathbf{A}$), so its inverse exists. The coefficients for decomposing the vector $\mathbf{x}$ onto the eigenbasis of the matrix $\mathbf{A}$ are

$$\mathbf{a} = \mathbf{V}^{-1}\mathbf{x}.$$

If the inverse matrix $\mathbf{V}^{-1}$ decomposes a vector into a set of coefficients $\mathbf{a}$, then multiplying the coefficients vector $\mathbf{a}$ by the original matrix must reassemble the vector $\mathbf{x}$. Looking back at the three steps we defined above, we can use multiplication by $\mathbf{V}^{-1}$ to complete step 1 and multiply by $\mathbf{V}$ to perform step 3. For step 2, we need to scale the individual coefficients by the appropriate eigenvalues. We define a scaling matrix $\mathbf{\Lambda}$ as a diagonal matrix of the eigenvalues:

In other words, if $\mathbf{V}^{-1}$ decomposes a vector, $(\mathbf{V}^{-1})^{-1} = \mathbf{V}$ must undo the decomposition.

We use the uppercase Greek lambda ($\mathbf{\Lambda}$) to denote the matrix of eigenvalues λ_i (lowercase lambda).

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}.$$

Notice that

$$\mathbf{\Lambda a} = \begin{pmatrix} \lambda_1 a_1 \\ \lambda_2 a_2 \\ \vdots \\ \lambda_n a_n \end{pmatrix}$$

so the matrix $\mathbf{\Lambda}$ scales the ith entry of the input vector by the ith eigenvalue.

We now have matrix operations for decomposing onto an eigenbasis ($\mathbf{V}^{-1}$), scaling by eigenvalues ($\mathbf{\Lambda}$), and reassembling the output vector ($\mathbf{V}$). Putting everything together, we see that matrix multiplication can be expressed as an *eigendecomposition* by

$$\mathbf{Ax} = \mathbf{V\Lambda V}^{-1}\mathbf{x}.$$

Eigendecomposition is the last time we will use the prefix "eigen-". Feel free to use it on other everyday words to appear smarter.

Equivalently, we can say that the matrix $\mathbf{A}$ itself can be as the product of three matrices ($\mathbf{A} = \mathbf{V\Lambda V}^{-1}$) if $\mathbf{A}$ has a complete set of eigenvectors. There are two ways to interpret the dependence on a complete set of eigenvectors. Viewed technically, the matrix $\mathbf{V}$ can only be inverted if it is full rank, so $\mathbf{V}^{-1}$ does not exist if one or more eigenvectors is missing. More intuitively, the eigendecomposition defines a unique mapping between the input and output vectors. Uniqueness requires a basis, since a vector decomposition is only unique if the set of vectors form a basis. If the matrix **A** is defective, its eigenvectors do not form an eigenbasis and there cannot be a unique mapping between inputs and outputs.

19.2 Singular Value Decomposition

The eigendecomposition is limited to square matrices with a complete set of eigenvectors. However, the idea that matrices can be factored into three operations (decomposition, scaling, and reassembly) generalizes to all matrices, even non-square matrices. The generalized equivalent of the eigendecomposition is called the *Singular Value Decomposition*, or (SVD).

Singular Value Decomposition. *Any $m \times n$ matrix* **A** *can be factored into the product of three matrices*

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathsf{T}}$$

where

- **U** *is an orthogonal $m \times m$ matrix.*
- $\boldsymbol{\Sigma}$ *is a diagonal $m \times n$ matrix with nonzero entries.*
- **V** *is an orthogonal $n \times n$ matrix.*

The square matrices **U** and **V** are *orthogonal*, i.e. their columns form an orthonormal set of basis vectors. As we discussed previously, the inverse of an orthogonal matrix is simply its transpose, so $\mathbf{U}^{-1} = \mathbf{U}^{\mathsf{T}}$ and $\mathbf{V}^{-1} = \mathbf{V}^{\mathsf{T}}$. The $\mathbf{V}^{\mathsf{T}}$ term in the decomposition has the same role as the $\mathbf{V}^{-1}$ matrix in an eigendecomposition — projection of the input vector onto a new basis. The matrix **U** in SVD reassembles the output vector analogous to the vector **V** in an eigendecomposition.

If the entries in **A** were complex numbers, the matrices **U** and **V** would be *unitary*. The inverse of a unitary matrix is the complex conjugate of the matrix transpose.

The matrix $\boldsymbol{\Sigma}$ is diagonal but not necessarily square. It has the same dimensions as the original matrix **A**. For a 3×5 matrix, the $\boldsymbol{\Sigma}$ has the form

$$\boldsymbol{\Sigma} = \begin{pmatrix} \sigma_1 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 & 0 \end{pmatrix}.$$

If the matrix **A** was 5×3, we would have

$$\boldsymbol{\Sigma} = \begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The elements along the diagonal of $\mathbf{\Sigma}$ are called *singular values*. If $\mathbf{A}$ is an $m \times n$ matrix, the maximum number of nonzero singular values is $\min\{m, n\}$. The are the analogues of eigenvalues for non-square matrices. However, the singular values for a square matrix are not equal to the eigenvalues of the same matrix. Singular values are always nonnegative. If we arrange $\mathbf{\Sigma}$ such that the singular values are in descending order, the SVD of a matrix is unique.

The columns in $\mathbf{U}$ and $\mathbf{V}$ are called the left and right *singular vectors*, respectively. Just as there is a relationship between eigenvalues and eigenvectors, the columns in $\mathbf{U}$ and $\mathbf{V}$ are connected by the singular values in $\mathbf{\Sigma}$. If $\mathbf{A} = \mathbf{U\Sigma V}^\mathsf{T}$, then

$$\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i$$

and

$$\mathbf{A}^\mathsf{T}\mathbf{u}_i = \sigma_i \mathbf{v}_i$$

where $\mathbf{v}_i$ is ith right singular vector (the ith column in $\mathbf{V}$); $\mathbf{u}_i$ is the ith left singular vector (the ith column in $\mathbf{U}$); and σ_i is the ith singular value (the ith nonzero on the diagonal of $\mathbf{\Sigma}$).

