Section 3: All You Need to Know to Understand AI

DO NOT POLLUTE! AVOID PRINTING, OR PRINT 2-SIDED MULTIPAGE.

3.1 Introduction

It is difficult to understand artificial intelligence (AI) without a basic knowledge of mathematics and programming. This section covers the bare minimum topics in calculus, algebra, coding, and probability, required to discuss and understand AI.

These terms may sound technical, but there is no reason to feel intimidated. We will unpack each concept from first principles, complementing abstract concepts with clear, concrete, intuitive examples. Many of these examples will be so simple that they may even seem simple or childish at first, yet they capture the same core ideas that underlie modern AI systems.

3.2 All You Need to Know about Calculus

Functions

A function is like a vending machine: you put something in (say a coin, a bill, or a card), you press your button of choice, and the machine gives you something back (like a snack and some change). If you put in the same coin and press the same button again, you will always get the same snack and change. The machine may be complicated inside, but from the outside, all that matters is that an **input** goes in (coin), you specify some **parameters** (button), and a consistent **output** comes out (snack). Just like that, a mathematical function takes a number as input, follows a fixed rule that depends on some parameters, and produces another number as output.

Functions lie at the core of AI. As a matter of fact, AI systems are just functions: the input to an AI system might be an image, a piece of text, or a sound recording. The output might be a classification of the image, a response to the text, or an interpretation of the sound. As we have seen, however, all of these forms of data (images, text, sounds) are ultimately represented as numbers inside the computer. The same is true of the outputs (labels), which are also encoded numerically.

This means that, at its core, an AI system can be understood as a process that receives a collection of numbers as input and returns another collection of numbers as output. This is precisely what a function does: it takes numbers in and produces numbers out. In other words, AI systems are nothing more than very sophisticated functions.

Therefore, understanding functions is essential for understanding AI.

Formal Definition

Formally, a function f assigns each element of one set, called *domain*, to an element of another set, called *range*. The domain and range sets are typically denoted as $\mathcal{X}$ and $\mathcal{Y}$. This can be summarized in mathematical notation as

$$f : \mathcal{X} \to \mathcal{Y}.$$

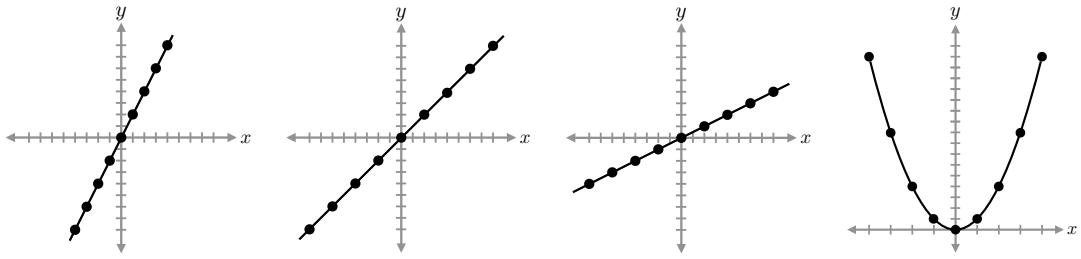
The elements of the domain, denoted as x are called *inputs*. The elements of the range, denoted as $f(x)$ or y, are called *outputs*. The function is specified by a rule or formula that describes the relationship between inputs and outputs. For example,

$$f(x) = 2x$$

is a function that maps any number to its double. For example, f maps 3 to 6.

Plots

Functions are often visualized using plots where the where the horizontal axis (typically called the x-axis) represents inputs, the vertical axis (typically called the y-axis) represents outputs, and the related pairs are marked with points (which together usually form segments). Here are a few examples:



$f(x) = 2x$ $f(x) = x$ $f(x) = \frac{1}{2}x$ $f(x) = x^2$

Types of Functions and Parameters

Depending on the type of relationship that they describe, functions can be classified into several groups, for example:

- Linear functions: $f(x) = ax + b$, where a and b are parameters that determine the slope (angle) and the offset (distance from the origin).
- Polynomial functions: $f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_dx^d$, where $a_0, \ldots, a_d$ are parameters and d is the *degree*.
- Exponential functions: $f(x) = ce^{ax+b}$, where a, b, and c are parameters.
- Logaritmic functions: $f(x) = c\log(ax + b)$, where a, b, and c are parameters.
- Trigonometric functions: $f(x) = c\sin(ax + b)$, where a, b, and c are parameters.

AI as a Function

AI often focuses on prediction and classification. For example, we may want to predict the risk that a patient will develop cancer based on their health history, genetic markers, and lifestyle information, or we may want to classify bacteria as drug-resistant or drug-susceptible based on laboratory measurements. These and any other AI task can be framed as functions: the input is the data encoded in the numerical vector $\mathbf{x}$, and the AI system is the function f whose output is the numerical label y.

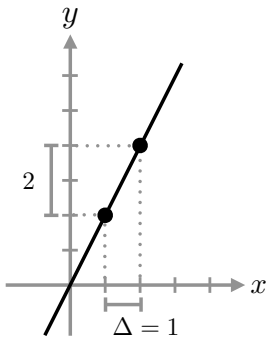
Derivatives

Ultimately, the parameters of a function determine how an AI system behaves. These parameters are the adjustable internal settings that control how inputs are turned into outputs. For example, the parameter a in the function $f(x) = ax$ determines how the input will be scaled to produce the output. If $a = 2$, the output will be the double of the input; if $a = 3$, the output will be the triple of the input. Learning is precisely the process of finding the parameter values that make the system produce the desired outputs.

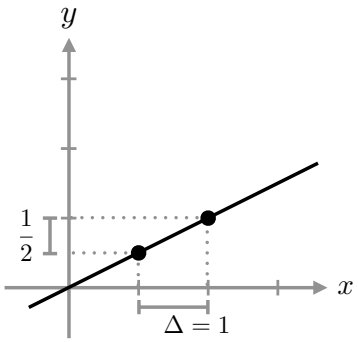
The general method used to find these parameters is called **gradient descent**. You can think of it as repeatedly asking: *if I nudge this parameter a little, does the output improve or get worse?* Derivatives provide exactly this information: they tell the system in which direction to adjust each parameter. By taking many small, guided steps, the AI gradually settles on parameter values that work well across the training data and, ideally, also generalize to new, unseen data.

The derivative of a function f, denoted as f', measures how the function's output changes in response to changes in its input. It provides a precise way to quantify the rate of change or the *slope* of the function at any given point.

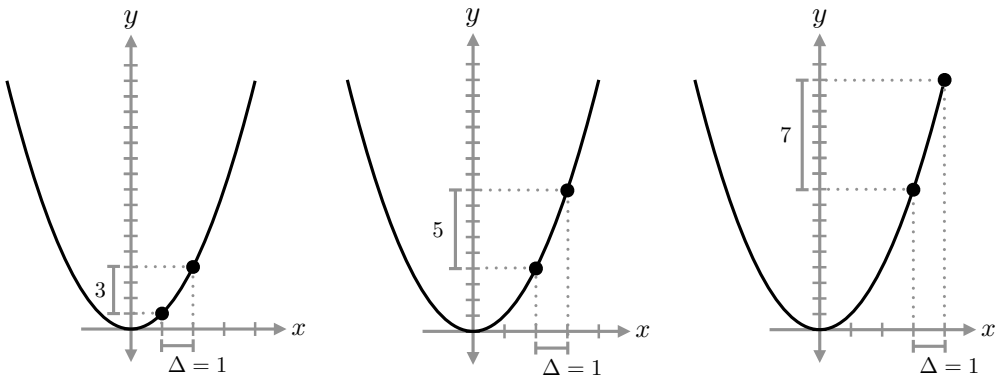
To build some intuition, consider our previous function $f(x) = 2x$: whenever x changes a distance $\Delta = 1$, $f(x)$ changes a distance of 2:



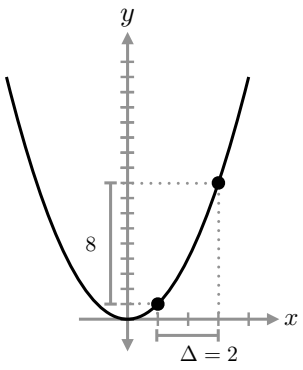
It turns out that the derivative of this function is $f'(x) = 2$. Similarly, for the function $f(x) = \frac{1}{2}x$, we have that whenever x changes a distance $\Delta = 1$, $f(x)$ changes a distance of $\frac{1}{2}$:



