ml_notes.akkefa.com

Release 0.0.1

Ikram Ali

Apr 23, 2024

CONTENTS

1 Contents 3

2 Indices and tables 393

Index 395

Greetings! I am Ikram Ali, and I excel at founding and developing machine learning engineering and data science teams. My current focus areas are **NLP (Natural Language Processing) and MLOps**, where I am passionate about advancing these technologies.

My career strategy involves a deep commitment to mastering the multidisciplinary skills essential for leading data science initiatives. This includes not only Research and Data Engineering but also Machine Learning Engineering and comprehensive Project Management, spanning Agile and Product Management techniques. My broad skill set allows me to efficiently lead cross-functional teams and effectively tackle the challenges of transitioning a model from its initial ideation through to full-scale production.

- https://www.linkedin.com/in/akkefa/
- https://www.github.com/akkefa

I would like to offer concise definitions and comprehensible explanations of Machine Learning and Deep Learning.

CHAPTER

ONE

CONTENTS

1.1 ML Notation / Equations

1.1.1 Notation

Symbol	Formula	Explained
μ	$\sum_x kP(X=x) = \int_{-\infty}^{\infty} xf(x)dx$	
$V(X)$ or σ^2	$E[(X-E[X])^2] = E[(X-\mu)^2] = E[X^2] - E[X]^2$	
σ	$\sqrt{V(X)}$	Standard deviation
$Cov(X,Y)$	Covariance of X and Y	Covariance of X and Y
$\bar{X}$	The sample	The sample mean is an average value
δ	$\delta(v)$	Activation fucntions, sigmoid, relu, etc.

1.1.2 Equations

Cosine Similarity

Cosine similarity is a metric used to measure the similarity between two vectors in a multi-dimensional space. Cosine similarity measures the cosine of the angle between two non-zero vectors in an n-dimensional space.

Formula = dot product / normalized sum of squares

$$\cos(x, y) = \frac{x \cdot y}{\sqrt{x^2} \cdot \sqrt{y^2}} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Properties

- **Scale Invariance** Cosine similarity is scale-invariant, meaning it is not affected by the magnitude of the vectors, only by their orientations.
- One hot and multi hot vectors easily.

```
import torch
from torch.nn import functional as F
```

```
v1 = torch.tensor([0, 0, 1], dtype=torch.float32)
v2 = torch.tensor([0, 1, 1],dtype=torch.float32)

print(F.cosine_similarity(v1, v2 , dim=0))

print(F.normalize(v1, dim=0) @ F.normalize(v2, dim=0))

print(torch.norm(v1) / torch.norm(v2))

print( torch.matmul(v1, v2.T) / ( torch.sqrt( torch.sum(v1 ** 2)) * torch.sqrt( torch.
↪sum(v2 ** 2))) )
```

```
tensor(0.7071)
tensor(0.7071)
tensor(0.7071)
tensor(0.7071)
```

```
/tmp/ipykernel_965/3833807425.py:10: UserWarning: The use of `x.T` on tensors of␣
↪dimension other than 2 to reverse their shape is deprecated and it will throw an error␣
↪in a future release. Consider `x.mT` to transpose batches of matrices or `x.
↪permute(*torch.arange(x.ndim - 1, -1, -1))` to reverse the dimensions of a tensor.␣
↪(Triggered internally at ../aten/src/ATen/native/TensorShape.cpp:3637.)
  print( torch.matmul(v1, v2.T) / ( torch.sqrt( torch.sum(v1 ** 2)) * torch.sqrt( torch.
↪sum(v2 ** 2))) )
```

1.2 What is Probability

1.2.1 Definition

- Probability is the branch of mathematics that deals with the occurrence of a random event.
- Probability is the measure of the likelihood of an event to happen.

Probability is the study of randomness and uncertainty. Probability theory is widely used in the area of studies such as statistics, finance, gambling, artificial intelligence, machine learning, computer science, game theory, and philosophy.

Applications of probability

Some of the applications of probability are predicting results of the following events:

Minor

- that a customer will buy milk if they are also buying bread.
- Of getting at least 2 heads in 5 coin flips.
- Getting 3 and 5 on throwing a die.
- Pulling a green candy from a bag of red candies.
- Winning a lottery 1 in many millions.
- # of customers arriving at a bank in a week

Major

- It is used for risk assessment and modelling in various industries
- Weather forecasting or prediction of weather changes
- Probability of a team winning in a sport based on players and strength of team
- In the share market, chances of getting the hike of share prices

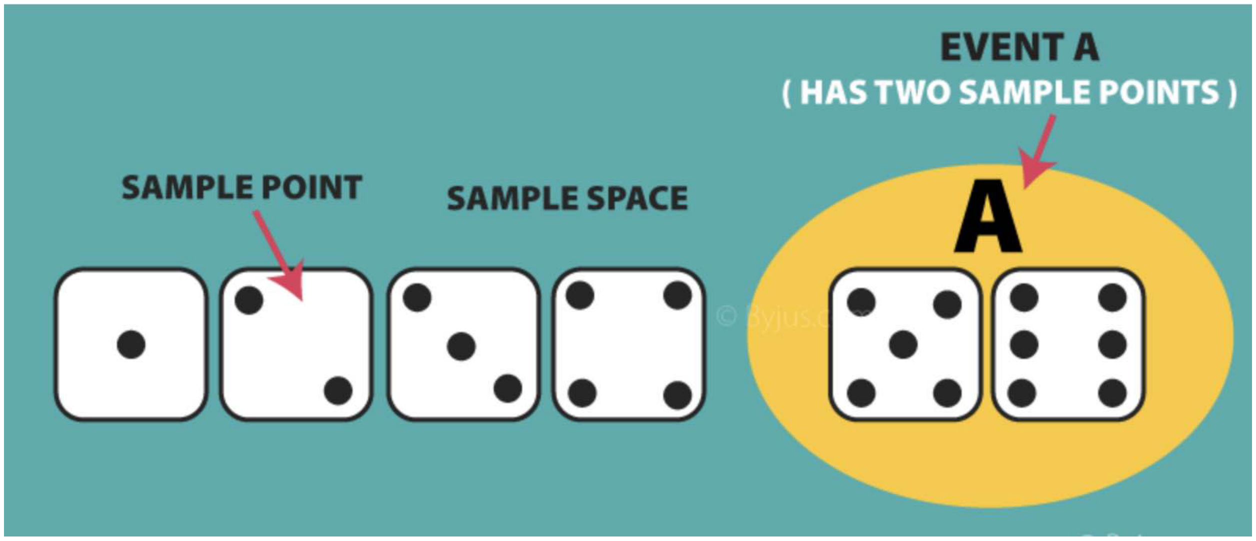
1.2.2 Probability Terminology

The first thing we do when we start thinking about the probability list a number of things that could possibly happen. Some of the important probability terms are discussed here.

Sample Space

Sample space of an experiment, denoted S, is the set of all possible outcomes of an experiment or trial.

- Suppose that we toss a die. Six numbers, from 1 to 6, can appear face up, but we do not yet know which one of them will appear. The sample space is S = {1,2,3,4,5,6}.
- For tossing is a fair coin, the sample space is S = {H, T}



Experiment or Trial

Experiment is any action or process that generates observations or outcomes.\ E.g. The tossing of a coin, selecting a card from a deck of cards, throwing a dice etc.

Outcome or Sample Point

An outcome is a possible result of an experiment or trial.
E.g. The outcome of tossing a coin is a head or a tail.
Roll a die, the outcome is a number between 1 and 6. Each of the six numbers is a sample point

Event

Event is any possible outcome, or combination of outcomes, of an experiment.
E.g. Getting a Head while tossing a coin is an event.

Cardinality

Cardinality of a sample space or an event, is the number of outcomes it contains. $|S|$ represents the cardinality of the sample space.
Tossing a coin, $|S|$ = 2, Rolling a die, $|S|$ = 6
Flip a coin twice, S = {00,01,11,10} $|S|$ = 4
Flip a coin until you get a tail. $S = \{1, 01, 001, 0001, \ldots\}$ $|S| = \infty$

Population

Those individuals or objects from which we want to acquire information or draw a conclusion.
E.g. All valves produced by a specific manufacturing plant.
All adult females in the United States.
All smokers

Sample

Most of the time, the population is so large, we can only collect data on a subset of it. We will call this our sample.

Sets and Subsets

A set is defined as a group of objects (i.e., sets can be made up of letters, numbers, names, etc.).
A subset is defined as a set within a set. set A is a subset of set B if and only if every element of A is also in B.

Empty Set

The set that contains nothing, denoted $\emptyset$.

Complement

A^c = A complement. This is a shorthand way of saying when A does not occur. This set is made up of everything not in A.

Parameter

Parameters are the unknown values of an entire population, such as the mean and standard deviation. Samples can estimate population parameters but their exact values are usually unknowable.

Interview Question

Q: What is the sample space of rolling Two Dice?
Ans: The total number of joint outcomes (a,b) is 6 times 6 which is 36.

Axioms of Probability

Axiom 1
For any event, 'A' the probability of possible outcomes is either 0 or 1, where 0 is the event which never occurs, and 1 is the event will certainly occur. For any event $A, 0 \leq P(A) \leq 1$.

Axiom 2
The sum of probabilities of all possible outcomes is 1.Probability of the sample space S is $P(S) = 1$.

Axiom 3
If A_n mutually exclusive events (intersection of any two is the empty set) then $P\left(\bigcup_{i=1}^{k} A_n\right) = \sum_{k=1}^{n} P\left(A_k\right)$

Axiom 4
The complement of any event A is the event that consists of all the outcomes that are not in A.

Axiom 5
If both A and B are independent, then the conditional probability that event B occurs given that event A has already occurred. P (A and B) = P (A) P (B | A). This is called the General rule of multiplication.

1.2.3 Counting

Despite the trivial name of this topic, be assured that learning to count is not as easy as it sounds.

Naive Probability

The probability of an event occurring, if the likelihood of each outcome is equal, is:

$$P(\text{ Event }) = \frac{\text{number of favorable outcomes}}{\text{number of outcomes}}$$

When we are working with probabilities, our notation will be P(A). this means the **Probability that event A occurred**. So, if A is the event of flipping heads in one flip of a fair coin, then P(A) = .5

This Naive Definition is a reasonable place to start, because it's likely how you have calculated probabilities up to this point. Of course, this is not always the correct approach for real world probabilities (hence the name `naive`).

Multiplication Rule

To understand the Multiplication Rule, visualize a process that has multiple steps, where each step has multiple choices. For example, say that you are ordering a pizza.

1. Size (small, medium, or large)
2. Topping (pepperoni, meatball, sausage, extra cheese)
3. Order Type (delivery or pickup)

Using the multiplication rule, we can easily count the number of distinct pizzas that you could possibly order. Since there are 3 choices for size, 4 choices for toppings, and 2 choices for pickup.

we simply have 3 4 2 = 24 different pizza options.