19.3 Applications of the SVD

19.3.1 Rank of a matrix

The rank of a matrix $\mathbf{A}$ is equal to the number of nonzero singular values (the number of nonzero values along the diagonal of $\mathbf{\Sigma}$). This is true for both square and nonsquare matrices. Notice that the way we defined the diagonal matrix $\mathbf{\Sigma}$ implies that the number of singular values must be at most $\min\{m, n\}$ for an $m \times n$ matrix. This requirement agrees with our knowledge that $\text{rank}(\mathbf{A}) \le \min\{m, n\}$.

19.3.2 The matrix pseudoinverse

Our definition of a matrix inverse applies only to square matrices. For nonsquare matrices we can use the SVD to construct a pseudoinverse. We represent the pseudoinverse of a matrix $\mathbf{A}$ as $\mathbf{A}^+$. We simply reverse and invert the factorization of $\mathbf{A}$, i.e.

$$\mathbf{A}^+ = (\mathbf{V}^\mathsf{T})^{-1}\mathbf{\Sigma}^+\mathbf{U}^{-1}$$

We can simplify this expression with knowledge that $\mathbf{V}$ and $\mathbf{U}$ are orthogonal, so $(\mathbf{V}^\mathsf{T})^{-1} = (\mathbf{V}^\mathsf{T})^\mathsf{T} = \mathbf{V}$ and $\mathbf{U}^{-1} = \mathbf{U}^\mathsf{T}$. Thus

$$\mathbf{A}^+ = \mathbf{V\Sigma}^+\mathbf{U}^\mathsf{T}.$$

The matrix $\mathbf{\Sigma}^+$ is the pseudoinverse of the diagonal matrix $\mathbf{\Sigma}$. This is simply the transpose of $\mathbf{\Sigma}$ where each entry on the diagonal is replaced by its multiplicative inverse. For example, a 3×5 matrix $\mathbf{\Sigma}$:

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma_1 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 & 0 \end{pmatrix}$$

the pseudoinverse $\mathbf{\Sigma}^+$ is

$$\mathbf{\Sigma}^+ = \begin{pmatrix} 1/\sigma_1 & 0 & 0 \\ 0 & 1/\sigma_2 & 0 \\ 0 & 0 & 1/\sigma_3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Chapter 20

Low Rank Approximations

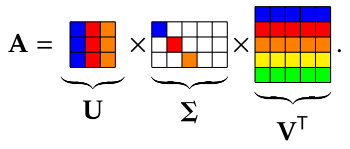
The previous chapter introduced the Singular Value Decomposition (SVD) and showed how every matrix can be decomposed into the product of three matrices. Any $m \times n$ matrix $\mathbf{A}$ is equal to $\mathbf{U\Sigma V}^\mathsf{T}$ where

1. An $m \times m$ orthogonal matrix $\mathbf{U}$ that forms a basis for the output dimension.
2. An $m \times n$ diagonal matrix $\mathbf{\Sigma}$ that holds the singular values.
3. An $n \times n$ orthogonal matrix $\mathbf{V}$ that forms a basis for the input dimension.

We can use the SVD to help us understand how multiplication transforms vectors between dimensions. Assume that a matrix $\mathbf{A}$ has dimensions 3×5. If $\mathbf{y} = \mathbf{Ax}$, the input vector $\mathbf{x}$ is 5-dimensional and the output vector $\mathbf{y}$ is 3-dimensional. The input vector $\mathbf{x}$ lost two dimensions somewhere in the matrix $\mathbf{A}$. Where did they go?

Let's look at the SVD of the matrix $\mathbf{A}$. We're not interested in the particular numbers in the matrices—just the overall structure. Visually,

We're showing $\mathbf{V}^\mathsf{T}$ in this diagram, so the column vectors in $\mathbf{V}$ appear as row vectors in $\mathbf{V}^\mathsf{T}$.



There's lots to analyze here. First, only the colored entries contain nonzero numbers. The white squares in $\mathbf{\Sigma}$ and zero since the singular values only appear along the diagonal. Also note that the number of singular values never exceeds the *smaller* dimension of the original matrix. Since $\mathbf{A}$ is a 3×5 matrix, there can be no more than three nonzero singular values.

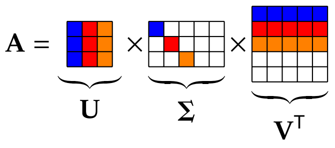
The columns of zeros in the matrix $\mathbf{\Sigma}$ will "zero-out" the last two rows (green and yellow) of the matrix $\mathbf{V}^\mathsf{T}$ during multiplication. Thus the green and yellow rows do not contribute at all to the output vector. This is how the matrix $\mathbf{A}$ moves from five to three dimensions:

1. The input vector is decomposed onto the 5-dimensional basis formed by the columns in $\mathbf{V}$. Normally this decomposition requires multiplying the input vector by the inverse $\mathbf{V}^{-1}$; however, since $\mathbf{V}$ is an orthogonal matrix, its inverse equals its transpose.

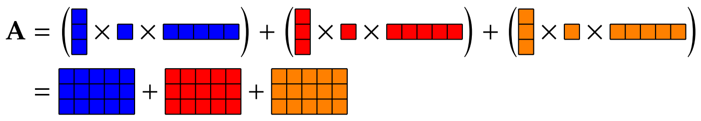
2. Multiplying by $\mathbf{\Sigma}$ scales the first three dimensions by the corresponding singular values. The last two dimensions are dropped.