It turns out that the derivative of this function is $f'(x) = \frac{1}{2}$. In general, the derivative of any linear function of the form $f(x) = ax + b$ happens to be $f'(x) = a$ for every x. In general, the slope (derivative) of a function depends on the point we are looking at. For example, consider $f(x) = x^2$. A change of $\Delta = 1$ from $x = 1$ to $x = 2$ produces a change in $f(x)$ equal to 3, whereas the same change from $x = 2$ to $x = 3$ produces a change in $f(x)$ equal to 5, and the same change from $x = 3$ to $x = 4$ produces a change in $f(x)$ equal to 7:

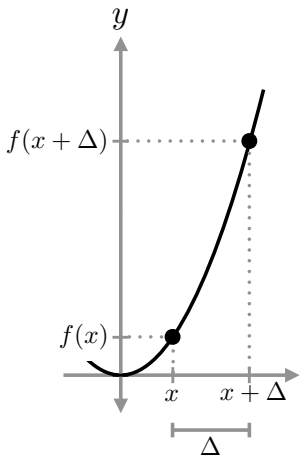


Moreover, the change in $f(x)$ generally also depends on the size of the change in x. For example, a change of $\Delta = 2$ from $x = 1$ to $x = 3$ produces a change in $f(x)$ of 8:



Hence, to really know the slope of $f(x)$ at any given point x, we need to consider the change in $f(x)$ as Δ when as small as possible. This results in the formal definition of the derivative:

$$f'(x) := \lim_{\Delta \to 0} \frac{f(x + \Delta) - f(x)}{\Delta}.$$



This definition may look a bit scary. However, the derivative of most functions can be calculated easily without the need to compute this limit. For example, the derivative of polynomial functions can be obtained with a simple trick, often called the *power rule*:

The power rule. For each term in a polynomial of the form ax^d, simply multiply the coefficient a by the exponent d and reduce the exponent by 1 to obtain dax^{d-1}.

For example, we can use this trick to see that the derivative of $f(x) = x^2$ is equal to $f'(x) = 2x$.

The process through which AI systems learn is essentially by computing derivatives to tweak their parameters the right way until they produce the desired behavior.

3.3 All You Need to Know about Algebra

At this point we have looked at functions $f(x)$ that depend on a single variable, x. This could be very useful if, for example, we wanted to predict the glucose level of a person as a function of their weight. However, it is unlikely that we could make an accurate prediction based on a single variable. In general, responses such as the glucose level may depend on many variables, such as height, weight, diet, physical activity, genetics, and more. Linear algebra offers us a simple way to organize and manipulate all those variables in a structured and systematic way, through *vectors* and *matrices*.

Vectors

A vector is simply an ordered list of numbers, for example:

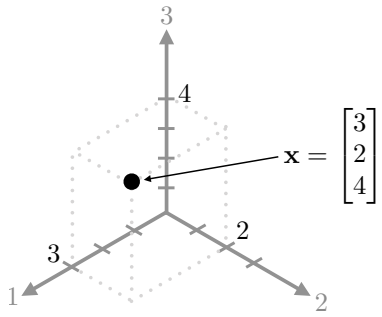
$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}.$$

That's it!

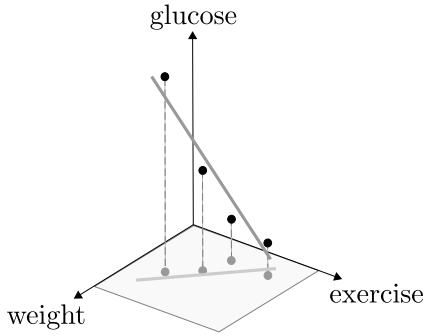
Recall that we use **bold** notation to indicate that $\mathbf{x}$ is a vector, as opposed to x, which is a scalar (single variable). We use $\mathbf{x} \in \mathbb{R}^{\mathrm{D}}$ to indicate that $\mathbf{x}$ is a vector with D entries, each of which is a real-valued number. Vectors come in all sizes, depending on what they represent. For example, we can use a vector $\mathbf{x} \in \mathbb{R}^7$ to store the information of a patient, where its entries represent: height, weight, age, sex, number of cigarettes they smoke per week, cups of wine (or equivalent) they drink per week, and hours of physical activity per week:

$$\mathbf{x} = \begin{bmatrix} 177 \\ 150 \\ 30 \\ 1 \\ 0 \\ 2 \\ 6 \end{bmatrix} \begin{array}{l} \leftarrow \text{height} \\ \leftarrow \text{weight} \\ \leftarrow \text{age} \\ \leftarrow \text{sex} \\ \leftarrow \text{cigarettes} \\ \leftarrow \text{alcohol} \\ \leftarrow \text{activity.} \end{array}$$

Vectors can be interpreted as points in a *space* where each entry of the vector corresponds to a coordinate in the space. For example:



Whenever the entries of a vector are associated to *features* (e.g., height or weight), vectors are called **feature vectors** and their coordinate space is called **feature space**. This way, the points defined by the vectors represent **samples** that can often be visualized to obtain insights and find patterns. For example, consider the following data depicted in feature space:



In this figure, each black point represents a sample corresponding to a patient. From this data visualization we can see a trend showing that glucose increases with weight and decreases with exercise.

Matrices

Similar to vectors, matrices are rectangular arrays of numbers. For example:

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

That's it!

Recall that we use **bold** and capitalized notation to indicate that $\mathbf{X}$ is a matrix, as opposed to a vector $\mathbf{x}$ or a scalar x. We use $\mathbf{X} \in \mathbb{R}^{\mathrm{D}\times\mathrm{N}}$ to indicate that $\mathbf{X}$ is an $\mathrm{D} \times \mathrm{N}$ matrix with a real-valued entries. Matrices are very handy structures to arrange and manipulate vectors. For example, consider the following vectors in $\mathbb{R}^7$ corresponding to 3 samples (patients):

$$\mathbf{x}_1 = \begin{bmatrix} 177 \\ 150 \\ 30 \\ 1 \\ 0 \\ 2 \\ 6 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 160 \\ 170 \\ 50 \\ 1 \\ 5 \\ 10 \\ 0 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} 165 \\ 140 \\ 20 \\ 0 \\ 0 \\ 0 \\ 10 \end{bmatrix} \begin{matrix} \leftarrow \text{height} \\ \leftarrow \text{weight} \\ \leftarrow \text{age} \\ \leftarrow \text{sex} \\ \leftarrow \text{cigarettes} \\ \leftarrow \text{alcohol} \\ \leftarrow \text{activity.} \end{matrix}$$

We can store all these vectors into a 3×7 matrix, where each row represents a feature, and each column represents a sample:

$$\mathbf{X} = \begin{bmatrix} 177 & 160 & 165 \\ 150 & 170 & 140 \\ 30 & 50 & 20 \\ 1 & 1 & 0 \\ 0 & 5 & 0 \\ 2 & 10 & 0 \\ 6 & 0 & 10 \end{bmatrix}.$$

Matrix Operations

Matrices are particularly useful because they allow us to perform lots of operations between the features of all the samples in a systematic and seamless way. For example, suppose that we discover somehow that the magic formula to estimate the cancer risk of a patient as a function of its features above is:

$$risk = \frac{1}{1000}\left(-1 \cdot height + 2 \cdot weight + \frac{1}{2} \cdot age + 23 \cdot cigarettes + 27 \cdot alcohol - 10 \cdot exercise\right).$$

Suppose we want to estimate the risk of the patient with feature vector $\mathbf{x}_1$ above. Then we would have to compute:

$$risk = \frac{1}{1000}\left(-1(177) + 2(150) + \frac{1}{2}(30) + 23(0) + 27(2) - 10(6)\right). \tag{3.1}$$

That is already a lot to keep track of. If we had many more features, as is typically the case in modern datasets, keeping track of all these individual operations would quickly become overwhelming. This is why we use matrix operations: they allow us to perform many calculations at once while keeping everything organized, concise, and manageable.