Now that we have counted the total of number of possible pizzas, it is easy to solve various probability problems.

Interview Question

Q: What are the outcomes of flipping a fair coin and simultaneously rolling a fair die?
Ans: 6 x 2 = 12 outcomes.

Q: How many possible license plates could be stamped if each license plate were required to have exactly 3 letters and 4 numbers?
Ans: 26 x 26 x 26 x 10 x 10 x 10 x 10 = 175,650,000

Factorial

You may have used the factorial for simple arithmetic calculations.

Another use for the factorial function is to count how many ways you can choose things from a collection of things or find how many ways things can be arranged.

Example

Counting the the number of ways to order the letters A, B, and C. We will define a specific arrangement or order as a permutation. You could likely figure this out by just **writing out all of the permutations**:

```
{ABC,ACB,BAC,BCA,CAB,CBA}
```

It's clear that there are 6 permutations. what if you had to do the same for all 26 letters in the alphabet? if you didn't feel like writing out the 26 letters over and over and over, you could use the factorial for a more elegant solution.

the number of permutations when ordering A,B and C is 3!

3 2 1 = 6

Another example, In how many ways can 7 different books be arranged on a shelf?

We could use the Multiplication Principle to solve this problem. We have seven positions that we can fill with seven books. There are 7 possible books for the first position, 6 possible books for the second position, five possible books for the third position, and so on. The Multiplication Principle tells us therefore that the books can be arranged in:

$$7654321 = 5040$$

Alternatively, we can use the simple rule for counting permutations. P = 7! = 5040

Python Solution

```
from math import factorial

print(factorial(3))
print(factorial(6))
```

```
6
720
```

Binomial Coefficient

The binomial coefficient is a mathematical formula that counts the number of ways to choose k items from a collection of n items. This is perhaps the most useful counting tool. which in english is pronounced **n choose x** = $\binom{n}{k}$.

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

With replacement

means the same item can be chosen more than once.

Without replacement

means the same item cannot be selected more than once.

Permutation

Permutation relates to the act of arranging all the members of a set into some sequence or order.

Any ordered sequence of k objects taken from a set of n distinct objects is called a permutation of size k.

When selecting more than one item without replacement and `order does matter`.

Example

```
Suppose an organization has 60 members. One person is selected at random to be the␣
↪president, another
person is selected as the vice-president, and a third is selected as the treasurer.
How many ways can this be done? (This would be the cardinality of the sample space.)
```

$$P_{3,60} = 60.59.58 = \frac{60!}{57!} = 205,320$$

Combination

When selecting more than one item without replacement and `order does not matter`.

Given n distinct objects, any unordered subset of size k of the objects is called a combination.

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

Example

```
Suppose we have 60 people and want to choose a 3 person team (order is not important).␣
↪How many combinations are possible?
```

```
Suppose we have the same 60 people, 35 are female and 25 are male. We need to select a␣
↪committee of 11 people.
How many ways can such a committee be formed?
```

$$C_{60,11} = \frac{60!}{11!(60-11)!} = |S|$$

```
What is the probability that a randomly selected committee will contain at least 5 men␣
↪and at least 5
women? (Assume each committee is equally likely.)
```

P(at least 5M and at least 5W on committee)

$$= P(5m + 6w) + p(6m + 5w)$$

$$= \frac{\begin{pmatrix} 25 \\ 5 \end{pmatrix}\begin{pmatrix} 35 \\ 6 \end{pmatrix}}{\begin{pmatrix} 60 \\ 11 \end{pmatrix}} + \frac{\begin{pmatrix} 25 \\ 6 \end{pmatrix}\begin{pmatrix} 35 \\ 5 \end{pmatrix}}{\begin{pmatrix} 60 \\ 11 \end{pmatrix}}$$

```
What is the probability of drawing the ace of spades twice in a row? (Assume that any␣
↪card drawn on the first draw will
be put back in the deck before the second draw.)
```

$$P(\text{ace of spades}) \times P(\text{ace of spades}) = \left(\frac{1}{52}\right)^2 = \frac{1}{2704} = 0.00037 = 0.037\%$$

```
You draw a card from a deck of cards. After replacing the drawn card back in the deck␣
↪and shuffling thoroughly,
what is the probability of drawing the same card again?
```

$$P(\text{any card}) = \frac{52}{52} = 1$$

$$P(\text{same card as first draw}) = \frac{1}{52} \approx 0.019$$

$$P(\text{any card})P(\text{same card as first draw}) = (1)(\frac{1}{52}) = \frac{1}{52} \approx 0.019$$

```
Use $n \choose k$ to calculate the probability of throwing three heads in five coin␣
↪tosses.
```

$$\binom{n}{k} = \binom{5}{3} = \frac{5!}{3!(5-3)!} = \frac{5!}{(3!)(2!)} = \frac{5 \times 4 \times 3 \times 2 \times 1}{(3 \times 2 \times 1)(2 \times 1)} = \frac{120}{(6)(2)} = \frac{120}{12} = 10$$

```
Twelve (12) patients are available for use in a research study. Only seven (7) should be␣
↪assigned to receive the study
treatment. How many different subsets of seven patients can be selected?
```

$$\binom{n}{k} = \binom{12}{7} = \frac{12!}{7!(12-7)!} = 792$$

Torch combinations

```
import torch

a = torch.tensor([1, 2, 3])
print(torch.combinations(a))
print(torch.combinations(a, r=3))
torch.combinations(a, with_replacement=True)
```

```
tensor([[1, 2],
        [1, 3],
        [2, 3]])
tensor([[1, 2, 3]])
```

```
tensor([[1, 1],
        [1, 2],
        [1, 3],
        [2, 2],
        [2, 3],
        [3, 3]])
```

Difference Between Permutation and Combination

Permutation	Combination
Order matters	Order doesn't matter
Number of ways to arrange the elements of a set.	Number of ways to choose k elements from a set of n elements.
Arranging people, digits, numbers, alphabets, letters, and colours.	Selection of menu, food, clothes, subjects, the team.
Picking a President, VP and Waterboy from a group of 10.	Picking a team of 3 people from a group of 10.
Listing your 3 favorite desserts, in order, from a menu of 10. P(10,3) = 720.	Choosing 3 desserts from a menu of 10. C(10,3) = 120.

Sampling Table

	Order Matters	Order Doesn't Matter
With Replacement	n^k	$\binom{n+k-1}{k}$
Without Replacement	$\frac{n!}{k!(n-k)!}$	$\binom{n}{k}$

Interview Questions

- There are 25 students in a class. Find the number of ways in which a committee of 3 students is to be formed?
  25 choose 3 2300
- In a meeting between two countries, each country has 12 delegates. All the delegates of one country shake hands with all delegates of the other country. Find the number of handshakes possible?
  Total number of handshakes = 12 x 12 = 144
- How many groups of 6 persons can be formed from 8 men and 7 women?
  Total number of person = 8 men + 7 women = 15
  15 choose 6 = 5005

1.3 Bayes Theorem

1.3.1 Definition

Bayes theorem is also known as the formula for the **probability of causes**.

Theorem states that the conditional probability of an event, based on the occurrence of another event, is equal to the likelihood of the second event given the first event multiplied by the probability of the first event.

1.3.2 Conditional Probability

Two events A and B from the `same sample space S`. Calculate the probability of event A knowing that event B has occurred. B is the "conditioning event". $P(A|B)$

Conditional Probability is $P(A \mid B)=\frac{P(A \cap B)}{P(B)}, \quad P(B)>0$

1.3.3 Multiplication Rule

This leads to the multiplication rule $P(A \cap B)=P(B) P(A \mid B)=P(A) P(B \mid A)$

1.3.4 Bayes Theorem

Bayes Theorem $P(A \mid B)=\frac{P(B \mid A) P(A)}{P(B)}$

Example

Bayes Theorem can detect spam e-mails.

- Assume that the word **offer** occurs in 80% of the spam messages.
- Also assume **offer** occurs in 10% of my desired e-mails.

Question

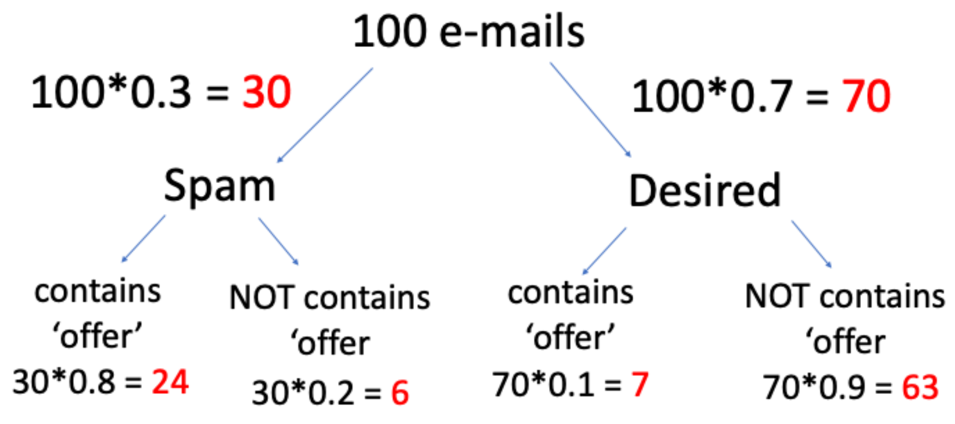
If 30% of the received e-mails are considered as a scam, and I will receive a new message which contains 'offer', what is the probability that it is spam?

I received 100 e-mails.

- I have 30 spam e-mails
- 70 desired e-mails.

The percentage of the word 'offer' that occurs in spam e-mails is 80%. It means 80% of 30 e-mail and it makes 24. Now, I know that 30 e-mails of 100 are spam and 24 of them contain 'offer' where 6 of them not contains 'offer'.

The percentage of the word 'offer' that occurs in the desired e-mails is 10%. It means 7 of them (10% of 70 desired e-mails) contain the word 'offer' and 63 of them not.


100 e-mails
100*0.3 = 30
100*0.7 = 70
Spam
Desired
contains 'offer' 30*0.8 = 24
NOT contains 'offer 30*0.2 = 6
contains 'offer' 70*0.1 = 7
NOT contains 'offer 70*0.9 = 63

The question was what is the probability of spam where the mail contains the word 'offer':

1. We need to find the total number of mails which contains 'offer' ; 24 +7 = 31 mail contain the word 'offer'
2. Find the probability of spam if the mail contains 'offer' ;

In 31 mails 24 contains 'offer' means 77.4% = 0.774 (probability)

NOTE: In this example, I choose the percentages which give integers after calculation. As a general approach, you can think that we have 100 units at the beginning so if the results are not an integer, it will not create a problem. Such that, we cannot say 15.3 e-mails but we can say 15.3 units.