3. The surviving three dimensions are projected into the output space by the basis vectors in the matrix $\mathbf{U}$.

Since the green and yellow rows in $\mathbf{V}^\mathsf{T}$ are going to be zeroed-out, let's change their color to white to match the zeros in the matrix $\mathbf{\Sigma}$. Then they'll be easier to ignore.



The second thing to note is that the blue entries in all three matrices are only multiplied by other blue entries. The same is true for the red and orange entries. You can follow through the multiplication and separate it into three parts, one blue, one red, and one orange.



So the matrix $\mathbf{A}$ is actually the sum of three separate matrices—one blue, one red, and one orange. The blue column in the matrix $\mathbf{U}$ is the first left singular vector ($\mathbf{u}_1$). The blue square in $\mathbf{\Sigma}$ is the first singular value (σ_1). The blue row in the matrix $\mathbf{V}^\mathsf{T}$ is actually a column vector in the untransposed matrix $\mathbf{V}$ and is the first right singular vector ($\mathbf{v}_1$).

Let's write this all out again using mathematical symbols instead of boxes for the matrices. We'll retain the same color scheme as a guide and drop rows 4–5 in

$\mathbf{V}^\mathsf{T}$ since these rows are zeroed-out in the final product.

$$\mathbf{A} = \underbrace{\begin{pmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \mathbf{u}_3 \end{pmatrix}}_{\mathbf{U}} \underbrace{\begin{pmatrix} \sigma_1 & & \\ & \sigma_2 & \\ & & \sigma_3 \end{pmatrix}}_{\boldsymbol{\Sigma}} \underbrace{\begin{pmatrix} \mathbf{v}_1^\mathsf{T} \\ \mathbf{v}_2^\mathsf{T} \\ \mathbf{v}_3^\mathsf{T} \end{pmatrix}}_{\mathbf{V}^\mathsf{T}}$$

$$= \sigma_1 \mathbf{u}_1 \mathbf{v}_1^\mathsf{T} + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^\mathsf{T} + \sigma_3 \mathbf{u}_3 \mathbf{v}_3^\mathsf{T}$$

The SVD decomposes any matrix into a weighted sum of matrices created by the pairs of singular vectors $\mathbf{u}_i \mathbf{v}_i^\mathsf{T}$. The weights in this sum are the singular values σ_i. These scalars define how much each pair of singular vectors ($\mathbf{u}_i$ and $\mathbf{v}_i$) contribute to the matrix. Some pairs have large singular values and therefore contribute a lot. Pairs with small singular values contribute very little. As an extreme, any pair of singular vectors associated with a zero singular value *does not contribute at all*.

The singular values in the matrix $\boldsymbol{\Sigma}$ are by convention ordered by decreasing magnitude ($\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k$). Since the singular vectors are the weights for the singular vectors, the singular vectors are similarly ordered by decreasing importance. The first pair of singular vectors $\mathbf{u}_1$, $\mathbf{v}_1$ has the largest influence on the overall matrix, followed by the second pair $\mathbf{u}_2$, $\mathbf{v}_2$, and so on. Some pairs of singular vectors are associated with such a small singular value that they can be removed with little effect, as demonstrated by the following numerical example.

Remember that singular values are always nonnegative.

In this example, we used the SVD to decompose a 4×3 matrix. Since the output dimension (4) is larger than the input dimension (3), the matrix $\mathbf{U}$ contains "extra" columns that are zeroed-out by the matrix $\boldsymbol{\Sigma}$.

$$\mathbf{A} = \begin{pmatrix} 0.67 & 0.99 & 0.61 \\ 0.70 & 0.13 & 0.81 \\ 0.12 & 0.18 & 0.11 \\ 0.24 & 0.77 & 0.12 \end{pmatrix}$$

$$= \underbrace{\begin{pmatrix} -0.75 & -0.23 & 0.62 & 0.03 \\ -0.51 & 0.79 & -0.33 & 0.09 \\ -0.14 & -0.04 & -0.15 & -0.98 \\ -0.39 & -0.57 & -0.70 & 0.18 \end{pmatrix}}_{\mathbf{U}} \underbrace{\begin{pmatrix} 1.76 & 0 & 0 \\ 0 & 0.75 & 0 \\ 0 & 0 & 0.01 \\ 0 & 0 & 0 \end{pmatrix}}_{\boldsymbol{\Sigma}} \underbrace{\begin{pmatrix} -0.55 & 0.33 & -0.76 \\ -0.64 & -0.75 & 0.13 \\ -0.53 & 0.57 & 0.63 \end{pmatrix}}_{\mathbf{V}^\mathsf{T}}$$

Notice how small the third singular value is ($\sigma_3 = 0.01$) compared to the other two singular values ($\sigma_1 = 1.76$ and $\sigma_2 = 0.75$). The singular vectors associated with σ_3 (in orange) contribute little to the overall matrix $\mathbf{A}$ because they are multiplied by such a small singular value. We could probably change the singular value to zero and not change the matrix much. Let's delete σ_3 and call the resulting matrix $\mathbf{A}_2$ since it contains only two of the three original singular values.