- **Matrix Multiplication.** Given matrices $\mathbf{A} \in \mathbb{R}^{m \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times n}$, their product, denoted by $\mathbf{AB}$, is another matrix of size $m \times n$ whose $(i,j)^{th}$ entry is given by:

$$[\mathbf{AB}]_{ij} \;=\; \sum_{\ell=1}^{k} \mathbf{A}_{i\ell}\mathbf{B}_{\ell j}.$$

  Intuitively, the $(i,j)^{th}$ entry of $\mathbf{AB}$ is given by the multiplication of the i^{th} row of $\mathbf{A}$ and the j^{th} column of $\mathbf{B}$. For example, if

$$\mathbf{A} \;=\; \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \qquad \mathbf{B} \;=\; \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}, \tag{3.2}$$

  Then:

$$\mathbf{AB} \;=\; \begin{bmatrix} (1\cdot 1 + 2\cdot 3 + 3\cdot 5) & (1\cdot 2 + 2\cdot 4 + 3\cdot 6) \\ (4\cdot 1 + 5\cdot 3 + 6\cdot 5) & (4\cdot 2 + 5\cdot 4 + 6\cdot 6) \end{bmatrix} \;=\; \begin{bmatrix} 22 & 28 \\ 49 & 64 \end{bmatrix}.$$

- **Scalar Multiplication** Given a scalar c and a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, their product, denoted by $c\mathbf{A}$, is an $m \times n$ matrix whose $(i,j)^{th}$ entry is given by c times the $(i,j)^{th}$ entry of $\mathbf{A}$. For example, with $\mathbf{A}$ as in (3.2), and $c = 7$,

$$c\mathbf{A} \;=\; 7\mathbf{A} \;=\; \begin{bmatrix} 7 & 14 & 21 \\ 28 & 35 & 42 \end{bmatrix},$$

- **Transposition.** The transpose of a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, denoted by $\mathbf{A}^\mathsf{T}$, is an $n \times m$ matrix whose $(i,j)^{th}$ entry is given by the $(j,i)^{th}$ entry of $\mathbf{A}$. Intuitively, transposing a matrix is like flipping its rows and columns along the diagonal. For example, with $\mathbf{A}$ as in (3.2),

$$\mathbf{A}^\mathsf{T} \;=\; \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

- **Identity Matrix.** The identity matrix of size $m \times m$, denoted by $\mathbf{I}_m$, is a squared matrix whose diagonal entries are all ones, and off-diagonal entries are all zeros. For example, the 3×3 identity matrix is

$$\mathbf{I}_3 \;=\; \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

  Whenever there is no possible confusion about the size of the identity matrix, people often drop the m subindex, and simply denote it as $\mathbf{I}$.

- **Trace.** Given a squared matrix $\mathbf{A} \in \mathbb{R}^{m \times m}$, its trace, denoted by $\mathsf{tr}(\mathbf{A})$ is the sum of its diagonal entries:

$$\mathsf{tr}(\mathbf{A}) \;=\; \sum_{i=1}^{m} \mathbf{A}_{ii}.$$

  For example, $\mathsf{tr}(\mathbf{I}_3) = 3$.

- **Inverse.** Given a squared matrix $\mathbf{A} \in \mathbb{R}^{m \times m}$, its inverse, denoted by $\mathbf{A}^{-1}$ is a matrix such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. For example, $\mathbf{I}^{-1} = \mathbf{I}$. As another example, consider

$$\mathbf{A} \;=\; \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{bmatrix}.$$

  Then

$$\mathbf{A}^{-1} \;=\; \begin{bmatrix} 3 & -3 & 1 \\ -2.5 & 4 & -1.5 \\ 0.5 & -1 & 0.5 \end{bmatrix}.$$

  You can verify that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. Here is a special trick to invert 2×2 matrices:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} \;=\; \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}.$$

  Of course, this requires that $ad - bc \neq 0$.

- **Hadamard Product.** Given two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$, their Hadamard product (aka point-wise product) is a matrix of size $m \times n$, denoted as $\mathbf{A} \odot \mathbf{B}$, whose $(i,j)^{th}$ entry is given by the product of the (i,j) entries of $\mathbf{A}$ and $\mathbf{B}$, i.e.,

$$[\mathbf{A} \odot \mathbf{B}]_{ij} \;=\; \mathbf{A}_{ij}\mathbf{B}_{ij}.$$

  For example, with $\mathbf{A}, \mathbf{B}$ as in (3.2),

$$\mathbf{A} \odot \mathbf{B}^{\mathsf{T}} \;=\; \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \odot \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \;=\; \begin{bmatrix} 1 & 6 & 15 \\ 8 & 20 & 36 \end{bmatrix}.$$

- **Vector Operations.** Notice that vectors are 1-column matrices, so all matrix operators that apply to non-squared matrices also apply to vectors.

- **Norms.** Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, its Frobenius norm is defined as:

$$\|\mathbf{A}\|_F \;:=\; \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} \mathbf{A}_{ij}^2}.$$

  In the particular case of vectors, this is often called the ℓ_2-norm or Euclidean norm, and is denoted by $\|\mathbf{x}\|_2$, or simply $\|\mathbf{x}\|$. Norms are important because they essentially quantify the *size* of a matrix or vector. Just as there are several ways to quantify the size of a person (e.g., age, height, weight), there are also several ways to quantify the *size* of a matrix or vector, for which we can use different norms. Another example is the ℓ_1-norm, which for matrices is defined as:

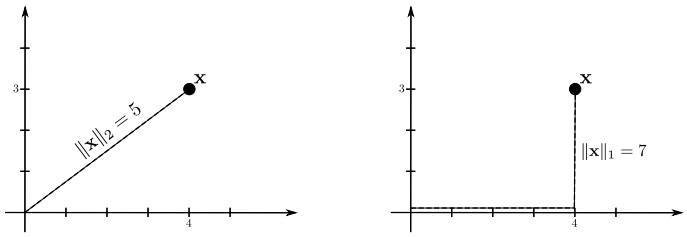
$$\|\mathbf{A}\|_1 \;:=\; \max_j \sum_{i=1} |\mathbf{A}_{ij}|,$$

  and for vectors is defined as

$$\|\mathbf{x}\|_1 \;:=\; \sum_j |x_j|,$$

  also known as the taxi-cab or Manhattan norm. Intuitively, the ℓ_2-norm measures the *point-to-point* distance, while the ℓ_1-norm measures the *taxi-cab* distance. For example, for the same vector $\mathbf{x} = [4 \;\; 3]$, here are two different notions to quantify its size:

$$\|\mathbf{x}\|_2 \;=\; \sqrt{4^2 + 3^2} = 5 \qquad\qquad \|\mathbf{x}\|_1 \;=\; |4| + |3| = 7.$$



The norm $\|\mathbf{x}\|$ of a vector $\mathbf{x}$ is essentially its size. Norms are also useful because they allow us to measure distance between vectors (through their difference). For example, consider the following images:

and vectorize them to produce vectors $\mathbf{x}, \mathbf{y}, \mathbf{z}$. We want to do face clustering, i.e., we want to know which images correspond to the same person. If $\|\mathbf{x}-\mathbf{y}\|$ is small (i.e., $\mathbf{x}$ is similar to $\mathbf{y}$), it is reasonable to conclude that the first two images correspond to the same person. If $\|\mathbf{x}-\mathbf{z}\|$ is large (i.e., $\mathbf{x}$ is very different from $\mathbf{z}$), it is reasonable to conclude that the first and second images corresponds to different persons.

Norms satisfy the so-called *triangle inequality*: $\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$. This allows you to draw useful conclusions. For example, knowing that $\|\mathbf{x}-\mathbf{y}\|$ is small and that $\|\mathbf{x}-\mathbf{z}\|$ is large allows us to conclude that $\|\mathbf{y}-\mathbf{z}\|$ is also large. Intuitively, this allows us to conclude that if $\mathbf{x}, \mathbf{y}$ correspond to the same person, and $\mathbf{x}$ and $\mathbf{z}$ corresponds to different persons, then $\mathbf{y}$ and $\mathbf{z}$ also correspond to different persons. In other words, nothing weird will happen.