Solution with Bayes' Equation:

A = Spam

B = Contains the word 'offer'

$$P(\text{ spam } \mid \text{ contains offer }) = \frac{P(\text{ contains offer } \mid \text{ spam }) * P(\text{ spam })}{P(\text{ contains offer })}$$

P(contains offer|spam) = 0.8 (given in the question)

P(spam) = 0.3 (given in the question)

Now we will find the probability of e-mail with the word 'offer'. We can compute that by adding 'offer' in spam and desired e-mails. Such that;

P(contains offer) = 0.3*0.8 + 0.7*0.1* = 0.31

$$P(\text{ spam } \mid \text{ contains offer }) = \frac{0.8 * 0.3}{0.31} = 0.774$$

As it is seen in both ways the results are the same. In the first part, I solved the same question with a simple chart and for the second part, I solved the same question with Bayes' theorem.

1.3.5 Law of Total Probability

$B=(B\cap A)\cup(B\cap A^c)$

$P(B)=P(B\cap A)+P\left(B\cap A^c\right)=P(B\mid A)P(A)+P\left(B\mid A^c\right)P\left(A^c\right)$

1.3.6 Independence and Mutually Exclusive Events

Two events are `independent` if knowing the outcome of one event does not change the probability of the other.

- Flip a two-sided coin repeatedly. Knowing the outcome of one flip does not change the probability of the next.

Two events, A and B, are independent if $P(A|B)=P(A)$, or equivalently $P(B|A)=P(B)$.

`Recall:` $P(A\mid B)=\frac{P(A\cap B)}{P(B)}$

then, if A and B are independent, we get the multiplication rule for independent events:

$P(A\cap B)=P(A)P(B)$

Example

Suppose your company has developed a new test for a disease. Let event A be the event that a randomly selected individual has the disease and, from other data, you know that 1 in 1000 people has the disease. Thus, P(A) = .001. Let B be the event that a positive test result is received for the randomly selected individual. Your company collects data on their new test and finds the following:

- $P(B|A)$ = .99
- $P(B^c|A)$ = .01
- $P(B|A^c)$ = .02

Calculate the probability that the person has the disease, given a positive test result. That is,

find $P(A|B)$.

$$\begin{aligned}
P(A\mid B)&=\frac{P(A\cap B)}{P(B)}\\
&=\frac{P(B\mid A)P(A)}{P(B)}\leftarrow \text{ Bayes theorem}\\
&=\frac{P(B\mid A)P(A)}{P(B\mid A)P(A)+P\left(B\mid A^c\right)P\left(A^c\right)}\leftarrow \text{ Law of Total Probability}\\
&=\frac{(.99)(.001)}{(.99)(.001)+(.02)(.999)}\\
&=.0472
\end{aligned}$$

$$\begin{aligned}
P(A)=.001\leftarrow &\text{ prior prob of } A\\
P(A\mid B)=.0472\leftarrow &\text{ posterior prob of } A
\end{aligned}$$

1.4 Random Variables

The first step to understand random variable is to do a fun experiment. Go outside in front of your house with a pen and paper. Take note of every person you pass and their hair color & height in centimeters. Spend about 10 minutes doing this.

Congratulations! You have conducted your first experiment! Now you will be able to answer some questions such as:

- How many people walked past you?
- Did many people who walked past you have blue hair?
- How tall were the people who walked past you on average?

You pass 10 people in this experiment, 3 of whom have blue hair, and their average height may be 165.32 cm. In each of these questions, there was a number; a measurable quantity was attached.

1.4.1 Definition

A random variable rv is a real-valued function, whose domain is the entire *sample space* of an experiment. Think of the domain as the set of all possible values that can go into a function. A function takes the domain/input, processes it, and renders an output/range. This set of real values obtained from the random variable is called its `range`.

A random variable (rv) is a function that maps events (from the sample space S) to the real numbers. It's a function which performs the mapping of the outcomes of a random process to a numeric value.

The domain of a random variable is a sample space, which is represented as the collection of possible outcomes of a random event. For instance, when a coin is tossed, only two possible outcomes are acknowledged such as heads or tails.

Denoted by

Random variables Denote by a capital letters near the end of the alphabet `(e.g. X, Y ).`

Note: Why is it called a random variable?
Because we think of it as a variable that take random value intuitively. Formally they are function.

1.4.2 Probability Distribution

A Probability Distribution is a graph, table, or function that gives the probability for each value of the random variable.

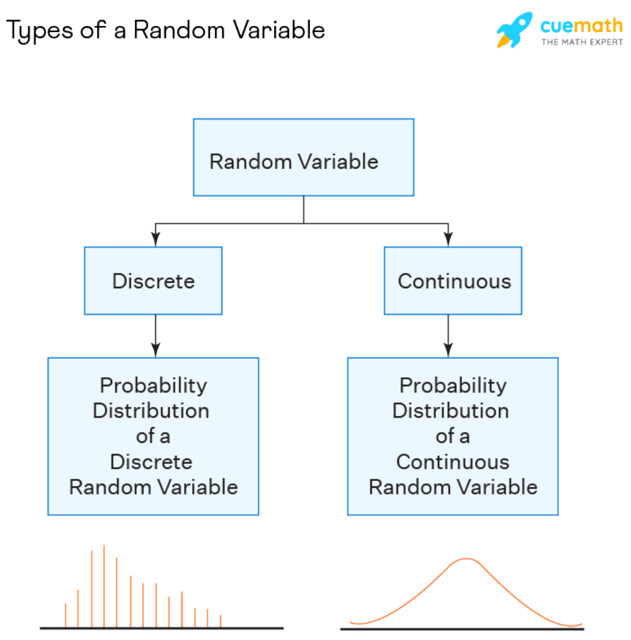
Requirments

1. The sum of the probabilities is 1. $\sum f(x) = 1$.
2. Every probability p_i is a number between 0 and 1. $0 \leq f(x) \leq 1$

Difference between random variables and probability distributions

A random variable is a numerical description of the outcome of a statistical experiment. The probability distribution for a random variable describes how the probabilities are distributed over the values of the random variable.

1.4.3 Types of Random Variables


Types of a Random Variable
cuemath
THE MATH EXPERT
Random Variable
Discrete
Continuous
Probability Distribution of a Discrete Random Variable
Probability Distribution of a Continuous Random Variable

Discrete random variable
A discrete random variable is a type of random variable that has a countable number of distinct values that can be assigned to it, such as in a coin toss.

Continuous random variable
A continuous random variable stands for any amount within a specific range or set of points and can reflect an infinite number of potential values, such as the average rainfall in a region.

Big Picture In statistics, we will model populations using random variables (e.g. mean, variance) of these random variables will tell us about the population we are studying.

1.4.4 Probability mass function (P.M.F)

The probability that a discrete random variable X takes on a particular value x that is $P(X = x)$ is denoted by

$$\text{p.m.f} = f(x) = f_x(x) = f_y(y)$$

Properties

The probability mass function, $P(X = x) = f(x)$, of a discrete random variable X is a function that satisfies the following properties:

1. All of the probabilities must be positive. $P(X = x) = f(x) > 0$, if $x \in$ the support S
2. Sum of all probabilities of same sample space equals to 1. $\sum_{x \in S} f(x) = 1$
3. $P(X \in A) = \sum_{x \in A} f(x)$

$$\text{Random variable} = X = \begin{cases} 1, & \text{if "Heads"} \\ 0, & \text{if "Tails"} \end{cases} = \begin{cases} P(X=1), & \text{if "Heads"} \\ P(X=0), & \text{if "Tails"} \end{cases}$$

$$PMF = f(x) = f_x(x) = P(X = x) = \begin{cases} 1/2, & \text{if } x = 0 \\ 1/2, & \text{if } x = 1 \\ 0, & \text{otherwise} \end{cases}$$

$$p(x) = P(X = x) = P(\text{ all } x \in S \mid X(s) = x)$$

Interview Question

Q: Let $f(x) = cx^2$ for $x = 1, 2, 3$. Determine the constant c so that the function $f(x)$ satisfies the conditions of being a probability mass function?

Answer: Using property no 2

$$\begin{aligned} \sum_{x=1}^{3} f(x) &= \sum_{x=1}^{3} cx^2 = c\sum_{x=1}^{3} x^2 \\ &= c\left[1^2 + 2^2 + 3^2\right] = c[1 + 4 + 9] \\ &= c(14) \overset{\text{set}}{=} 1 = c = 1/14 \\ f(x) &= \frac{1}{14}x^2 \text{ for } x = 1, 2, 3 \end{aligned}$$

1.4.5 Cumulative distribution function (CDF)

The cumulative distribution function (CDF or cdf) of the random variable X has the following definition:

$$F_X(t) = P(X \leq t) = \sum_{x \leq y} P(X = t) = \int_{-\infty}^{t} f(t)dt$$

Properties

The cdf of random variable X has the following properties:

1. The cdf, $F_X(t)$, ranges from 0 to 1 . This makes sense since $F_X(t)$ is a probability.
2. If X is a discrete random variable whose minimum value is a, then $F_X(a) = P(X \leq a) = P(X = a) = f_X(a)$. If c is less than a, then $F_X(c) = 0$.
3. If the maximum value of X is b, then $F_X(b) = 1$.
4. Also called the distribution function.

Example

Suppose X is a discrete random variable. Let the pmf of X be equal to

$$f(x) = \frac{5-x}{10}, \quad x = 1, 2, 3, 4$$

Suppose we want to find the cdf of X. The cdf is $F_X(t) = P(X \leq t)$.

- For $t = 1, P(X \leq 1) = P(X = 1) = f(1) = \frac{5-1}{10} = \frac{4}{10}$.
- For $t = 2, P(X \leq 2) = P(X = 1 \text{ or } X = 2) = P(X = 1) + P(X = 2) = \frac{5-1}{10} + \frac{5-2}{10} = \frac{4+3}{10} = \frac{7}{10}$
- For $t = 3, P(X \leq 3) = \frac{5-1}{10} + \frac{5-2}{10} + \frac{5-3}{10} = \frac{4+3+1}{10} = \frac{9}{10}$.
- For $t = 4, P(X \leq 4) = \frac{5-1}{10} + \frac{5-2}{10} + \frac{5-3}{10} + \frac{5-4}{10} = \frac{10}{10} = 1$.

1.4.6 Probability density function (PDF)

X = f(x) is the probability density function of the continues random variable X.

$$P(a \leq X \leq b) = \int_a^b f(x)dx$$

f(x) = Curve under which area represent the probability $P(a \leq X \leq b) = \int_a^b f(x)dx$

1.4.7 Expected Value (Mean or Average)

The concept was first devised in the 17th century to analyze gambling games and answer questions such as:

- How much do I gain - or lose - on average, if I repeatedly play a given gambling game?
- How much can I expect to gain - or lose - by making a certain bet?