Notice that we call the inner matrix $\boldsymbol{\Sigma}_2$ since we've modified it to only have two singular values.

$$\mathbf{A}_2 = \underbrace{\begin{pmatrix} -0.75 & -0.23 & 0.62 & 0.03 \\ -0.51 & 0.79 & -0.33 & 0.09 \\ -0.14 & -0.04 & -0.15 & -0.98 \\ -0.39 & -0.57 & -0.70 & 0.18 \end{pmatrix}}_{\mathbf{U}} \underbrace{\begin{pmatrix} 1.76 & 0 & 0 \\ 0 & 0.75 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}}_{\boldsymbol{\Sigma}_2} \underbrace{\begin{pmatrix} -0.55 & 0.33 & -0.76 \\ -0.64 & -0.75 & 0.13 \\ -0.53 & 0.57 & 0.63 \end{pmatrix}}_{\mathbf{V}^\mathsf{T}}$$

$$= \begin{pmatrix} 0.67 & 0.99 & 0.61 \\ 0.70 & 0.13 & 0.81 \\ 0.12 & 0.18 & 0.11 \\ 0.24 & 0.77 & 0.12 \end{pmatrix}$$

Deleting the small singular value σ_3 had almost no effect on the matrix. The matrices $\mathbf{A}$ and $\mathbf{A}_2$ appear identical due to rounding, but there are some small differences. To emphasize the connection between the singular values and the corresponding singular vectors, we can also zero-out the orange values in $\mathbf{U}$ and $\mathbf{V}$ since they are only multiplied by the singular value σ_3 that we've set to zero. While we're at it, we can zero-out the fourth column in $\mathbf{U}$ which is never used because there are only three singular values for the 4×3 matrix.

$$\mathbf{A}_2 = \underbrace{\begin{pmatrix} -0.75 & -0.23 & 0 & 0 \\ -0.51 & 0.79 & 0 & 0 \\ -0.14 & -0.04 & 0 & 0 \\ -0.39 & -0.57 & 0 & 0 \end{pmatrix}}_{\mathbf{U}_2} \underbrace{\begin{pmatrix} 1.76 & 0 & 0 \\ 0 & 0.75 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}}_{\boldsymbol{\Sigma}_2} \underbrace{\begin{pmatrix} -0.55 & 0.33 & -0.76 \\ -0.64 & -0.75 & 0.13 \\ 0 & 0 & 0 \end{pmatrix}}_{\mathbf{V}_2^\mathsf{T}}$$

$$= \begin{pmatrix} 0.67 & 0.99 & 0.61 \\ 0.70 & 0.13 & 0.81 \\ 0.12 & 0.18 & 0.11 \\ 0.24 & 0.77 & 0.12 \end{pmatrix}$$

Matrices like $\mathbf{A}_2$ are called *low rank approximations* to the original matrix $\mathbf{A}$. In section 19.3 we said that the rank of a matrix is equal to the number of nonzero singular values in its SVD. Zeroing-out a singular value creates a new matrix with a lower rank. In the example above, the matrix $\mathbf{A}_2$ is a "rank 2" approximation to the matrix $\mathbf{A}$.

We always delete the smallest singular values when creating a low rank approximation. This ensures the smallest loss of information when reconstructing the original matrix. In the matrix $\mathbf{A}$ from above, the third singular value was so small that its deletion was almost unnoticeable.

How small is small for a singular value? It all depends on the size of the other singular values. We judge the size of a singular value relative to the others. For the matrix $\mathbf{A}$, the sum of the singular values was $1.76 + 0.75 + 0.01 = 2.52$, so the third singular value represented only $0.01/2.52 = 0.4\%$ of the information in the matrix. We can use singular values to assess the information distribution of a matrix since the singular vectors have been normalized. Only the singular values determine the relative contribution of the products $\sigma_i \mathbf{u}_i \mathbf{v}_i^\mathsf{T}$ that make up the original matrix.

20.1 Image Compression

A low rank approximation for a matrix contains several rows and columns of zeros. If a singular value in the matrix $\boldsymbol{\Sigma}$ is zero, the corresponding column in $\mathbf{U}$ and the corresponding row in $\mathbf{V}^\mathsf{T}$ can also be set to zero as shown in the previous section. We can go a step further and remove the columns of zeros entirely. The number of rows in $\mathbf{U}$ and the number of columns in $\mathbf{V}^\mathsf{T}$ determine the dimensions of the product $\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\mathsf{T}$, so deleting a column from $\mathbf{U}$ or a row from $\mathbf{V}^\mathsf{T}$ will not change the dimensions of the product. A low rank approximation requires less memory to store on a computer and can be used as a form of image compression.

Before we see an example of image compression, we should mention that there are two types of compression. Low rank approximations are *lossy* compression schemes since they destroy information that can't be recovered. However, removing small singular values deletes only a small amount information, so the compressed image retains most of its features. *Lossless* compression retains all the information in an image by finding ways to store the image more efficiently. For example, a lossless compressor might replace frequent sequences of bits with an abbreviation to reduce the image size. We don't talk about lossless compression in this book, but many of these algorithms also use linear algebra.

A digital image can be represented as a matrix of pixels. For simplicity we'll discuss greyscale (black and white) images where each pixel is a number between zero (black) and one (white). Color images have either three (RGB) or four (CMYK) entries for each pixel to describe the color intensities.

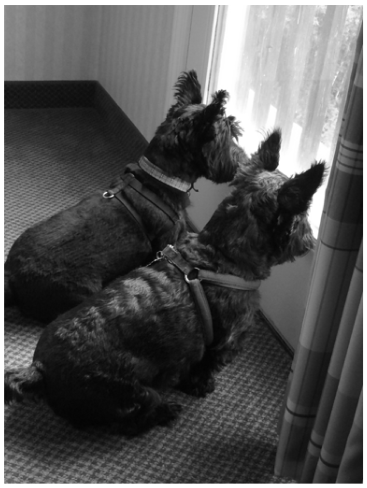


Figure 20.1: A 512×384 pixel, grayscale image of two adorable Scottie dogs. The upper dog is Duncan. The lower dog is Rocky. They are best friends.