- **Inner Product.** Given two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$, their inner product is defined as:

$$\langle \mathbf{A}, \mathbf{B} \rangle \;:=\; \mathsf{tr}(\mathbf{A}\mathbf{B}^{\mathsf{T}}).$$

For example, with $\mathbf{A}$ as in (3.2),

$$\langle \mathbf{A}, \mathbf{A} \rangle \;=\; \mathsf{tr}(\mathbf{A}\mathbf{A}^{\mathsf{T}}) \;=\; \mathsf{tr}\left(\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}\begin{bmatrix}1 & 4\\ 2 & 5\\ 3 & 6\end{bmatrix}\right) \;=\; \mathsf{tr}\left(\begin{bmatrix}14 & 32\\ 32 & 77\end{bmatrix}\right) \;=\; 91.$$

Notice that for vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$, $\mathbf{x}^{\mathsf{T}}\mathbf{y}$ will always be a scalar, so we can drop the trace, and simply write $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}\mathbf{y}^{\mathsf{T}}$. For example, with the same setup as above,

$$\langle \mathbf{w}, \mathbf{x}_1 \rangle \;=\; \mathbf{w}\mathbf{x}_1^{\mathsf{T}} \;=\; 132.$$

Inner products are of particular importance because they measure the similarity between matrices and vectors. In particular, the angle θ between two vectors is given by:

$$\cos\theta \;=\; \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\|\|\mathbf{y}\|}.$$

More generally, the larger the inner product between two objects (in absolute value), the more similar they are.

All these matrix operators are useful because they allow us to write otherwise complex burdensome operations in simple and concise matrix form. For example, instead of the cumbersome expression in (3.1), we could just construct the *weight* vector

$$\mathbf{w} = \begin{bmatrix} -1 \\ 2 \\ \frac{1}{2} \\ 0 \\ 23 \\ 27 \\ -10 \end{bmatrix}$$

and write

$$risk = \frac{1}{1000}\mathbf{w}^\mathsf{T}\mathbf{x}_1 = .$$

It is also convenient that all these operations are implemented in most coding languages. Hence, we do not even need to perform them manually. For example, to compute $\mathbf{w}^\mathsf{T}\mathbf{x}_1$ in Matlab we just need to write:

```
w = [-1; 2; .5; 0; 23; 27; -10];
x1 = [177; 150; 30; 1; 0; 2; 6];
w'*x1
```

3.4 All You Need to Know about Coding

One of the greatest achievements of modern AI are large language models (LLMs) like as ChatGPT. Their main appeal is that they allow us to communicate with computers using natural language. That is, in much the same way we communicate with other humans. Intuitively, it is like these computers can *speak human*. We can give them instructions, ask them questions, and describe them tasks in everyday language. Sometimes the system understands us well, and sometimes it does not.

Programming, by contrast, is like speaking the computers own language. When we write code, we tell the computer exactly what to do, step by step, leaving no room for ambiguity or interpretation.

Just as there are many human languages, such as English, Spanish, Swahili, or Mandarin, there also exist many computer languages. Examples include Python, Julia, Java, Matlab, and C++. These languages differ mostly in syntax. For example, the following Python and Matlab codes tell the computer to display the items stored in a list (for now don't focus on understanding how these codes works; we will do that later; for now just look at their similarity):

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

```
fruits = {"apple", "banana", "cherry"};
for i = 1:length(fruits)
    disp(fruits{i});
end
```

Notice that there are only a few subtle differences in syntax, such as squared brackets instead of curly brackets, or the use of *disp* instead of *print*, or the colons and semicolons. Therefore, if you learn one coding language, you can translate to others without too much trouble.

In this course we will be using mostly Python, because it has lots of *libraries* (collections of *pre-coded* functions) that we can use to our advantage. For example, instead of coding ourselves all the matrix operations above, we can simply *import* the *NumPy* library, where someone else already coded all those operations for us. For instance, coding matrix multiplication from scratch would look something like this (again, for now don't focus on understanding how this code works; we will do that later; for now just look at its complexity):

```
# Ensure matrices A and B are compatible for multiplication
rows_A = len(A)
cols_A = len(A[0])
rows_B = len(B)
cols_B = len(B[0])

if cols_A != rows_B:
    raise ValueError("Number of columns in A must equal the number of rows in B.")

# Initialize the result matrix with zeros
result = [[0 for _ in range(cols_B)] for _ in range(rows_A)]

# Perform matrix multiplication
for i in range(rows_A):
    for j in range(cols_B):
        for k in range(cols_A):
            result[i][j] += A[i][k] * B[k][j]

# Print the result
for row in result:
    print(row)
```

Instead, we can simply use the NumPy library that already has this operation as the pre-defined function *dot*:

```
import numpy
result = numpy.dot(A, B)
print(result)
```

Similar to NumPy, there exists lots of other Python libraries that contain pre-coded functions useful for AI. We will be using these libraries throughout this course, but in order to do so we first need to understand how coding works.

The Building Blocks of Coding

A code is essentially a list of instructions. Each of these instructions tells the computer to do something. Surprisingly, there are only 6 basic instructions! In other words, coding is like speaking a language with only 6 words! Here they are:

1. **Assignment.** This is the process of assigning values to variables. For example:

```
x = 5
```

```
y = 20
z = x+y
s = "Hello"
```

is telling the computer that the variables x and y take the values 5 and 20, respectively, that z takes whatever is the value of $x + y$, which is $5 + 20 = 25$, and that the variable s takes the value "Hello".

2. **Print.** Intuitively, this function asks the computer to *tell* something to us. For example:

```
print(s)
```

is asking the computer to tell us what is the value of the variable s. In this case, the computer would tell us "Hello"

3. **If/Else.** The *if/else* instruction has three components: (i) a condition, (ii) a list of instructions to execute in case the condition is met, and (iii) an optional list of instructions to execute in case the condition is not met. Here is an example:

```
if z < 5:
    a = 1
else:
    a = 0
```

Here (i) the condition is $x > 5$. (ii) The list of instructions to execute in case the condition is met is $a = 1$. (iii) The optional list of instructions to execute in case the condition is not met is $a = 0$. In this case, since $z = 25$, which is larger than 5, only the second list of instructions will be executed, and hence a will take a value of 0.

The *else* instruction can be omitted if the second list of instructions is empty.

4. **For.** The *for* loop is a list of instructions to be *repeated* a specified number of times. Here is an example:

```
for i in [1,2,3,4,5,6,7,8,9,10]:
    a = a + i
```

This code is going to repeat the instruction $a = a + i$ ten times. Each time, the variable i will take a different value in the provided list.

5. **While.** The *while* loop has two components (i) a condition, and (ii) a list of instructions to *repeat* as long as the condition is met. Here is an example:

```
while z > 5:
    print(z)
    z = z-1
```

This code will repeat the instructions print(z) and z=z-1 so long as z is larger than 5. Notice that these instructions iteratively change the value of z, otherwise these instructions would repeat indefinitely.

6. **Functions.** A coding function is nothing more than a sequence of instructions, packed in a way that can be reused. They are particularly useful when there is a sequence of instructions that you want to use multiple times. They are also useful to keep code tidy and organized. Coding functions have four main components: (i) the name of the function, (ii) an optional input, (iii) an optional output, and (iv) the sequence of instructions that will be executed any time that the function is called. Here is an example of a function that computes the inner product of two vectors of the same size:

```
def inner_product_vectors(x,y):
  inner_product = 0
  for i in range(len(x)):
    inner_product = inner_product + x[i]*y[i]
  return inner_product
```

(i) The name of the function always comes after the instruction *def*, which essentially tells the computer that you are about to define a new function. In this example, the name of the function is *inner_product_vectors*, but we could have chosen *any* name we wanted. Nonetheless, it is generally useful to name functions as something intuitive related to what they do. (ii) The optional inputs come right after the name in parentheses. In this case, the inputs are the vectors x and y. (iii) The optional outputs are the variables that come after the instruction *return*. (iv) All other instructions below *def* and before *return* will be repeated any time that the function is called. Once we define a function, we can call it whenever we want by simply using its name. For example:

```
result = inner_product_vectors(x,y)
```

Since inner products are a critical part of manipulating vectors, and we will be using vectors to represent data, we expect to compute lots of inner products. Hence, it makes sense to pack this operation in a function, so that we can reuse it many times. Libraries like NumPy are essentially collections of lots of functions like this, which we can import easily with a simple command:

```
import numpy
```

After that, we can run the functions in the library with the following syntax:

```
numpy.inner(x,y)
```

The fun part is that we can combine these 6 simple instructions to tell the computer what we want it to do. For example, if I want the computer to say "Hello" 10 times, I can use the following code:

```
s = "Hello"
for i in [1,2,3,4,5,6,7,8,9,10]:
    print(s)
```