For example, if you play a game where you gain 2$ with probability 1/2 and you lose 1$ with probability 1/2, then the expected value of the game is half a dollar

$$2 \times \tfrac{1}{2} + (-1) \times \tfrac{1}{2} = \tfrac{1}{2} = 0.5$$

it means that if you play this game many times, and the number of times each of the two possible outcomes occurs is proportional to its probability, then on average you gain 1/2$ each time you play the game.

Definition

The expected value or mean of a random variable is a weighted average of all possible outcomes. In the case of a continuum of possible outcomes, the expectation is defined by integration.

Denoted by μ_x or $E(X)$.

$$\mu = \mu_x = E(X) = \sum_x kP(X = x) = \int_{-\infty}^{\infty} xf(x)dx$$

Example

5 exams result : 70 +80 + 80 + 90 + 90

$Avg = \frac{70+80+80+90+90}{5} = \frac{1}{5}(70) + \frac{2}{5}(80) + \frac{2}{5}(90) = 82.5$

Let X represent the outcome of a roll of a fair six-sided die. The possible values for X are 1, 2, 3, 4, 5, and 6, all of which are equally likely with a probability of $1/6$ The Expected Value of X is

$E[X] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = (1+2+3+4+5+6)/6 = 3.5$

---

x	1	2	3
P(X=x)	1/4	1/4	1/2

$E[X] = (1)(1/4) + (2)(1/4) + (3)(1/2) = 9/4 = 2.25 = \sum_x xP(X = x)$

Imagine a game in which, on any play, a player has a 20% chance of winning $3 and an 80 probability mass function of the random variable, the amount won or lost on a single play is$:so the average amount won (actually lost, since it is negative)

$$E(X) = (\$3)(0.2) + (-\$1)(0.8) = \$ - 0.20$$

In the long run you guaranteed to lose no more than 20 cents.

Pytorch implementation

```python
import torch

# Create a tensor
T = torch.Tensor([2.453, 4.432, 0.754, -6.554])
print("T:", T)

# Compute the mean and standard deviation
mean = torch.mean(T)
print("mean:", mean)
```

```
T: tensor([ 2.4530,  4.4320,  0.7540, -6.5540])
mean: tensor(0.2713)
```

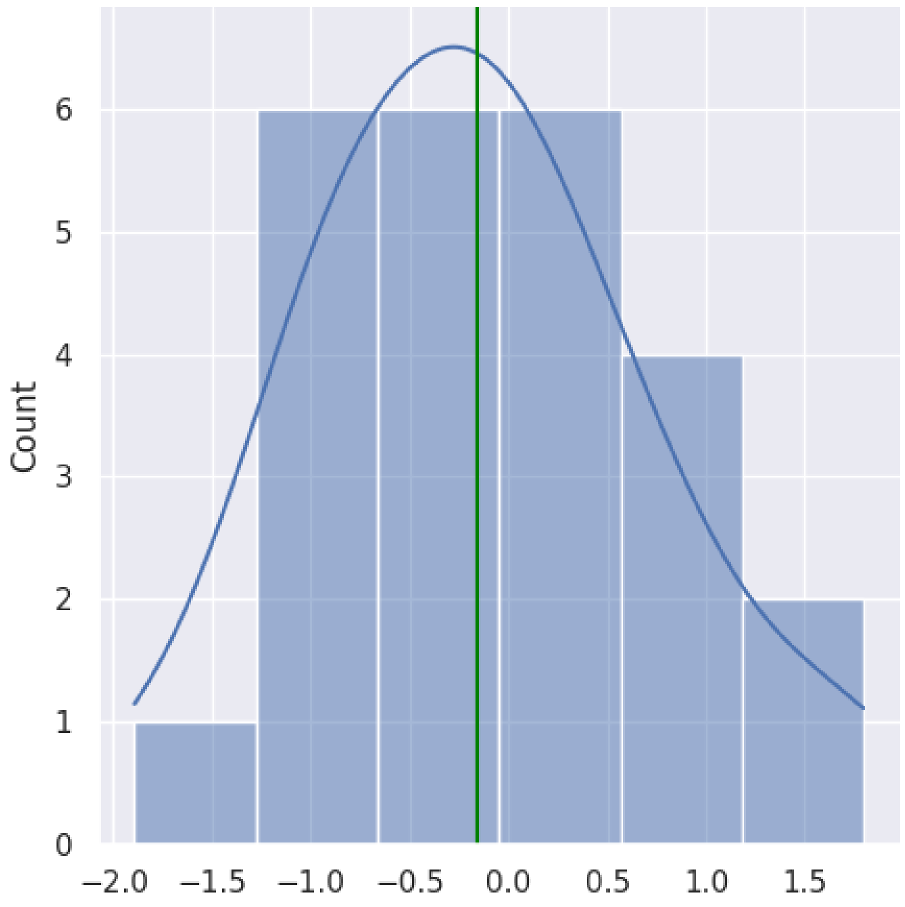
```python
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_theme(style="darkgrid")
data = torch.randn(25)

print(data)

```

(continues on next page)

(continued from previous page)

```
print("Mean :", torch.mean(data))

_ =sns.displot(data,kde=True, )
plt.axvline(torch.mean(data), color='green')
plt.show()
```

```
tensor([ 0.1304, -0.2829,  0.7720,  0.6261, -0.7537, -0.6111,  0.0727,  0.8777,
        -0.0064,  0.5796, -1.1902, -1.0818, -0.8887, -0.9166, -0.3301,  1.7974,
        -0.0081,  0.1470, -1.2143, -1.8930, -0.6445,  1.4764, -0.4136,  0.1353,
        -0.3242])
Mean : tensor(-0.1578)
```



Properties

Expectation is a linear operator, which means for our purposes it has a couple of nice properties.

Expected value of a constant

A perhaps obvious property is that the expected value of a constant is equal to the constant itself.

$$E[c] = c$$

Scalar multiplication of a random variable

If X is a random variable and a is a constant, then

$$E[aX] = aE[X]$$

Expectation of a product of random variables

Let X and Y be two random variables. In general, there is no easy rule or formula for computing the expected value of their product. However, if X and Y are statistically independent, then

$$E[XY] = E[X]E[Y]$$

Expectation of a sum of random variables

$$E(X + Y) = E(X) + E(Y)$$

If random variables is function

$$E(g(X)) = \begin{cases} \sum_k g(k)P(X = k), X \text{ is discrete} \\ \int_{-\infty}^{\infty} g(x)f(x)dx, X \text{ is continuous.} \end{cases}$$

$E(aX + b) = \sum_k (aX + b)P(X = k)$

$E(aX + b) = a\sum_k kP(X = k) + b\sum_k P(X = k)$

$E(aX + b) = aE(x) + b * 1 = aE(x) + b$

Law of the Unconscious Statistician

IF X with pdf $f_x(x)$ and g is a function `Find [()]`

Let Y=g(X). The pdf for Y is:

$f_Y(y) = f_X\left(g^{-1}(y)\right) \cdot \left|\frac{d}{dy} g^{-1}(y)\right| =$ So, $E[g(X)] = E[Y] = \int_{-\infty}^{\infty} y \cdot f_Y(y) dy$

$= \int_{-\infty}^{\infty} y \cdot f_x\left(g^{-1}(y)\right) \cdot \left|\frac{d}{dy} g^{-1}(y)\right| dy$

Let $x = g^{-1}(y)$. Then $dx = \frac{d}{dy} g^{-1}(y) dy$

$E[g(X)] = \int_{-\infty}^{\infty} g(x) f_X(x)) dx$

1.4.8 Variance

- Measures how far we expect our random variable to be from the mean.
- Measures of **spread** of a distribution.
- Variance is a measure of dispersion.

Denoted by

$$\sigma^2 \text{ or } V(X).$$

$$V(X) = E[(X - E[X])^2] = E[(X - \mu)^2] = E[X^2] - E[X]^2$$

To better understand the definition of variance, we can break up its calculation in several steps:

1. Compute the expected value of X, denoted by $\mathrm{E}[X]$.
2. Construct a new random variable $Y = X - \mathrm{E}[X]$ equal to the deviation of X from its expected value.
3. Take the square $Y^2 = (X - \mathrm{E}[X])^2$ which is a measure of distance of X from its expected value (the further X is from $\mathrm{E}[X]$, the larger Y^2)
4. Finally, compute the expectation of Y^2 to know the average distance:

$$\mathrm{E}\left[Y^2\right] = \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] = \mathrm{Var}[X]$$

From these steps we can easily see that

- variance is always positive because it is the expected value of a squared number.
- the variance of a constant variable X (i.e., a variable that always takes on the same value) is zero; in this case, we have that $X = \mathrm{E}[X], Y^2 = 0$ and $\mathrm{E}\left[Y^2\right] = 0$
- the larger the distance Y^2 is on average, the higher the variance.

For continuous rv

If X is a continuous random variable, the variance is defined by the integral of the probability density function. $V(X) = \int_{-\infty}^{\infty} (x-\mu_x)^2 f(x)dx$

$V(X) = \int_{-\infty}^{\infty} (x-\mu_x)^2 f(x)dx$

$= \int_{-\infty}^{\infty} \left(x^2 - 2\mu_x x + \mu_x^2\right) f(x)dx$

$= \int_{-\infty}^{\infty} x^2 f(x)dx - 2\mu_x \int_{-\infty}^{\infty} x f(x)dx + \mu_x^2 \int_{-\infty}^{\infty} f(x)dx$

$V(X) = E(X^2) - E(X)^2$

Properties

Addition to a constant

Let $a \in \mathbb{R}$ be a constant and let X be a random variable.

$$Var[a+X] = Var[X]$$

Thanks to the fact that $\mathrm{E}[a+X] = a + \mathrm{E}[X]$ (by linearity of the expected value), we have

$$\begin{aligned}\mathrm{Var}[a+X] &= \mathrm{E}\left[(a+X-\mathrm{E}[a+X])^2\right] \\ &= \mathrm{E}\left[(a+X-a-\mathrm{E}[X])^2\right] \\ &= \mathrm{E}\left[(X-\mathrm{E}[X])^2\right] \\ &= \mathrm{Var}[X]\end{aligned}$$

Multiplication by a constant

Let $a \in \mathbb{R}$ be a constant and let x be a random variable.

$$Var[aX] = a^2 Var[X]$$

Thanks to the fact that $E[aX] = aE[X]$ (by linearity of the expected value), we obtain

$$\begin{aligned}Var[aX] &= E\left[(aX - E[aX])^2\right] \\ &= E\left[(aX - aE[X])^2\right] \\ &= \mathrm{E}\left[a^2(X-\mathrm{E}[X])^2\right] \\ &= a^2\mathrm{E}\left[(X-\mathrm{E}[X])^2\right] \\ &= a^2\,\mathrm{Var}[X]\end{aligned}$$

`Find Var[aX] = ?`

Let Y = aX. Then, $\mu_y = E[Y] = E[aX] = E[a\mu_x] = aE[\mu_x] = aE[X]$

==> $Var[aX] = Var[Y] = Var[(Y-\mu_y)^2] = a^2 Var[(X-\mu_x)^2] = a^2 V(X)$

`For Function`

$V(g(X)) = \begin{cases} \sum_k (g(k) - E(g(X)))^2 P(X=k), & X \text{ discrete} \\ \int_{-\infty}^{\infty} (g(x) - E(g(X)))^2 f(x)dx, & X \text{ continuc} \end{cases}$

Find V(a X+b)

$V(aX+b) = E[(aX+b-E(aX+b))^2]$

$= E[(ax+\cancel{b} - aE(x) - \cancel{b})^2]$

$= E[(a^2(x-E(x))^2]$

$= a^2E[(x-E(x)^2] = a^2V(x)$

Variance measure the spread the data B shift the data but doest not affect the spread.