Figure 20.1 is an image of two Scottish Terriers looking out a window. The image is 512 pixels high and 384 pixels wide, creating a 512×384 matrix. The rank of this matrix can be no larger than $\min\{512, 384\} = 384$, so the SVD of the image matrix will have at most 384 nonzero singular values.

Let's compress this image using a low rank approximation with the following steps on the image matrix $\mathbf{A}$.

1. We decompose the matrix $\mathbf{A}$ using the SVD: $\mathbf{A} = \mathbf{U\Sigma V}^\mathsf{T}$.
2. We keep only the k largest singular values remove the zeroed-out columns in $\mathbf{U}$ and $\mathbf{V}$ to create smaller matrices $\mathbf{U}_k$, $\mathbf{\Sigma}_k$, and $\mathbf{V}_k$.
3. We multiply these matrices together to form a low rank approximation for the original image $\mathbf{A}_k = \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^\mathsf{T}$.

Figure 20.2 shows six low rank approximations for the image in Figure 20.1. The first image is actually a full rank reconstruction since it uses all 384 singular values. The second image uses the largest 96 singular values, so it contains only a fraction of the information in the first image; however, the low rank approximation is almost indistinguishable from the full image. It is only at very low ranks where the image becomes blurry, although the dogs are still recognizable with only 12 singular values.

There are two reasons why we can compress an image while retaining its overall features. The first is that most of the information in an image comes from a few singular vectors. The distribution of information is skewed as shown in Figure 20.3. The smallest singular vectors give diminishing returns and contribute little to the overall information content of a matrix.

The second reason why we can compress images is that information is distributed. Each pair of singular vectors holds information about every pixel, which is expected since the product $\sigma_i \mathbf{u}_i \mathbf{v}_i$ is a matrix with the same dimensions as the original image. Figure 20.4 gives a visual representation for each pair of singular vectors. Each panel was created by zeroing-out all but one of the singular values. Most of the image has been reconstructed by the time reach the final (smallest) singular values, so representations of these singular vectors often amount to little more than noise. Many machine learning algorithms exploit this feature by "de-noising" inputs by zeroing-out any small singular vectors before using the images.

20.1.1 When does compression save space?

You might have noticed in Figure 20.2 that the SVD with all 384 singular values used *more* memory than the original image (175.2% of the original image size). To

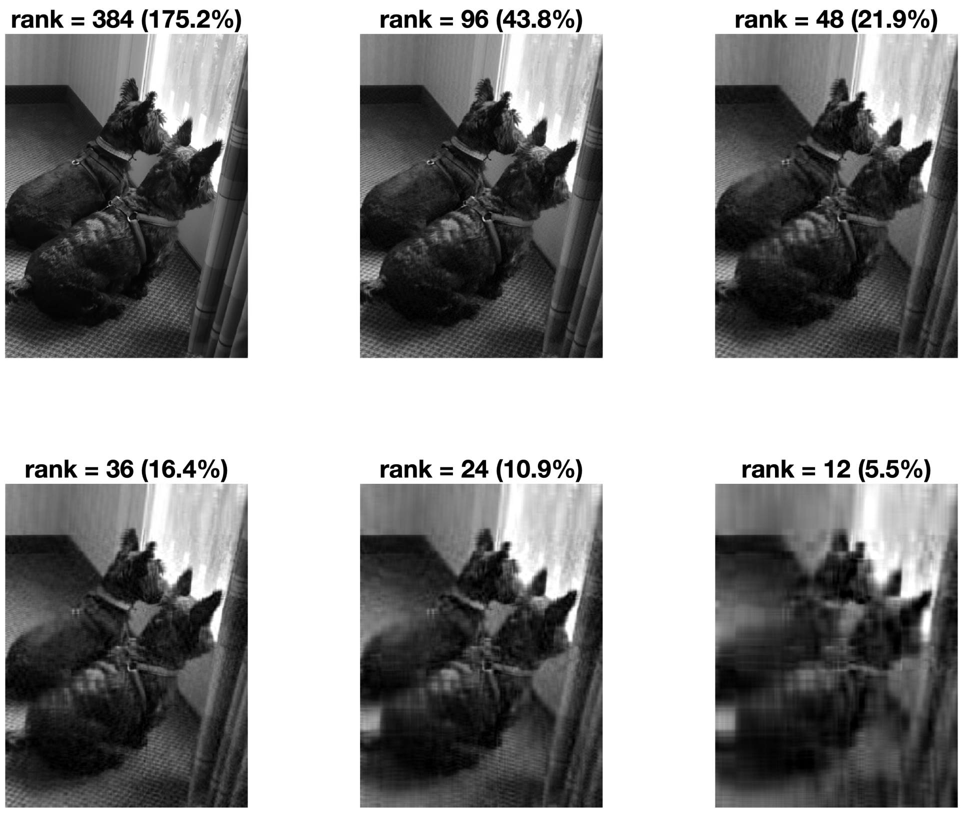


Figure 20.2: Low rank approximations for the image in Figure 20.1. The title of each panel shows the number of singular values used in the approximation (k) and the size of the compressed image relative to the original (%).