Like with any language, there may be many ways to say the same thing. For example, we can use a while loop to achieve the same goal:

```
n = 0
s = "Hello"
while n<10:
    print(s)
    n = n+1
```

Alternatively, we can use the *predefined* Python function *range(*a,b*)*, which produces a list with the digits starting with a and ending *before* b.

```
for i in range(1:11):
    print("Hello")
```

This might be a better alternative than the first option, especially if we want to repeat the print instruction many times, not just 10. Hopefully, at this point there is nothing obscure or intimidating about the *range* function. In fact, with the tools you know at this point you should be able to code it yourself. Here is one way:

```
def my_range(a,b):
  L = []
  i = a
  while i<b:
    L.append(i)
    i = i+1
  return(L)
```

We can verify that it works the same as Python's *range* function as follows:

```
print("Result using Python's range function:")
for i in range(1,11):
  print(i)
print("Result using my range function:")
for i in my_range(1,11):
  print(i)
```

If we ran this code, we would obtain the following result:

```
Result using Python's range function:
1
2
3
4
5
6
7
8
9
10
Result using my range function:
1
2
3
4
5
6
7
8
9
10
```

Coding Tricks and Nuisances

Naturally, there are lots of tricks to make code more efficient and concise. For example, we can code the same inner product as above in a single line:

```
result = sum([x[i] * y[i] for i in range(len(x))])
```

There are also lots of libraries that implement matrix operations, functions to read and write files, to process images, to run AI algorithms, and more. Part of being a proficient programmer involves being familiar with useful libraries and becoming comfortable learning how to use them, which comes with practice. Similarly, there are often nuisances we need to be careful about. For example, sometimes it is necessary to transform variables into specific types (strings, integers, floats, etc.) with *casting* functions like:

```
x = str(3)
y = int(3)
z = float(3)
```

A good programmer also knows how to identify and handle these nuisances, which also takes practice.

Where to Write and How to Run my Code?

There are several tools you can use to write and run your code. These range from very basic tools, like *TextEdit* and the *Terminal*, to Integrated Development Environments (IDEs) like *Visual Studio Code* and Cloud tools like *Google Cloud Platform*. There is typically a trade-off between the amount of control and the ease of use that each option offers. For example, IDEs allow you to customize lots of options, like which resources (CPUs or GPUs) you want to assign to each job. However, this may require you to deal with additional nuisances, like installing specific packages for your operating system, configurating hardware drivers, and verifying versions compatibilities.

In this course we will use one tool that takes care of most nuisances for us, so that we do not have to install anything, and we can get to coding right away. This tool is called *Google Colab*. It allows you to write your code directly on a web app, and run it on Google's servers, which already have installed and configured everything you need. To use it you need to create a Google Drive account, which allows you to manage the files you'll have on Google's servers. Once you have an account, you can just go to `https://drive.google.com`, create a new Colab file and start coding. These steps are summarized in Figure 3.1, but there are many additional resources online to guide you step by step.

3.5 All You Need to Know about Probability

Many (if not all) data in AI systems involve randomness. I hope these few examples below help convincing you of the ubiquitousness and rich nature of such randomness. Probability theory allows us to model randomness in a precise mathematical way. Furthermore, despite the uncertainty produced by randomness, probability theory allows us to draw *likely* conclusions in a sensible manner, and quantify how certain we are about these conclusions.

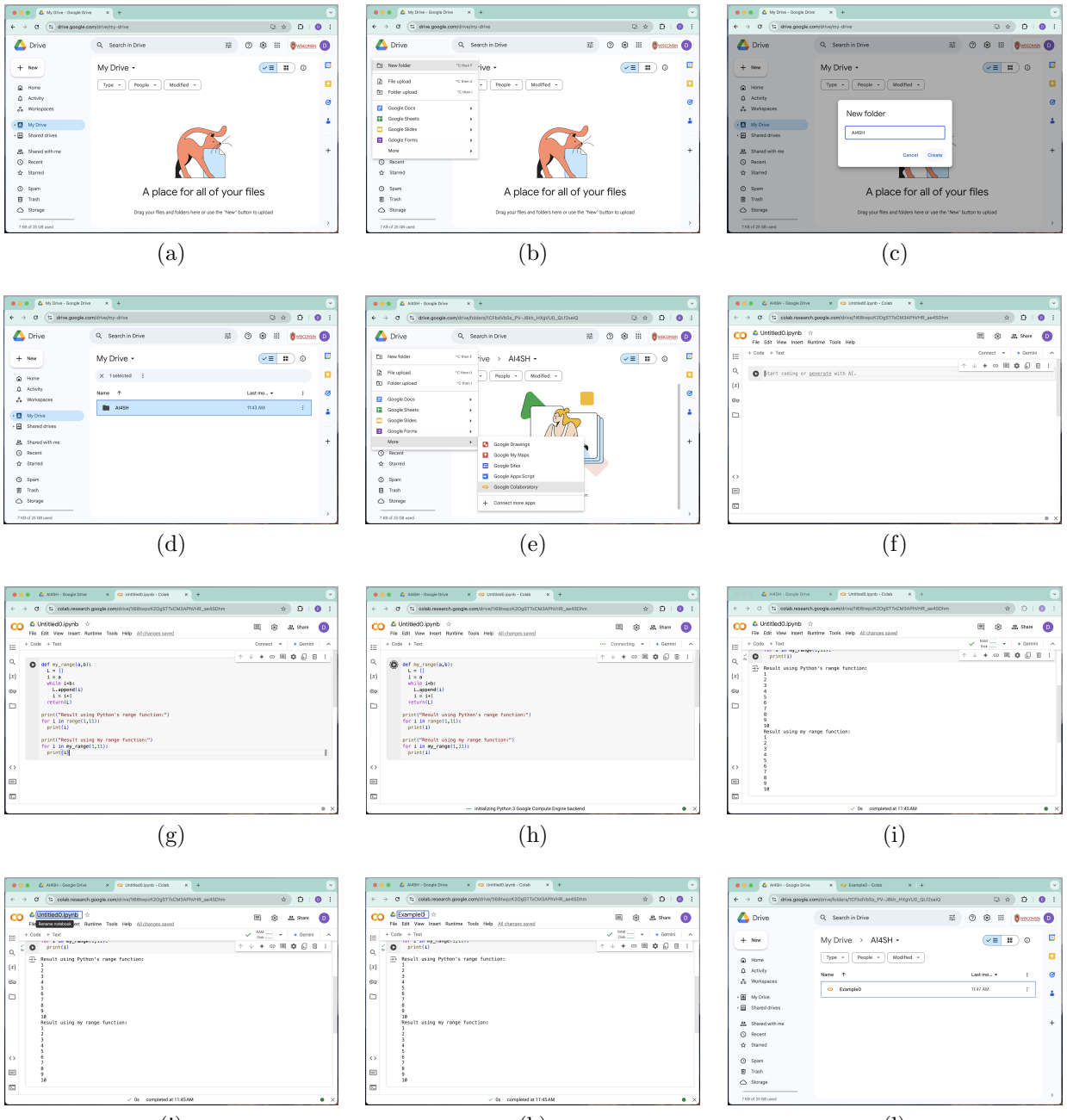


Figure 3.1: **Getting started with Google Colab.** Go to `https://drive.google.com`. After creating an account and log in, you should see something like **(a)**. Here you can click on *New* to **(b)** create a folder and **(c)** name it however you want (e.g., AI4SH). **(d)** This folder will now appear in your main *My Drive* repository. You can access it, and **(e)** click *New*, *More*, and Google Colaboratory to create a Colab file. **(f)** Your browser will open your new Colab file on a new tab. **(g)** Here you can type your code, for example, our *my_range* function above. **(h)** You can run this code by clicking the *run* (play) button. At this point, your code will start running on Google's servers. **(i)** Once Google's computers are done, they will display the output. **(j)** You can rename your Colab file any way you want at any point. **(k)** Here I renamed it Example0. **(l)** This file is now accessible on your Drive repository, under the folder you created, and you can access/edit later.