Find Var[aX]

Let Y=aX. Then

$$\begin{aligned}\mu_Y = E[Y] = E[aX] = aE[X] = a\mu_X\\ Var[aX] = Var[Y] = E\left[(Y-\mu_Y)^2\right]\\ = a^2E\left[(X-\mu_X)^2\right]\\ = a^2Var[X]\end{aligned}$$

Find Var[X + Y]

$$Var[X+Y] = Var[X] + Var[Y]$$

- We will see that this is true if X and Y are independent.
- Need concept of "covariance".

1.4.9 Standard Deviation

The standard deviation is the square root of the variance. $\sigma_x = \sqrt{V(X)}$

1.4.10 Indicator function

The indicator function of an event is a random variable that takes

- value 1 when the event happens;
- value 0 when the event does not happen.

Let A = Set of real numbers

$$I_A(x) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases}$$

Other definition

The indicator function of a subset A of a set X is a function.

Indicator function$_A(X) = \mathbf{1}_A(x) = \begin{cases} 1, & \text{if } A \cap X \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$

Notation= $\nVdash_A(x)$

1.4.11 Random Sample

A collection of random variables is independent and identically distributed if each random variable has the same probability distribution as the others and all are mutually independent.

$$\text{Random Sample} = \text{X}_1, X_2, X_3, ..., X_n$$

Suppose that $X_1, X_2, X_3, ..., X_n$ is a random sample from the Normal distribution with parameters μ and $sigma^2$. **Mu and sigma are same for all random variables**

$$X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} N(\mu, \sigma^2)$$

Suppose that $X_1, X_2, X_3, ..., X_n$ is a random sample from the gamma distribution with parameters $alpha$ and β.

$$X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \Gamma(\alpha, \beta)$$

Example

A good example is a succession of throws of a fair coin: The coin has no memory, so all the throws are **independent**. And every throw is 50:50 (heads:tails), so the coin is and stays fair - the distribution from which every throw is drawn, so to speak, is and stays the same: **identically distributed**.

Independent and identically distributed random variables (IID)

$$\text{Random Sample == IID}$$

1.5 Discrete Distributions

A discrete distribution is a distribution of data in statistics that has discrete values. Discrete values are countable, finite, non-negative integers, such as 1, 10, 15, etc.

1.5.1 Bernoulli Distribution

The Bernoulli distribution is a univariate discrete distribution used to model random experiments that have binary outcomes.

Bernoulli Random Variable

A Bernoulli RV $X \sim Bern(p)$ is a random variable that is either 0 or 1 with probability p or $1-p$ respectively. Suppose that you perform an experiment with two possible outcomes: either success or failure.

Let X be a discrete random variable. $x \in 0, 1$

$$f_x(x) = P(X = x) = \begin{cases} 1-p, & \text{if x = 0} \\ p, & \text{if x = 1} \\ 0, & \text{otherwise} \end{cases}$$

P.M.F

$$P(X = 1) = p \tag{1.1}$$
$$P(X = 0) = 1 - p \tag{1.2}$$

Using the indicator function notation

$$I_A(x) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases}$$

$$P(X = x) = p^x (1 - p)^{1-x} \cdot I_{\{0,1\}}(x)$$

Mean (Expected Value)

The expected value of a Bernoulli random variable X is

$$E[X] = p$$

Proof

$$\tag{1.3}$$

$$E[X] = \sum_x kP(X = k) \tag{1.4}$$
$$= 0 * P(x = 0) + 1 * P(x = 1) \tag{1.5}$$
$$= 0 * (1 - p) + 1 * p \tag{1.6}$$
$$= p \tag{1.7}$$

Variance

The variance of a Bernoulli random variable X is

$$Var[X] = p(1 - p)$$

Proof

$E(X^2) = \sum_k k^2 P(X = k) = 1^2 * p = p$

$$\begin{aligned} V(X) &= E[X^2] - E[X]^2 \\ &= p - p^2 \\ &= p(1 - p) \end{aligned}$$

1.5.2 Geometric Distribution

The geometric distribution is a discrete probability distribution that calculates the probability of the first success occurring during a specific trial.

The geometric distribution is the probability distribution of the number of failures we get by repeating a Bernoulli experiment until we obtain the first success.

Geometric Random Variable

Definition 1

A geometric rv $X \sim Geom(p)$ consists of

- independent Bernoulli trials,
- each with the same probability of success p or Failure (1-p),
- repeated until the first success is obtained.

Definition 2

The geometric rv is the distribution of the number of trials needed to get the first success in repeated independent Bernoulli trials.

Example If we toss a coin until we obtain head, the number of tails before the first head has a geometric distribution.

Attention: It can also define as number of failures before first success.

Parameter

The geometric distribution has one parameter, p = the probability of success for each trial. You denote the distribution as G(p), which indicates a geometric distribution with a success probability of p.

Uses

- Six in a series of die rolls?
- Person to support a law during a repeated sampling for an interview?
- Product to have a defect in a random sample from an assembly line?
- Successful attempt for a project or task?

Properties

1. Each trial is identical, and can result in a success or failure.
2. The probability of success, p, is constant from one trial to the next.
3. The trials are independent, so the outcome on any particular trial does not influence the outcome of any other trial.
4. Trials are repeated until the first success.

P.M.F

The *sample space* of geometric random variable is

$$S = \{1, 01, 001, 0001, 00001, 000001, \dots\}$$

Bernoulli trail success = 1 = P
Bernoulli trail failure = 0 = 1 - P

$P(X = 1) = p$
$P(X = 2) = (1 - p)p$
$P(X = 3) = (1 - p)(1 - p)p$
$P(X = 4) = (1 - p)(1 - p)(1 - p)p$ = failure, failure, failure, Success
$P(X = 5) = (1 - p)^4 p$
$P(X = x) = (1 - p)^{x-1} p$

$$f(x) = P(X = x) = (1 - p)^{x-1} p \quad for \;\; x = 1, 2, 3, 4, 5, \dots$$
$$f(x) = P(X = x) = (1 - p)^{x-1} \cdot p \cdot I_{\{1,2,3,\dots\}}(x)$$

Mean (Expected Value)

The expected value of a geometric random variable X is

$$\begin{aligned} E[X] &= \sum_{k=1}^{\infty} kP(X = k) \\ &= \sum_{k=1}^{\infty} k(1 - p)^{k-1} p \\ &= \frac{1}{p} \end{aligned}$$

Variance

The expected value of a geometric random variable X is

$$\begin{aligned} V(X) &= E[X^2] - E[X]^2 \\ &= \frac{1 - p}{p^2} \end{aligned}$$

Interview Question

Q: On each day we play a lottery in which the probability of winning is $1 What is the expected value of the number of days that will elapse before we win for the first time?$

Answer: Each time we play the lottery, the outcome is a Bernoulli random variable (equal to 1 if we win), with parameter $p = 0.01$. Therefore, the number of days before winning is a geometric random variable with parameter $p = 0.01$. Its expected value is

$$E[X] = \frac{1}{p} = \frac{1}{0.01} = 100$$

1.5.3 Binomial Distribution

The binomial distribution is a discrete probability distribution that calculates the probability an event will occur a specific number of times in a set number of opportunities.

Binomial Random Variable

Definition 1

A binomial rv $X \sim Bin(n, p)$ is a random variable that is the number of successes in n independent Bernoulli trials, each with probability p. The probability of success is p. The probability of failure is 1-p. The number of trials is n.

Definition 2

The binomial distribution is the distribution of the `number of successes = X` in a `fixed number = n` of independent Bernoulli trials.

Parameters

The binomial distribution has two parameters, n and p.

- n: the number of trials.
- p: the event or success probability.

Uses

Use the binomial distribution when your outcome is binary. Binary outcomes have only two possible values that are mutually exclusive.

- Six heads when you toss the coin ten times?
- 12 women in a sample size of 20?
- Three defective items in a batch of 100?
- Two flu infections over 20 years?

Properties

1. Experiment is n trials (n is fixed in advance)
2. Trials are identical and result in a success or a failure (i.e. Bernoulli trials) with P(success) = p and P(failure) = 1 - p.
3. Trials are independent (outcome of one trial does not influence any other)

P.M.F

The *sample space* of binomial random variable is

$$S = \{(x_1, x_2, \ldots, x_n) \mid x_i = \begin{cases} 1 \text{ if success} \\ 0 \text{ if failure} \end{cases}$$

$f(x) = P(X = 0) = P(\{00 \cdots 0\}) = (1-p)^n$
$f(x) = P(X = 1) = P(\{10 \cdots 0, 0100 \ldots, 0 \cdots 01\}) = n * p * (1-p)^{n-1}$
$f(x) = P(X = 2) = P(\{11 \cdots 0, 0110 \ldots, 00 \cdots 11\}) = \binom{n}{2} p^2 (1-p)^{n-2}$

`Explanation P(X=2):` Among n number of fixed trials, we have 2 bernoulli trials successes with probability P and rest are failures bernoulli trails with probability (1-p). So, we need to choose 2 from n to get the exact probability of success.

$$f(x) = P(X = x) = \binom{n}{x} p^x (1-p)^{n-x} \cdot I_{\{1,2,3,\ldots\}}(x)$$

Where k = 1 (success) and n-k = 0 (failure).

Suppose n = 4

$\mathrm{P}(X = 3) = \mathrm{P}(\text{SSSF or SSFS or SFSS or FSSS })$

Binomial Theorem

$\sum_{k=0}^{n} \binom{n}{k} p^k (1-p)^{n-k} = 1$

Mean (Expected Value)

The *expected value* of a binomial random variable X is

$$\begin{aligned} E[X] &= \sum_k kP(X = k) \\ &= \sum_{k=0}^{n} k \binom{n}{k} p^k (1-p)^{n-k} \\ &= n * p \end{aligned}$$

Proof

$$\begin{aligned} &= \mathrm{E}\left[\sum_{i=1}^{n} Y_i\right] \quad \text{(representation as a sum of } n \text{ independent Bernoulli r.v.)} \\ &= \sum_{i=1}^{n} \mathrm{E}\left[Y_i\right] \quad \text{(linearity of the expected value)} \\ &= \sum_{i=1}^{n} p \quad \text{(expected value of a Bernoulli r.v.)} \\ &= np \end{aligned}$$

RECALL: Bern(p) has expected value p. x1, x2 … xn are independent bern p. so $sum_{k=1}^{n} X_n = sum_{k=1}^{n} E[X_n] = n * p$

Variance

The variance of a binomial random variable X is

$V(X) = E(X^2) - E(X)^2 = n * p * (1 - p)$

`Recall:` Bern(p) has variance p * (1-p).