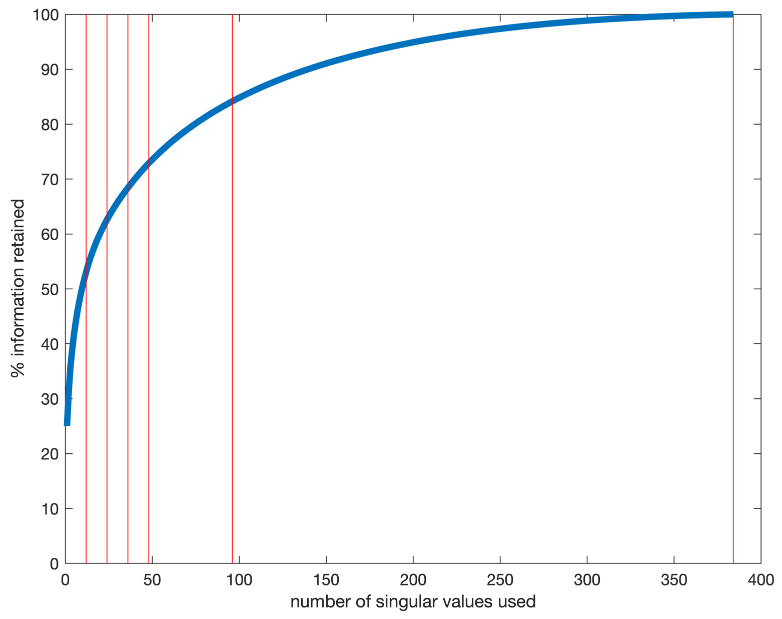


Figure 20.3: The cumulative sum of the singular values in Figure 20.1 reveals that most of the information is stored in the first few singular vectors. The red lines indicate the low rank approximations shown in Figure 20.2.

see why, consider an $m \times n$ image with $m \leq n$. (If m is the larger dimension, we can rotate the image so m is smaller.) This image requires mn units of memory, but the SVD requires

$$\underbrace{m \times m}_{\mathbf{U}} + \underbrace{m \times n}_{\mathbf{\Sigma}} + \underbrace{n \times n}_{\mathbf{V}^\mathsf{T}}.$$

units. Actually, not quite. Since $m < n$, we know that the extra $n - m$ rows in the matrix $\mathbf{V}^\mathsf{T}$ will be dropped, so we don't need to store them. (Matlab's `svd` command removes these rows by default.) Also, The matrix $\mathbf{\Sigma}$ is diagonal, so we only need to store the m nonzero values on the diagonal. The total storage for the full SVD is therefore $m^2 + m + mn$ units of memory, which is larger than the mn units in the original image.

A low rank approximation with only k nonzero singular values keeps only the first k columns of $\mathbf{U}$, k entries in $\mathbf{\Sigma}$, and the first k rows of $\mathbf{V}^\mathsf{T}$. This requires $mk + k + kn = k(m + n + 1)$ units of storage. Compared to the mn units of memory required to store the original image, the rank k approximation by SVD requires

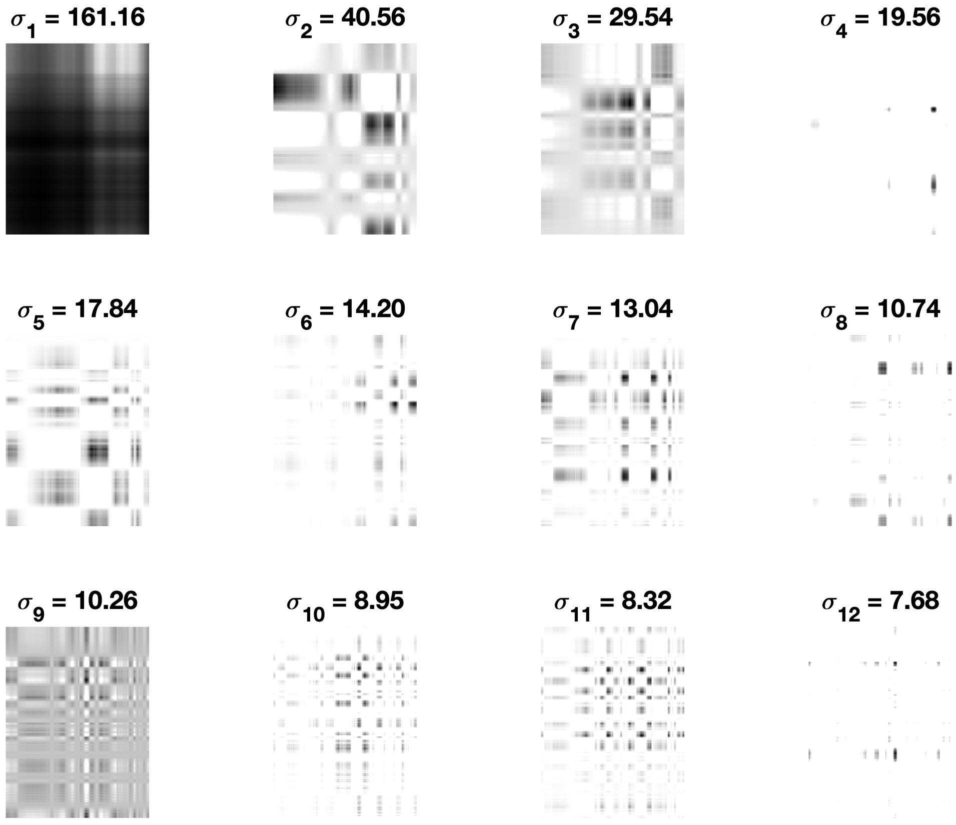


Figure 20.4: Visual representations of each pair of singular vectors. Each panel is the product $\sigma_i \mathbf{u}_i \mathbf{v}_i$ for a single index i. Notice how some of the panels map to rough shapes in the original image. The first panel (σ_1) shows the light from the window. The panels have been rescaled to $[0, 1]$ before plotting.

less memory when

$$k(m+n+1) < mn$$
$$\Rightarrow k < \frac{mn}{m+n+1}.$$

20.2 Recommender Systems

Here's a trick question: What is the missing value in the following matrix?

$$\begin{pmatrix} 2 & 1 & 3 \\ 1 & 2 & 4 \\ 3 & 0 & ? \end{pmatrix}$$

It's a trick question because the missing value could be anything. Any answer is correct! But what if you also knew that the above matrix had rank 2? That's an entirely different story. There are three rows in the matrix, but only two of them would be linearly independent. Any row in the rank 2 matrix must be a linear combination of the other two rows. It appears that row three is twice the first row minus the second row ($2 \times 2 - 1 = 3$ in column one, and $2 \times 1 - 2 = 0$ in column two). So the missing entry must be $2 \times 3 - 4 = 2$. Mystery solved.