Randomness in Computer Vision

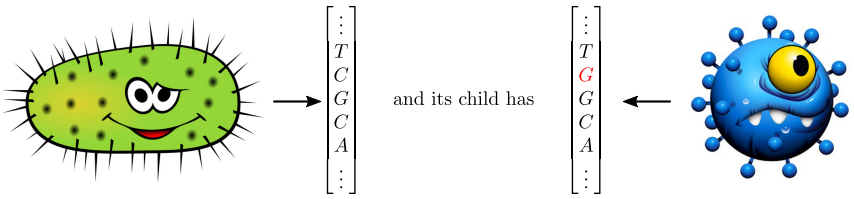
In computer vision we often want to track objects or people. The location of these objects over time is random. For example, look at this frame of a video:

You can predict where some of these people will be in a few seconds (based on their current locations and directions). However, you cannot be certain. What if someone trips? What if the wind moves the leaves of a tree and produces occlusions? All of this can be modeled probabilistically in a precise mathematical way. For instance, we can say that your predictions will be accurate with a probability p close to 1 (if nothing unusual happens), and inaccurate with probability $1 - \mathrm{p}$ (if something unusual happens). Furthermore, it is harder to predict where these people will be later in time. So we can refine our probability model and say your predictions will be accurate with probability p/t (where t is the amount of time), and inaccurate with probability $1 - \mathrm{p}/t$.

You can see that *probability theory is nothing but common sense reduced to calculation* — Pierre Laplace, 1812.

Randomness in Genomics

All organisms have a DNA sequence (genome), i.e., a very long vector of nucleotides (A, C, G, T). Every now and then, organisms *mutate*. That is, when organisms reproduce, some of their nucleotides get changed. For example, it is possible that a parent has the sequence



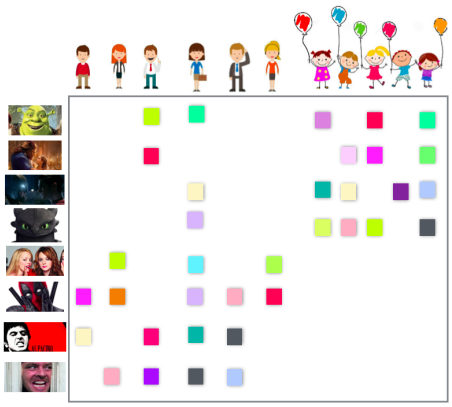
The location of these mutations can be modeled probabilistically. For instance, it is known that certain mutations are approximately distributed uniformly at random within each gene (delimited region of the genome), and that some well-identified genes are more likely to present mutations than others.

Mutations are more common in smaller organisms, like bacteria and viruses, that reproduce very rapidly. That is why pharmaceuticals, insurance companies, the National Health Service (NHS), the National Security Agency (NSA), and even the Department of Defense (DoD), among many others, are very interested in this phenomenon. The reasons are not as apocalyptic as a zombi virus, but close (does this sound familiar/relevant right now in 2020?). For example, some bacteria have become immune to most antibiotics. Similarly, it is

harder to produce vaccines for viruses that mutate quickly. In fact these mutations are the reason why we don't have a definitive flu vaccine yet. What about Corona virus?

Randomness in Recommender Systems

Let's now consider Netflix. Some users have rated some movies, some users have rated others. However, nobody has rated all of them. This produces an *incomplete* data matrix like this (colors indicate how much each person liked a movie)



The goal is to predict which users will like which items, in order to make good recommendations. Again, this can be modeled probabilistically. The movies that each user has rated (and hence the samples in this matrix) are somewhat random. For example, adults are more likely to watch (and enjoy) adult movies, while kids and parents are more likely to watch (and enjoy) children movies. This can be modeled probabilistically.

We could construct similar models for songs, shoes, clothes, restaurants, groceries, etc. So in fact this also applies to Amazon, Pandora, Spotify, Pinterest, Yelp, Apple, etc. If these companies recommend you an item you will like, you are more likely to buy it. You can see why all these companies have a great interest in this problem, and they are paying *a lot* of money to people who work on this.

The Basic Ingredients of Probability

Definition 3.1. There are three elemental concepts in basic probability theory:

$\emptyset$:= Sample space = set of all possible outcomes.

$\mathcal{A}$:= σ-Algebra = Set of all possible events.

$\mathbb{P}$:= Probability measure.

Example 3.1. Consider a fair die. Then

$$\emptyset = \{1, 2, 3, 4, 5, 6\}.$$

$$\mathcal{A} = \Big\{\{1\},\ldots,\{6\},\{1,2\},\ldots,\{5,6\},\ldots,\{1,2,3,4,5,6\}\Big\}$$
$$\mathbb{P}(\mathrm{x}) = 1/6 \text{ for every } \mathrm{x} = 1,\ldots,6.$$

Definition 3.2 (Probability measure)**.** A mapping $\mathbb{P} : \mathcal{A} \to [0,1]$ is a *probability measure* if it satisfies the next properties:

(i) $\mathbb{P}(A) \geq 0$ for every $A \in \mathcal{A}$.

(ii) $\mathbb{P}(\emptyset) = 1$.

(iii) $A \cap B = \emptyset$ for some $A, B \in \mathcal{A}$, then $\mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B)$.

Condition (iii) implies that $\mathbb{P}(A \cup B) \leq \mathbb{P}(A) + \mathbb{P}(B)$, which is often known as the *union bound*.

Conditional Probability

Definition 3.3 (Conditional probability)**.** Let $A, B \in \mathcal{A}$. The *conditional probability* that A occurs given B occurred is

$$\mathbb{P}(A|B) := \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}$$

Example 3.2. Continuing with Example 3.1, let $A = \{1,2\}$, $B = \{2,3\}$. The probability that A occurs is $\mathbb{P}(A) = 1/3$. However, if you already know that B occurred, then the probability that A also occurs increases to

$$\mathbb{P}(A|B) = \frac{1/6}{1/3} = \frac{1}{2}.$$

Independence

Definition 3.4 (Independent events)**.** Let $A, B \in \mathcal{A}$. We say A and B are *independent* if $\mathbb{P}(A|B) = \mathbb{P}(A)$.

In words, two events are independent if they provide no information of one an other.

Example 3.3. Consider two fair dice. Let A be the event that the first die is 1; let B be the event that the second die is 1. Then

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A = 1 \cap B = 1)}{\mathbb{P}(B = 1)} = \frac{1/36}{1/6} = \frac{1}{6} = \mathbb{P}(A).$$

Hence the events A and B are independent. This matches our intuition that one die has no influence on the outcome of the other.

Bayes Rule

Given the conditional probability $\mathbb{P}(A|B)$, Bayes rule gives us a formula for the *inverse* probability, $\mathbb{P}(B|A)$.

Proposition 3.1 (Bayes rule)**.** Let $A, B \in \mathcal{A}$. Then

$$\mathbb{P}(B|A) \;=\; \frac{\mathbb{P}(A|B)\mathbb{P}(B)}{\mathbb{P}(A)} \tag{3.3}$$

Proof. On one hand:

$$\mathbb{P}(A \cap B) \;=\; \mathbb{P}(A|B)\mathbb{P}(B).$$

By symmetry,

$$\mathbb{P}(A \cap B) \;=\; \mathbb{P}(B|A)\mathbb{P}(A).$$

It follows that

$$\mathbb{P}(A|B)\mathbb{P}(B) \;=\; \mathbb{P}(B|A)\mathbb{P}(A),$$

and solving for $\mathbb{P}(B|A)$ we obtain (3.3), as desired. □

Bayes rule plays a crucial role in modern applications.

Example 3.4. Geneticists have determined that 90% of the people with disease B have gene A active, i.e., $\mathbb{P}(A|B) = 0.9$. If you sequence your genome and find out that your gene A is active, what is the probability that you develop disease B? In other words, what is $\mathbb{P}(B|A)$? At first glance you might think it is very likely that you will develop disease B. However, to determine this you need to know $\mathbb{P}(A)$ and $\mathbb{P}(B)$. Of the whole population, if only 5% have disease B, while 45% have gene A active, what is $\mathbb{P}(B|A)$? This is a simple application of Bayes rule:

$$\mathbb{P}(B|A) \;=\; \frac{\mathbb{P}(A|B)\mathbb{P}(B)}{\mathbb{P}(A)} \;=\; \frac{(0.9)(0.05)}{0.45} \;=\; 0.1$$

Random Variables

Sometimes the sample space $\emptyset$ contains elements that are cumbersome to handle. For example, imagine making a list of all animals, $\emptyset = \{\textit{elephant}, \textit{giraffe}, \dots\}$. Other times, $\emptyset$ is not explicitly identified. Hence we want to translate from the world of outcomes to a more familiar and measurable space, like $\mathbb{R}$. That is essentially why we use random variables.