1.5.4 Negative Binomial Distribution

The negative binomial distribution is almost the same as a binomial distribution with one difference

- Binomial distribution has a fixed number of trials.

Repeat independent Bernoulli trials until a total of r successes is obtained. The negative binomial random variable X counts the number of failures before the rth success.

Negative Binomial Random Variable

The negative binomial rv $X \sim NB(r, p)$ is the distribution of the `number of trials = X` needed to get a `fixed number of successes = r`.

Properties

1. The number of successes r is fixed in advance.
2. Trials are identical and result in a success or a failure (Bernoulli trials with P(success) = p and P(failure) = 1-p.
3. Trials are independent (outcome of one trial does not influence any other)

PMF

$S = \{(x_1, x_2, \ldots, x_n) \mid x_i = \begin{cases} 1 \text{ if success on ith trail} \\ 0 \text{ if failure ith trail} \end{cases} \quad and \sum_{i=1} = r$

$P(y = 0) = P(\{11111\}) = (p)^5$

$P(Y = 1) = P(\{011111, 101111, 110111, 111011, 111101\}) = \binom{5}{4} p^5 (1 - p)^{5-4}$

$P(Y = 2) = \binom{6}{4} p^5 (1 - p)^{5-4}$

$P(X = k) = \binom{k+r-1}{r-1} (1 - p)^k p^r$

Mean (Expected Value)

$E(X) = \sum_k kP(X = k)$

$E(X) = \frac{r(1-p)}{p}$

Variance

$V(X) = \frac{r(1-p)}{p^2}$

Relationship between Geometric and Negative Binomial rv

$X \sim Geom(p)$

= Repeated, independent, identical, Bernoulli trails util first successes.

$Y \sim NB(1, p)$

= Count the number of failure until first success util first successes. =

$\underbrace{}_{Failure} \underbrace{}_{Failure} success$

`Note:` Y = X - 1. then E(Y) = E(X) - 1 = 1/p - 1 = $\frac{1-p}{p}$

$NB(r, p)$ = $\underbrace{}_{Failure} \underbrace{}_{Failure} success \underbrace{}_{Failure} \underbrace{}_{Failure} success \underbrace{}_{Failure} \underbrace{}_{Failure} rthsuccess$

means we have stack geometric rv in a row rth time. that's why we multiply by r in expected value and variance in NB rv.

1.5.5 Poisson Distribution

The Poisson distribution is a discrete probability distribution that describes probabilities for counts of events that occur in a specified observation space. It is named after **Siméon Denis Poisson**.

Suppose that an event can occur several times within a given unit of time. When the total number of occurrences of the event is unknown, we can think of it as a random variable.

Poisson Random Variable

A Poisson rv $X \sim Poisson(\lambda)$ is a discrete rv that describes the total number of events that happen in a certain time period.

Parameter

The Poisson distribution is defined by a single parameter, lambda (),

which is the mean number of occurrences during an observation unit. A rate of occurrence is simply the mean count per standard observation period. For example, a call center might receive an average of 32 calls per hour.

Uses

1. # of vehicles crossing a bridge in one day
2. # of gamma rays hitting a satellite per hour
3. # of cookies sold at a bake sale in one hour
4. # of customers arriving at a bank in a week

PMF

A discrete random variable X has Poisson distribution with parameter (λ > 0) if the probability mass function of X is

$$f(x) = P(X = x) = \begin{cases} \frac{e^{-\lambda}\lambda^x}{x!} & , x = 0, 1, 2, \ldots \\ 0 & , \text{ otherwise} \end{cases}$$

which may also be written as

$$f(x) = \frac{e^{-\lambda}\lambda^x}{x!} I_{\{0,1,2,\ldots\}}(x)$$

where

- k is the number of occurrences ($k = 0, 1, 2 \ldots$) It could be zero because nothing happened in that time period.
- e} is (e = 2.71828..)

While this pmf might appear to be highly structured, it really is the epitome of randomness. Imagine taking a 20 acre plot of land and dividing it into 1 square foot sections. (There are 871,200 sections!) Suppose you were able to scatter 5 trillion grass seeds on this land in a completely random way that does not favor one section over another. One can show that the number of seeds that fall into any one section follows a Poisson distribution with some parameter . More specifically, one can show that the Poisson distribution is a limiting case of the binomial distribution when n gets really large and p get really small. "Success" here is the event that any given seed falls into one particular section. We then want to count the number of successes in 5 trillion trials.

In general, the Poisson distribution is often used to describe the distribution of rare events in a large population.

All probabilities sum to 1

$\sum_{k=0}^{\infty} P(X = k) = \sum_{k=0}^{\infty} \frac{\lambda^k}{k!} e^{-\lambda} = e^{-\lambda} \sum_{k=0}^{\infty} \frac{\lambda^k}{k!} = e^{-\lambda} * e^{\lambda} = 1$

Mean (Expected Value)

$E(X) = \sum_{k=0}^{\infty} kP(X = k) = \sum_{k=0}^{\infty} k \frac{\lambda^k}{k!} e^{-\lambda} = \lambda \sum_{k=1}^{\infty} \frac{\lambda^{k-1}}{(k-1)!} e^{-\lambda} = \lambda$

$E\left(X^2\right) = \sum_{k=0}^{\infty} k^2 P(X = k) = \sum_{k=0}^{\infty} k^2 \frac{\lambda^k}{k!} e^{-\lambda} = \lambda(\lambda + 1)^e$

Variance

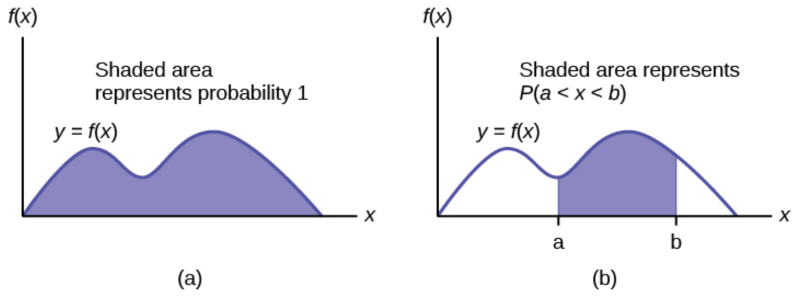
$V(X) = E\left(X^2\right) - (E(X))^2 = \lambda(\lambda + 1) - \lambda^2 = \lambda$

1.6 Continuous Distributions

1.6.1 Definition

A random variable is continuous if possible values comprise either a single interval on the number line or a union of disjoint intervals. X = f(x) is the probability density function of the continues random variable X.

We model a continuous random variable with a curve f(x), called a probability density function (pdf).



Applications

- In the study of the ecology of a lake, a rv X could be the depth measurements at randomly chosen locations.
- In a study of a chemical reaction, Y could be the concentration level of a particular chemical in solution.
- In a study of customer service, W could be the time a customer waits for service.
- f(x) represents the height of the curve at point x.
- For continuous random variables probabilities are areas under the curve.

Attention: We can't model continuous random variable using discrete rv method.

$$P(a \leq X \leq b) = \int_a^b f(x)dx$$

Properties

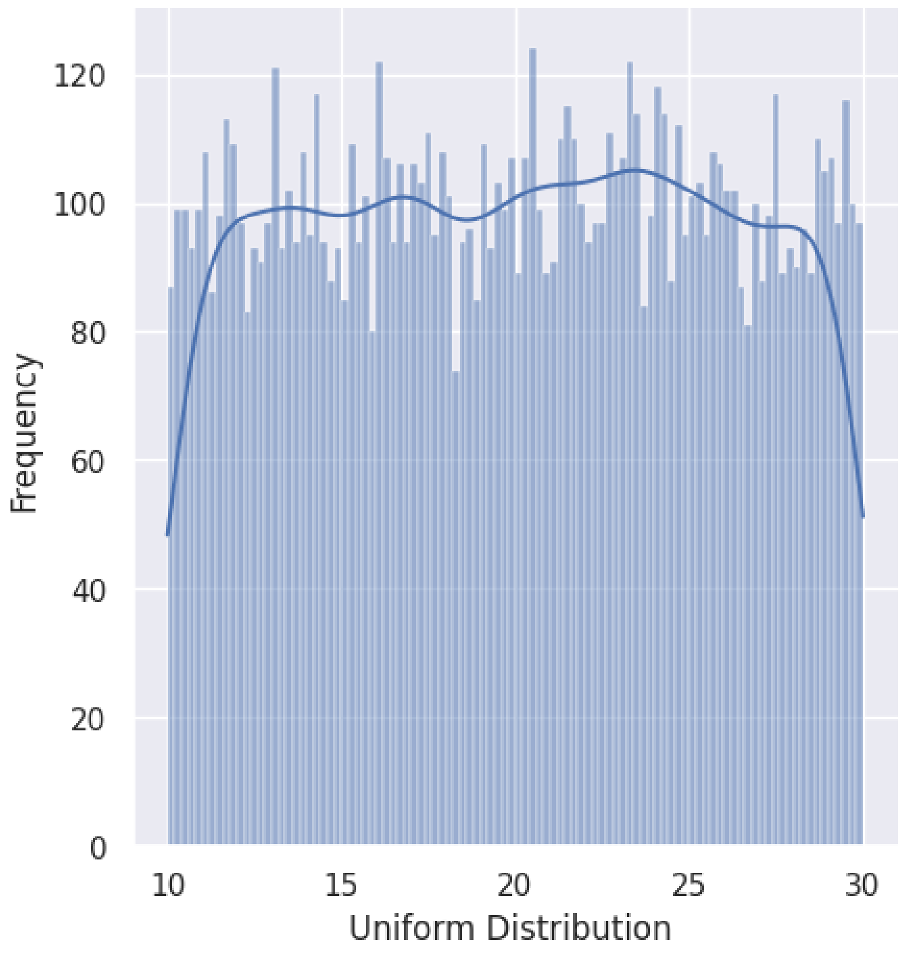
1. The probability density function $f : (-\infty, \infty) \to [0, \infty)$ so $f(x) \geq 0$.
2. $P(-\infty < X < \infty) = \int_{-\infty}^{\infty} f(x)dx = 1 = P(S)$
3. $P(a \leq X \leq b) = \int_a^b f(x)dx$