Rank deficient matrices like the one above have a remarkable property—a low rank approximation can sometimes be used to fill in missing entries. Filling in missing entries using low rank approximations is called *matrix completion*, and it's a machine learning technique worth billions of dollars. These machine learning techniques predict what a customer will want next given their history of purchases. Amazon shows you products that "you may also like"; Google targets ads based on your search results and Gmail messages; and Netflix populates its frontpage with shows it thinks you'll enjoy. Such recommendations are not guesses, but instead finely tuned algorithms that keep users buying products or spending time on streaming services. The best algorithms are worth huge sums of money, as evidenced by the "Netflix challenge," the company's million dollar contest to improve their recommender engine by 10%. The Netflix challenge was quickly shut down due to privacy concerns. A user's past preferences are so powerful that they can be used to reveal their identify even if the data are anonymized.

20.2.1 Ratings matrices

Online retailers like Amazon collect product ratings from users. Consumers can leave product reviews and rate items with 1–5 stars. We can visualize these data as a ratings matrix like the one below, where each row is a user and each column is an item for sale.

	item 1	item 2	item 3	item 4	item 5
user 1	4		5		
user 2		1			5
user 3	3			5	
user 4		2	3		4

Ignore for a second that Amazon probably has more than four users and five items. Focus instead on how powerful it would be to have the complete ratings matrix. If we could predict how each user would rate every product before they buy it, we could suggest that they purchase the items with high ratings. Completing the ratings matrix creates personalized recommendations for every user! Systems that create recommendations through matrix completion are called *recommender systems*, and they are in use by every retailer and social media site.

There are many methods for matrix completion, but one of the most popular combines several techniques from this book. The algorithm is depicted in Figure 20.5 and is described below. A detailed description of the algorithm is available in section 20.2.4.

1. A ratings matrix $\mathbf{R}$ with m users and n items contains actual ratings for a small number of user/item pairs. The recommender randomly initializes two rank k matrices: $\mathbf{P}_k$ $(m \times k)$ and $\mathbf{Q}_k^\mathsf{T}$ $(k \times n)$.

2. The two matrices $\mathbf{P}_k$ and $\mathbf{Q}_k^\mathsf{T}$ are multiplied to produce a completed ratings matrix $\hat{\mathbf{R}}$. The complete matrix contains estimates for every user/item pair.

3. The known entries in the original ratings matrix $\mathbf{R}$ are compared with the corresponding predictions in the completed matrix $\hat{\mathbf{R}}$. The disagreement between the actual and predicted ratings are used to update the entires in the low rank approximations $\mathbf{P}_k$ and $\mathbf{Q}_k^\mathsf{T}$.

4. The process iterates until the predicted ratings match the observed ratings. The predictions for the unobserved ratings are used to make recommendations.

Recommender systems have become so ubiquitous that some databases are optimized for performing matrix algebra on their data. Thanks to recommender systems, matrix multiplications, linear models, and low rank approximations have become the fundamental operations of data mining.

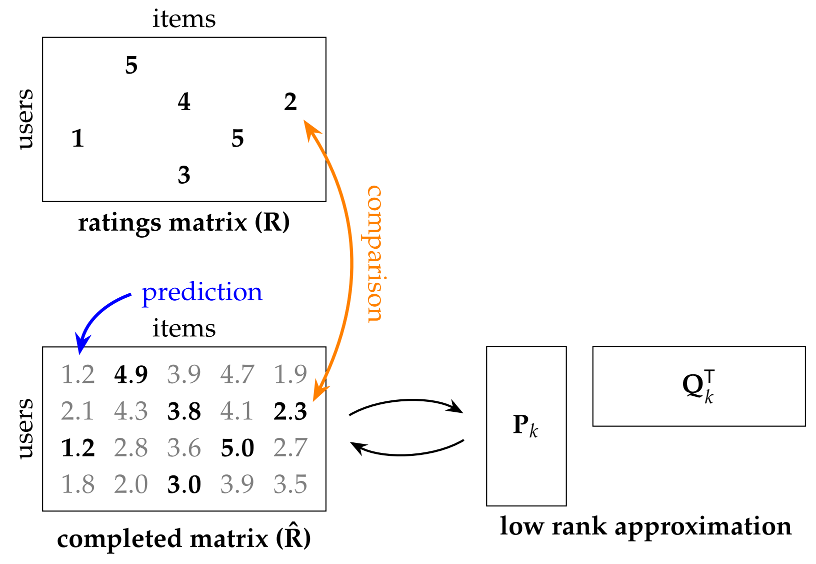


Figure 20.5: Recommender systems use low rank approximations to complete a ratings matrix. The method assumes that the completed ratings matrix is the product of smaller two matrices **P** and $\mathbf{Q}^\mathsf{T}$, each of rank k. Recommender systems search for values of the low rank matrices by comparing the known values in the ratings matrix to the corresponding predictions in the completed matrix.

20.2.2 Implicit vs. explicit ratings

Rating a product on a 1–5 scale is an *explicit* rating. Very few users provide this type of feedback, and you may be wondering how Amazon makes recommendations to you if you've never rated a product. Most recommender systems use *implicit* ratings from their users. Purchasing—or even viewing—an item is a form of rating, since users typically spend their money on things they like (or think they will like). Social media companies record how long you pause over each post when scrolling through your feed. The length of your pause is proportional to your interest in a post and can be used in a ratings matrix to predict what posts to show next. Every click, pause, or "like" on the internet is stored and used to predict ratings. Even brick-and-mortar stores use recommender systems, which is why they want you to join their store's loyalty card to track your purchases. (If you refuse, they can still track purchases across visits through your credit card number.)