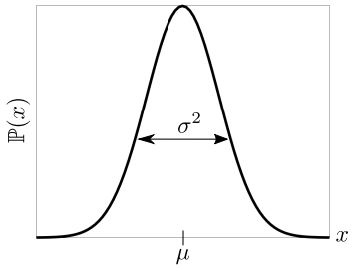


Figure 3.2: Normal density function $\mathbb{P}(\mathrm{x}|\boldsymbol{\theta}) = \mathcal{N}(\mu, \sigma^2)$. In this case, $\boldsymbol{\theta}$ denotes the parameters corresponding to the mean μ and the variance σ^2.

Definition 3.5 (Random variable)**.** A random variable is a mapping $x : \emptyset \to \mathbb{R}$.

For example, we could define a mapping x that maps *elephant* $\mapsto 1$, *giraffe* $\mapsto 2$, etc. Since $\mathbb{P}$ specifies a probability for every $A \in \mathcal{A}$, it also induces a probability in terms of x. For instance, the event $\{x \leq 0\}$ is equivalent to the event $\{\omega \in \emptyset \ : \ x(\omega) \leq 0\}$, and $\mathbb{P}(x \leq 0) = \mathbb{P}(\{\omega \in \emptyset \ : \ x(\omega) \leq 0\})$. Continuing with our example, $P(x \leq 2) = \mathbb{P}(\{\textit{elephant}, \textit{giraffe}\})$.

Densities

Intuitively, a probability measure $\mathbb{P}$ is the rule that assigns probability to the events in $\mathcal{A}$. For instance, in Example 3.1, $\mathbb{P}$ assigns an equal probability of $1/6$ to each of the possible outcomes $\mathrm{x} = 1, \ldots, 6$. To calculate the probability of an event, all we would have to do is compute

$$\mathbb{P}(x \in A) \ = \ \sum_{\mathrm{x} \in A} \mathbb{P}(\mathrm{x}).$$

This was easily done because x was discrete. In this case, $\mathbb{P}$ is called a *mass function*. Notice that x is used for a random variable, whereas x is used for a specific value that x may take. For discrete random variables, $\mathbb{P}(\mathrm{x})$ is essentially shorthand for $\mathbb{P}(x = \mathrm{x})$. If x were continuous, the probability that it takes a particular value is zero. So instead of a mass function, we use a *density function* $\mathbb{P}$ that indicates the probability that x falls within certain intervals. Then

$$\mathbb{P}(x \in A) \ = \ \int_A \mathbb{P}(\mathrm{x}) d\mathrm{x}.$$

In general, probability mass functions and densities depend on certain parameters $\boldsymbol{\theta}$. Whenever this want to be made noticed explicitly, we use the notation $\mathbb{P}(\cdot|\boldsymbol{\theta})$.

Example 3.5. The height of a person can be modeled as a Normal random variable with mean $\mu = 5'5"$ for females and $\mu = 5'10"$ for males (see Figure 3.2). However, the probability that your height is *exactly* $5'5"$ or $5'10"$ is zero. You are more likely to be somewhere in between $(5'3", 5'7")$ or $(5'8", 5'12")$.

Expectation

Definition 3.6 (Expectation)**.** For a continuous random variable x and an arbitrary function $f(x)$,

$$\mathbb{E}[f(x)] \;:=\; \int f(\mathrm{x})\mathbb{P}(\mathrm{x})d\mathrm{x}.$$

If x is a discrete random variable,

$$\mathbb{E}[f(x)] \;:=\; \sum_{\mathrm{x}} f(\mathrm{x})\mathbb{P}(\mathrm{x}).$$

Example 3.6. Special cases of expectations:

- **Probability:** $\mathbb{P}(x \in A) = \mathbb{E}[\mathbb{1}_{x\in A}]$.
- **Mean:** $\mu := \mathbb{E}[x]$.
- **Variance:** $\sigma^2 := \mathbb{E}[(x-\mu)^2]$.

Definition 3.7 (Conditional expectation)**.** The *conditional expectation* of a continuous random variable $f(x)$ given a random variable y is defined as

$$\mathbb{E}[f(x)|y] \;:=\; \int f(\mathrm{x})\mathbb{P}(\mathrm{x}|y)d\mathrm{x},$$

and if x is a discrete random variable,

$$\mathbb{E}[f(x)|y] \;:=\; \sum_{\mathrm{x}} f(\mathrm{x})\mathbb{P}(\mathrm{x}|y).$$

Notice that $\mathbb{E}[f(x)|y]$ is a function of the random variable y, and hence it is also a random variable. Notice the difference between $\mathbb{E}[f(x)|y]$ and $\mathbb{E}[f(x)|\mathrm{y}]$, which is no longer a random variable, because the random variable y is already known to have taken the value y.

Common Probability Measures

The tables below give examples of common probability measures, also known as *distributions*.

Discrete	Parameters ($\boldsymbol{\theta}$)	$\mathbb{P}(x = \mathrm{x})$	$\mathbb{E}[x]$	(x)
Bernoulli	p	$\mathbb{P}(x = 1) = \mathrm{p}$, $\mathbb{P}(x = 0) = 1 - \mathrm{p}$	p	$\mathrm{p}(1 - \mathrm{p})$
Binomial	n, p	$\mathbb{P}(x = \mathrm{x}) = \binom{\mathrm{n}}{\mathrm{x}} \mathrm{p}^{\mathrm{x}}(1 - \mathrm{p})^{\mathrm{n}-\mathrm{x}}$, $\mathrm{x} = 0, \ldots, \mathrm{n}$	np	$\mathrm{np}(1 - \mathrm{p})$
Poisson	λ	$\mathbb{P}(x = \mathrm{x}) = \dfrac{\lambda^{\mathrm{x}} e^{-\lambda}}{\mathrm{x}!}$, $\mathrm{x} = 0, 1, \ldots$	λ	λ

Continuous	Parameters ($\boldsymbol{\theta}$)	$\mathbb{P}(\mathrm{x})$	$\mathbb{E}[x]$	(x)
Uniform	a, b	$\mathbb{P}(\mathrm{x}) = \dfrac{1}{\mathrm{b} - \mathrm{a}}$, $\mathrm{a} \leq \mathrm{x} \leq \mathrm{b}$	$\dfrac{\mathrm{a} + \mathrm{b}}{2}$	$\dfrac{(\mathrm{b} - \mathrm{a})^2}{12}$
Exponential	λ	$\mathbb{P}(\mathrm{x}) = \lambda e^{-\lambda \mathrm{x}}$, $\mathrm{x} \geq 0$	$\dfrac{1}{\lambda}$	$\dfrac{1}{\lambda^2}$
Laplace	λ	$\mathbb{P}(\mathrm{x}) = \dfrac{\lambda}{2} e^{-\lambda \lvert \mathrm{x} \rvert}$	0	$\dfrac{2}{\lambda^2}$
Normal	μ, σ^2	$\mathbb{P}(\mathrm{x}) = \dfrac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{\mathrm{x} - \mu}{\sigma}\right)^2}$	μ	σ^2
Gamma	α, β	$\mathbb{P}(\mathrm{x}) = \dfrac{\beta^\alpha}{\Gamma(\alpha)} \mathrm{x}^{\alpha - 1} e^{-\beta \mathrm{x}}$ $\Gamma(\alpha) := \displaystyle\int_0^\infty \mathrm{x}^{\alpha - 1} e^{-\mathrm{x}} d\mathrm{x}$	$\dfrac{\alpha}{\beta}$	$\dfrac{\alpha}{\beta^2}$
Beta	α, β	$\mathbb{P}(\mathrm{x}) = \dfrac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \mathrm{x}^{\alpha - 1}(1 - \mathrm{x})^{\beta - 1}$	$\dfrac{\alpha}{\alpha + \beta}$	$\dfrac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}$
χ^2	k	$\mathrm{Gamma}(\mathrm{k}/2, 1/2)$		
F	α, β	$\mathbb{P}(\mathrm{x}) = \dfrac{\sqrt{\frac{(\alpha \mathrm{x})^\alpha \beta^\beta}{(\alpha \mathrm{x} + \beta)^{\alpha + \beta}}}}{\mathrm{x}\mathcal{B}(\alpha/2, \beta/2)}$ $\mathcal{B}(\alpha, \beta) := \displaystyle\int_0^1 \mathrm{x}^{\alpha - 1}(1 - \mathrm{x})^{\beta - 1} d\mathrm{x}$	$\dfrac{\beta}{\beta - 2}$, $\beta > 2$	$\dfrac{2\beta^2(\alpha + \beta - 2)}{\alpha(\beta - 2)^2(\beta - 4)}$, $\beta > 4$

Multivariate distributions

In many modern applications it is convenient to arrange random variables in vectors. For example, we might consider a *random vector*

$$\boldsymbol{x} \;=\; [x_1 \;\; x_2 \;\; x_3]$$

containing the information of a person's height, weight and cholesterol level. If the random variables in the vector are independently distributed, then its *joint* distribution is just the product of the univariate distributions of each component. In our example, $\mathbb{P}(\boldsymbol{x}) = \mathbb{P}(x_1, x_2, x_3)$ would simply factor into $\mathbb{P}(x_1)\mathbb{P}(x_2)\mathbb{P}(x_3)$. However, if the random variables in the vector are dependent (as is actually the case with height, weight and cholesterol level), then $\mathbb{P}(\boldsymbol{x})$ does not factor in this simple way.