Note: $P(X = a) = \int_a^a f(x)dx = 0$ for all real numbers a

1.6.2 Uniform rv

Random variable $X \sim U[a, b]$ has the uniform distribution on the interval [a, b] if its density function is

```python
import torch
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import uniform

sns.set_theme(style="darkgrid")

# random numbers from uniform distribution
n = 10000
start = 10
width = 20
data_uniform = uniform.rvs(size=n, loc = start, scale=width)
ax = sns.displot(data_uniform,
                  bins=100,
                  kde=True)
ax.set(xlabel='Uniform Distribution ', ylabel='Frequency')
plt.show()
```



$$f(x)=\begin{cases}\frac{1}{b-a} & \text{for } a\le x\le b,\\ 0 & \text{for } x<a \text{ or } x>b\end{cases}=\frac{1}{b-a}\cdot I_{(a,b)}(x)$$

CDF

$$\begin{aligned}F(x)=P(X\le x)&=\int_{-\infty}^{x} f(t)dt\\ &=\int_a^x \frac{1}{b-a}dt\end{aligned}$$

$$F(x)=\begin{cases}0 & \text{for } x<a\\ \frac{x-a}{b-a} & \text{for } a\le x\le b\\ 1 & \text{for } x>b\end{cases}$$

Expected Value and Variance

$$f(x)=\begin{cases}\frac{1}{b-a} & \text{for } a\le x\le b,\\ 0 & \text{for } x<a \text{ or } x>b\end{cases}$$

$$\begin{aligned}E(X)&=\int_a^b x\cdot\frac{1}{b-a}dx=\frac{1}{b-a}\frac{x^2}{2}\Big|_a^b=\frac{b^2-a^2}{2(b-a)}=\frac{b+a}{2}\\ E\left(X^2\right)&=\int_a^b x^2\frac{1}{b-a}dx=\frac{1}{b-a}\frac{x^3}{3}\Big|_a^b=\frac{b^3-a^3}{3(b-a)}=\frac{b^2+ab+a^2}{3}\\ V(X)&=E\left(X^2\right)-(E(X))^2\\ &=\frac{b^2+ab+a^2}{3}-\left(\frac{b+a}{2}\right)^2=\frac{(b-a)^2}{12}\end{aligned}$$

Example

For random variable $X\sim U(0,23)$. Find P(2 < X < 18)

$P(2<X<18)=(18-2)\cdot\frac{1}{23-0}=\frac{16}{23}$

1.6.3 Exponential Distribution

The exponential distribution is a continuous probability distribution that often concerns the amount of time until some specific event happens. It is a process in which events happen continuously and independently at a constant average rate. The exponential distribution has the key property of being memoryless.

Applications

The family of exponential distributions provides probability models that are widely used in engineering and science disciplines to describe **time-to-event** data.

- Time until birth
- Time until a light bulb fails
- Waiting time in a queue
- Length of service time
- Time between customer arrivals
- the amount of money spent by the customer
- Calculating the time until the radioactive particle decays

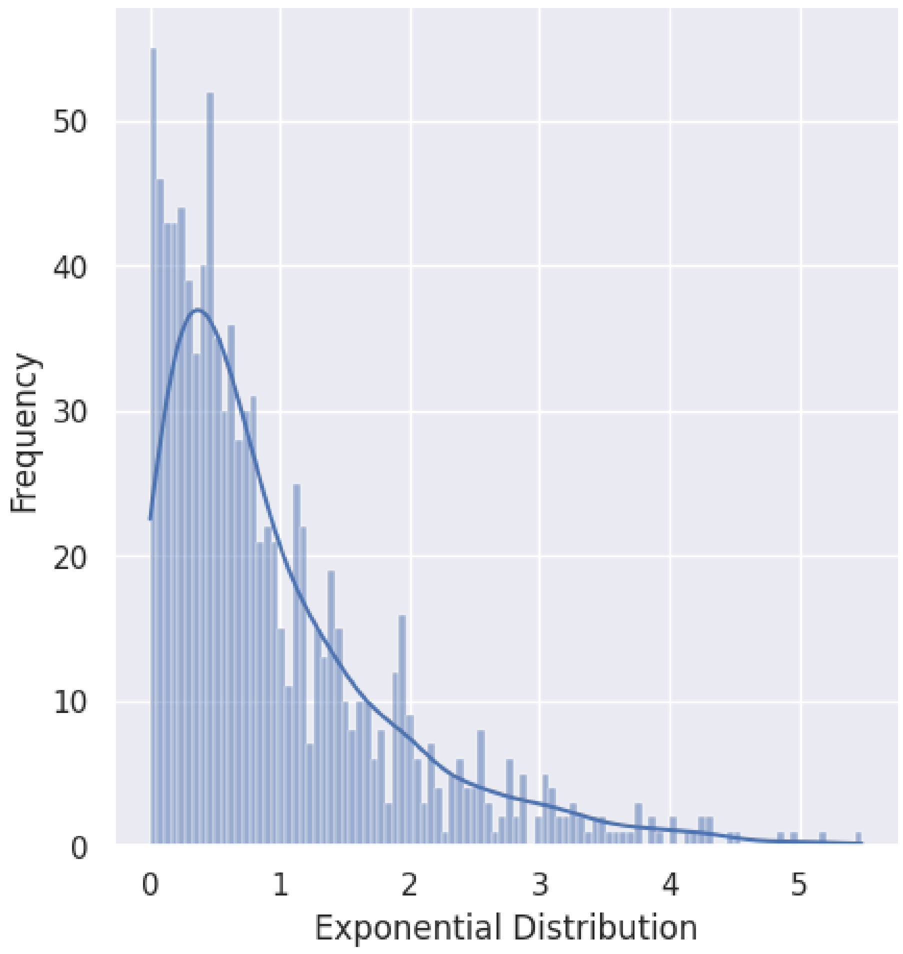
PDF

The continuous random variable, say X is said to have an exponential distribution, if it has the following probability density function:

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & x \ge 0, \\ 0 & x < 0. \end{cases} = \lambda e^{-\lambda x} I_{(0,\infty)}(x)$$

is called the distribution rate.

```
import torch
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import expon

sns.set_theme(style="darkgrid")

data_expon = expon.rvs(scale=1,loc=0,size=1000)
ax = sns.displot(data_expon,
                  kde=True,
                  bins=100)
ax.set(xlabel='Exponential Distribution', ylabel='Frequency')
plt.show()
```



Expected Value

The mean of the exponential distribution is calculated using the integration by parts.

$$
\begin{aligned}
E[X] &= \int_0^\infty x f(x) dx = \int_0^\infty x \lambda e^{-\lambda x} dx \\
&= \lambda \left[\left| \frac{-x e^{-\lambda x}}{\lambda} \right|_0^\infty + \frac{1}{\lambda} \int_0^\infty e^{-\lambda x} dx \right] \\
&= \lambda \left[0 + \frac{1}{\lambda} \frac{-e^{-\lambda x}}{\lambda} \right]_0^\infty \\
&= \lambda \frac{1}{\lambda^2} \\
&= \frac{1}{\lambda}
\end{aligned}
$$

$$
\begin{aligned}
E[X^2] &= \int_0^\infty x^2 f(x) dx \\
&= \int_0^\infty x^2 \lambda e^{-\lambda x} dx \\
&= \frac{2}{\lambda^2}
\end{aligned}
$$

Variance

To find the variance of the exponential distribution, we need to find the second moment of the exponential distribution

$$\begin{aligned} V(X) &= E(X^2) - E(X)^2 \\ &= \frac{2}{\lambda^2} - (\frac{1}{\lambda})^2 \\ &= \frac{1}{\lambda^2} \end{aligned}$$

Properties

The most important property of the exponential distribution is the memoryless property. This property is also applicable to the geometric distribution.

1.6.4 Normal (Gaussian) Distribution

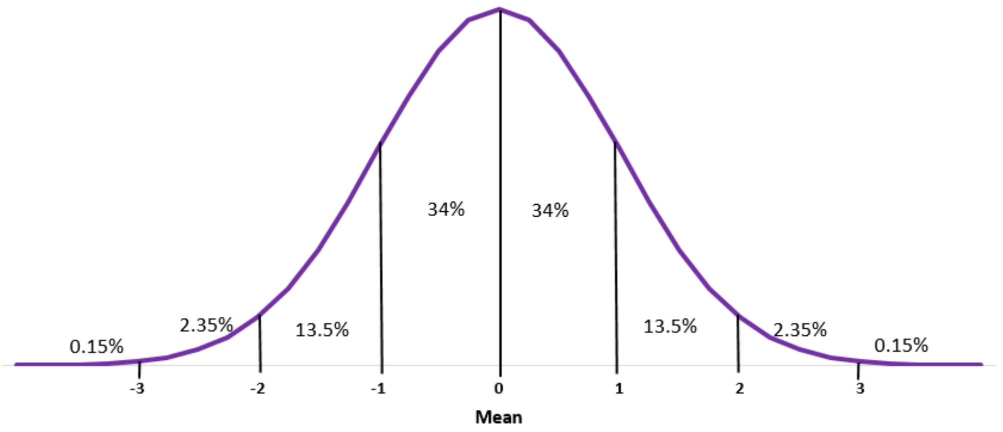
It is often called **Gaussian distribution**, in honor of Carl Friedrich Gauss (1777-1855), an eminent German mathematician who gave important contributions towards a better understanding of the normal distribution.

Normal Random Variable

A continuous random variable $X \sim N(\mu, \sigma^2)$ has the normal distribution with parameters μ and σ^2 if its density is given by

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2} \text{ for } -\infty < x < \infty$$

A normal distribution is a distribution that is solely dependent on two parameters of the data set: mean and the standard deviation of the sample.



Attention: This characteristic of the distribution makes it extremely simple for statisticians and hence any variable that exhibits normal distribution is feasible to be forecasted with higher accuracy. Essentially, it can help in simplying the model.

Parameters

- **Mu** is a location parameter. If you change the value of Mu, the entire bell curve is going to slide around.
- If you increase **Sigma squared**, it's going to get fatter and therefore shorter because the total area is one, So if it gets fatter, it has to come down. If Sigma squared gets smaller, it's going to get really tall and thin.

Properties

Normal distribution is simple to explain: Why?

The reasons are:

1. The mean, mode, and median of the distribution are equal.
2. We only need to use the mean and standard deviation to explain the entire distribution.

- f(x) is symmetric around $x = \mu$ as a consequence, deviations from the mean having the same magnitude.
- f(x) > 0 for all x and $\int_{-\infty}^{\infty} f(x)dx = 1$.
- $\mu + \sigma$ and $\mu - \sigma$ are inflection points on f(x).
- Mean and median are equal; both are located at the center of the distribution.