The one shortcoming of recommender systems is the *cold start problem*. Before a company has any data on your preferences (implicit or explicit), they cannot make recommendations. This explains why companies request demographic data when you sign up or frequently request ratings on your first purchases. The sooner they can collect data, the better they can customize their offerings to your preferences.

20.2.3 Why low rank approximations work

Ratings matrices are incredibly sparse. Most users have purchased only a tiny fraction of the products available on Amazon, and despite your best efforts at binge watching you have probably seen relatively few of the shows on Netflix. Recommender systems work because users and items are remarkably low rank. Even if Amazon has millions of customers, every customer can be reasonably approximated as a combination of a few thousand "customer types". A few thousand sub-genres are probably all that are necessary to make personalized recommendations for movies.

An interesting observation from recommender systems is that the number of user types must be equal to the number of categories of items. Said more technically, the user rank (row rank) must equal the item rank (column rank) for any ratings matrix, so the ranks of the matrices **P** and **Q** must be the same. Any method that finds similar users (rows of the ratings matrix) can be used equally well to find similar items (columns) by simply transposing the ratings matrix.

20.2.4 * Finding a low rank approximation for a ratings matrix

This section derives the update formulas used to find a low rank approximation for a ratings matrix **R** containing a sparse set of known ratings. We assume the user has specified a rank k for the low rank approximation—this is a hyperparameter that must be tuned to improve predictions.

The completed ratings matrix $\hat{\mathbf{R}}$ is the product of two dense, rank k matrices $\mathbf{P}_k$ and $\mathbf{Q}_k^\mathsf{T}$, so $\hat{\mathbf{R}} = \mathbf{P}_k\mathbf{Q}_k^\mathsf{T}$. Let's focus on a single known rating $\mathbf{R}(u,t)$ from user u for item t, which we'll abbreviate r_{ut}. The predicted value for this rating is

$$\hat{r}_{ut} = \mathbf{P}_k(u,:) \cdot \mathbf{Q}_k^\mathsf{T}(:,t).$$

We can eliminate the transpose on the matrix **Q** by remembering that column t in the transposed matrix is the same as row t in the untransposed matrix, so

$$\hat{r}_{ut} = \mathbf{P}_k(u,:) \cdot \mathbf{Q}_k(t,:).$$

The matrices $\mathbf{P}_k$ and $\mathbf{Q}_k$ are initialized with small, random values, so the predicted rating $\hat{r}_{ut}$ is likely not the same as the actual rating r_{ut}. We can update the entries in $\mathbf{P}_k$ and $\mathbf{Q}_k$ by first calculating the loss between the predicted and actual ratings. Using a quadratic loss function, the loss for a single rating is

$$L(u,t) = (\hat{r}_{ut} - r_{ut})^2 .$$

Let's first update the entries in the matrix $\mathbf{P}_k$ using gradient descent. The entry of the gradient for entry p_{ui} of the matrix $\mathbf{P}_k$ is

$$\frac{\partial L}{\partial p_{ui}} = 2\left(\hat{r}_{ut} - r_{ut}\right)\frac{\partial \hat{r}_{ut}}{\partial p_{ui}}.$$

The partial derivative of the predicted rating is

$$\begin{aligned}\frac{\partial \hat{r}_{ut}}{\partial p_{ui}} &= \frac{\partial}{\partial p_{ui}}\mathbf{P}_k(u,:)\cdot\mathbf{Q}_k(t,:)\\ &= \frac{\partial}{\partial p_{ui}}\sum_{j=1}^{k} p_{uj}q_{tj}\\ &= q_{ti}.\end{aligned}$$

So the gradient of the loss for entry p_{ui} is

$$\frac{\partial L}{\partial p_{ui}} = 2\left(\hat{r}_{ut} - r_{ut}\right)q_{ti}.$$

Similarly, the gradient entry for value q_{ti} in the matrix $\mathbf{Q}_k$ is

$$\frac{\partial L}{\partial q_{ti}} = 2\left(\hat{r}_{ut} - r_{ut}\right)p_{ui}.$$

We can update the entries in $\mathbf{P}_k$ and $\mathbf{Q}_k$ using gradient descent with rules

$$\begin{aligned}p_{ui} &\leftarrow p_{ui} - 2\alpha(\hat{r}_{ut} - r_{ut})q_{ti}\\ q_{ti} &\leftarrow q_{ti} - 2\alpha(\hat{r}_{ut} - r_{ut})p_{ui}\end{aligned}$$

for entries $i \in \{1 \ldots k\}$. Each update uses an estimate of the total loss at a single entry in the observed ratings matrix $\mathbf{R}$. The true loss function would consider all of the known ratings, so our single-rating estimate is a *stochastic gradient*. Surprisingly, stochastic gradient descent is a better global optimizer for many non-convex machine learning problems and requires far less computation per iteration on large problems. However, stochastic gradient descent requires more iterations than deterministic gradient descent, and the step size α often needs to be smaller to ensure stability.

Before starting gradient descent, a subset of the known ratings are held out for cross validation. Large systems with many users and items can easily be overfit because the low rank approximation will contain thousands or millions of entries. A regularization term is added to the loss function to prevent overfitting, and the best regularization parameter is determined by cross validation.

Recommender systems combine several topics from this course: matrix multiplication, rank, loss functions, gradient descent, cross validation, regularization, and matrix decompositions. They are an excellent example of how modern machine learning depends on a solid foundation in linear algebra.