Multivariate densities model dependent random variables. The one we will use most in this course is the multivariate Normal $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with mean vector $\boldsymbol{\mu} \in \mathbb{R}^{\mathrm{D}}$ and covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{\mathrm{D}\times\mathrm{D}}$, which has the following form:

$$\mathbb{P}(\boldsymbol{x}) \;=\; \frac{1}{\sqrt{(2\pi)^{\mathrm{D}}|\boldsymbol{\Sigma}|}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})^{\mathsf{T}}}.$$

Probability and Likelihood

A probability (mass or distribution) $\mathbb{P}(x|\boldsymbol{\theta})$ determines the frequency with which a random variable x takes each value, given some parameter $\boldsymbol{\theta}$. For example, if $x \sim \text{Bernoulli}(\theta)$, with $\theta = 1/2$, then the probability that x takes the value 1 is $\mathbb{P}(x = 1|\theta) = \theta = 1/2$.

Conversely, the *likelihood* $\mathbb{P}(\mathrm{x}|\boldsymbol{\theta})$ determines the probability that a parameter $\boldsymbol{\theta}$ was the one that generated a sample x. We emphasize this distinction using x instead of x, to indicate that x is already known, i.e., observed data that has already taken a specific value. Under the same Bernoulli example, if we observe $\mathrm{x} = 1$, then the likelihood of the parameter θ is $\mathbb{P}(\mathrm{x} = 1|\theta) \;=\; \theta$.

The probability and the likelihood may *look* a lot alike. The difference is very subtle, and mainly conceptually: the probability $\mathbb{P}(x|\boldsymbol{\theta})$ is a function where x is the variable, and $\boldsymbol{\theta}$ is fixed. In contrast, the likelihood $\mathbb{P}(\mathrm{x}|\boldsymbol{\theta})$ is a function where $\boldsymbol{\theta}$ is the variable, and x is fixed. We use $\mathbb{P}(x|\boldsymbol{\theta})$ when we know $\boldsymbol{\theta}$ and want to guess x; we use $\mathbb{P}(\mathrm{x}|\boldsymbol{\theta})$ when we have already observed data with the specific value x, and we want to guess the parameter $\boldsymbol{\theta}$ that generated it.

Example 3.7. Suppose $x_1, \ldots, x_6$ are *independently and identically distributed* (i.i.d.) according to a Bernoulli($1/4$) distribution. Then the probability that $x_1 = x_2 = x_3 = 1$, and $x_4 = x_5 = x_6 = 0$ is:

$$\begin{aligned}\mathbb{P}(x_1 = x_2 = x_3 = 1, x_4 = x_5 = x_6 = 0|\theta) \;&=\; \prod_{\mathrm{i}=1}^{3} \mathbb{P}(x_{\mathrm{i}} = 1|\theta) \cdot \prod_{\mathrm{i}=4}^{6} \mathbb{P}(x_{\mathrm{i}} = 0|\theta) \\ &=\; \theta^3(1-\theta)^3 \;=\; (1/4)^3 \; (3/4)^3.\end{aligned}$$

Instead, suppose that we observe $\mathrm{x}_1 = \mathrm{x}_2 = \mathrm{x}_3 = 1$, and $\mathrm{x}_4 = \mathrm{x}_5 = \mathrm{x}_6 = 0$. Then the likelihood of θ under this sample is:

$$\begin{aligned}\mathbb{P}(\mathrm{x}_1 = \mathrm{x}_2 = \mathrm{x}_3 = 1, \mathrm{x}_4 = \mathrm{x}_5 = \mathrm{x}_6 = 0|\theta) \;&=\; \prod_{\mathrm{i}=1}^{3} \mathbb{P}(\mathrm{x}_{\mathrm{i}} = 1|\theta) \cdot \prod_{\mathrm{i}=4}^{6} \mathbb{P}(\mathrm{x}_{\mathrm{i}} = 0|\theta) \\ &=\; \theta^3(1-\theta)^3.\end{aligned}$$

Based on this sample, which would be your intuitive best guess at the value of θ? Is this the same value that maximizes the likelihood $\mathbb{P}(\mathrm{x}_1, \ldots, \mathrm{x}_6|\theta)$?

Sums of Independent Random Variables

In many applications we want to know the distribution of the sum of independent random variables. The table below gives a few examples.

Example 3.8. Suppose there is an epidemic in a city with N habitants. The i^{th} person will independently contract the disease with probability p. We can model this as $x_{\mathrm{i}} \overset{iid}{\sim} \text{Bernoulli(p)}$, $\mathrm{i} = 1, \ldots, \mathrm{N}$. Let $m = \sum_{\mathrm{i}=1}^{\mathrm{N}} x_{\mathrm{i}}$ be the number of people that get infected. The Center for Disease Control wants to determine $\mathbb{P}(m > \mathrm{m})$. This raises the question: what is the distribution of m? Notice that m is the sum of i.i.d. Bernoulli random variables. However, m is clearly not Bernoulli. To begin with, m can take values in $\{0, \ldots, \mathrm{N}\}$, while a Bernoulli random variable can only take values in $\{0, 1\}$. So the question is: what is the distribution of a sum of N i.i.d. Bernoulli(p) random variables?

x_{i}	$\sum_{\mathrm{i}=1}^{\mathrm{N}} x_{\mathrm{i}}$
Bernoulli(p)	**Exercise**
Binomial($\mathrm{n_i}$, p)	$\text{Binomial}\left(\sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{n_i}, \mathrm{p}\right)$
Poisson(λ_{i})	$\text{Poisson}\left(\sum_{\mathrm{i}=1}^{\mathrm{N}} \lambda_{\mathrm{i}}\right)$
exp(λ)	Gamma(N, λ)
Gamma($\alpha_{\mathrm{i}}, \beta$)	$\text{Gamma}\left(\sum_{\mathrm{i}=1}^{\mathrm{N}} \alpha_{\mathrm{i}}, \beta\right)$
$\mathcal{N}(\mu_i, \sigma_i^2)$	$\mathcal{N}\left(\sum_{\mathrm{i}=1}^{\mathrm{N}} \mu_{\mathrm{i}}, \sum_{\mathrm{i}=1}^{\mathrm{N}} \sigma_{\mathrm{i}}^2\right)$
$\chi^2(\mathrm{k_i})$	$\chi^2\left(\sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{k_i}\right)$

Other Common Functions of Random Variables

In addition to sums, other common functions of random variables include

- **Linear multiplication:** For a constant matrix $\mathbf{A} \in \mathbb{R}^{\mathrm{D}\times\mathrm{M}}$ and a random vector $\boldsymbol{x} \in \mathbb{R}^{\mathrm{D}}$, $\mathsf{cov}(\boldsymbol{x}\mathbf{A}) = \mathbf{A}^{\mathsf{T}}\mathsf{cov}(\boldsymbol{x})\mathbf{A}$.
- **Squared Normal:** If $x \sim \mathcal{N}(0, 1)$, then $x^2 \sim \chi^2$.

- **Ratio of χ^2's:** If $x \sim \chi^2(\mathrm{k})$ is independent of $y \sim \chi^2(\ell)$, then $\dfrac{\ell x}{\mathrm{k} y} \sim F(\mathrm{k}, \ell)$.