Why is it so important

The normal distribution is extremely important because:

- many real-world phenomena involve random quantities that are approximately normal
- it plays a crucial role in the Central Limit Theorem, one of the fundamental results in statistics;
- its great analytical tractability makes it very popular in statistical modelling.

The following variables are close to normally distributed variables:

1. Height of a population
2. Blood pressure of adult human
3. Position of a particle that experiences diffusion
4. Measurement errors
5. Residuals in regression
6. Shoe size of a population
7. Amount of time it takes for employees to reach home
8. A large number of educational measures

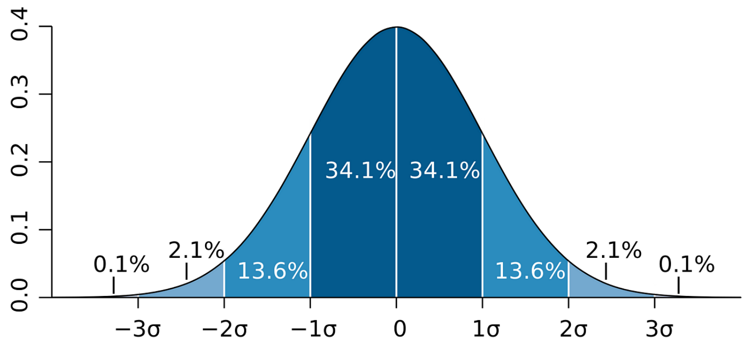
But how are so many variables approximately normally distributed?

Let's consider that the height of a population is a random variable. We can take a sample of heights, plot its distribution and calculate the sample mean. When we repeat this experiment whilst we increase the number of samples then the mean of the samples will end up being very close to normality.

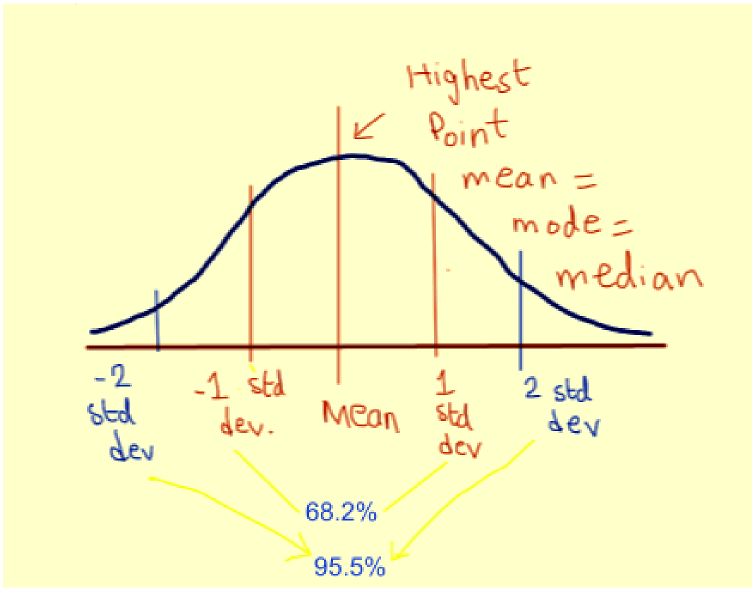
This is known as the Central Limit Theorem.

Probability Density Function

If we plot the normal distribution density function, it's curve has the following characteristics:



- Mean is the center of the curve. This is the highest point of the curve as most of the points are at the mean.
- There is an equal number of points on each side of the curve. The center of the curve has the most number of points.
- The total area under the curve is the total probability of all of the values that the variable can take.
- The total curve area is therefore 100%



- Approximately 68.2% of all of the points are within the range -1 to 1 standard deviation.
- About 95.5% of all of the points are within the range -2 to 2 standard deviations.

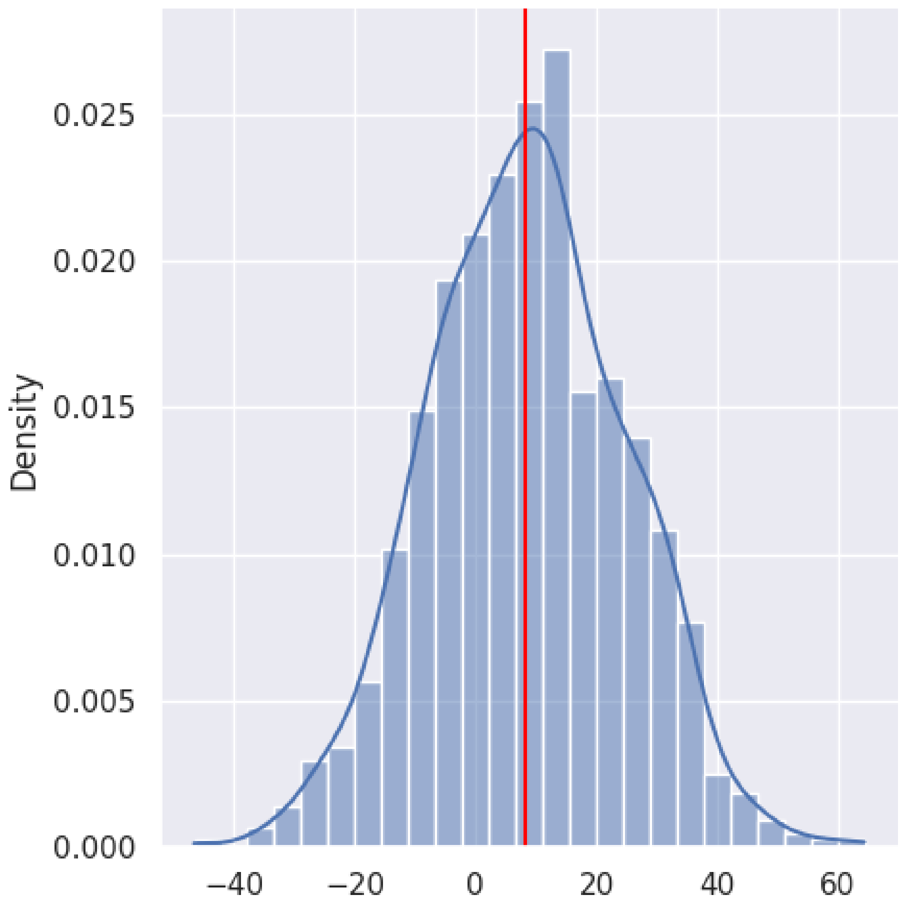
- About 99.7% of all of the points are within the range -3 to 3 standard deviations.

This allows us to easily estimate how volatile a variable is and given a confidence level, what its likely value is going to be. As an instance, in the grey bell-shaped curve above, there is a 68.2% chance that the value of the variable will be within 101–99.

Normal Probability Distribution Function

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma}e^{-(x-\mu)^2/2\sigma^2} = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \text{ for } -\infty < x < \infty$$

```python
import torch
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm


sns.set_theme(style="darkgrid")
sample = torch.normal(mean = 8, std = 16, size=(1,1000))

sns.displot(sample[0], kde=True, stat = 'density',)
plt.axvline(torch.mean(sample[0]), color='red', label='mean')

plt.show()
```

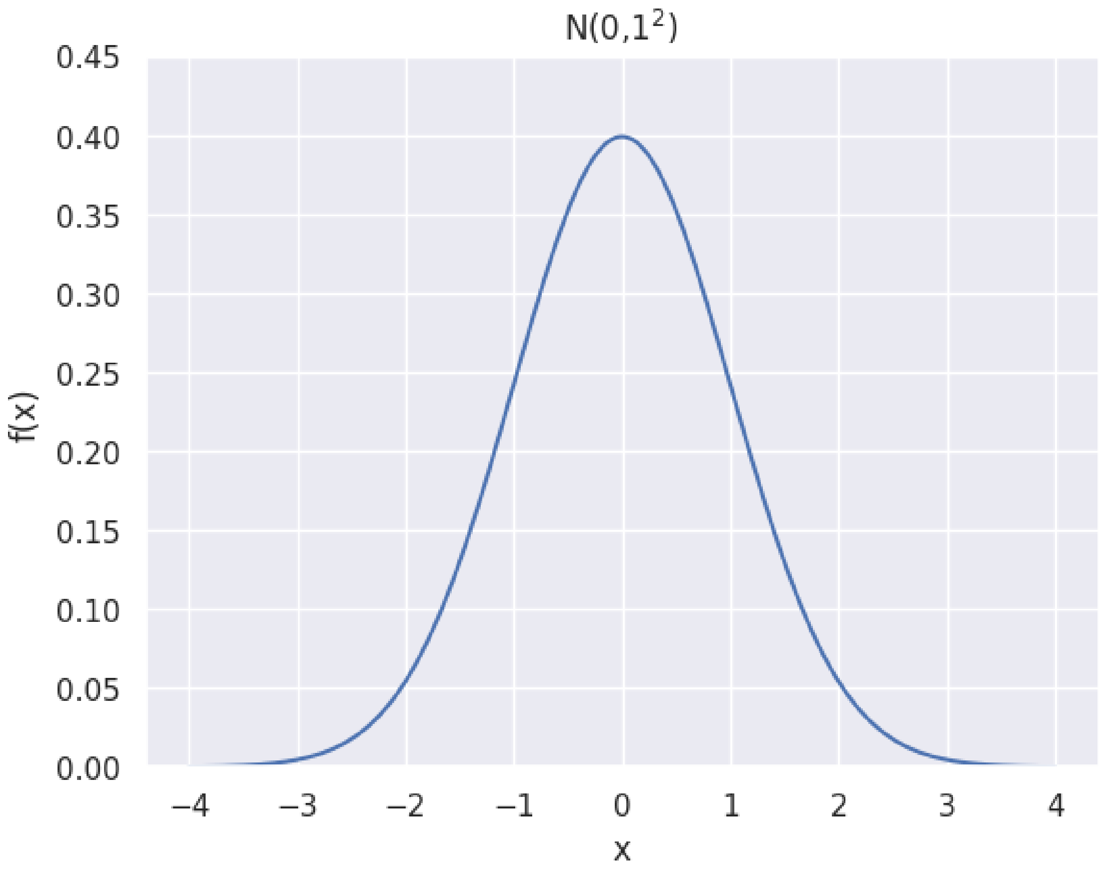


norm.pdf returns a PDF value. The following is the PDF value when =1, =0, =1.

We graph a PDF of the normal distribution using scipy, numpy and matplotlib. We use the domain of 4<<4, the range of 0<()<0.45, the default values =0 and =1. plot(x-values,y-values) produces the graph.

```
print(norm.pdf(x=1.0, loc=0, scale=1))

x = torch.arange(-4,4,0.001)
fig, ax = plt.subplots()

ax.set_title('N(0,$1^2$)')
ax.set_xlabel('x')
ax.set_ylabel('f(x)')
ax.plot(x, norm.pdf(x))
ax.set_ylim(0,0.45)
plt.show()
```

```
0.24197072451914337
```



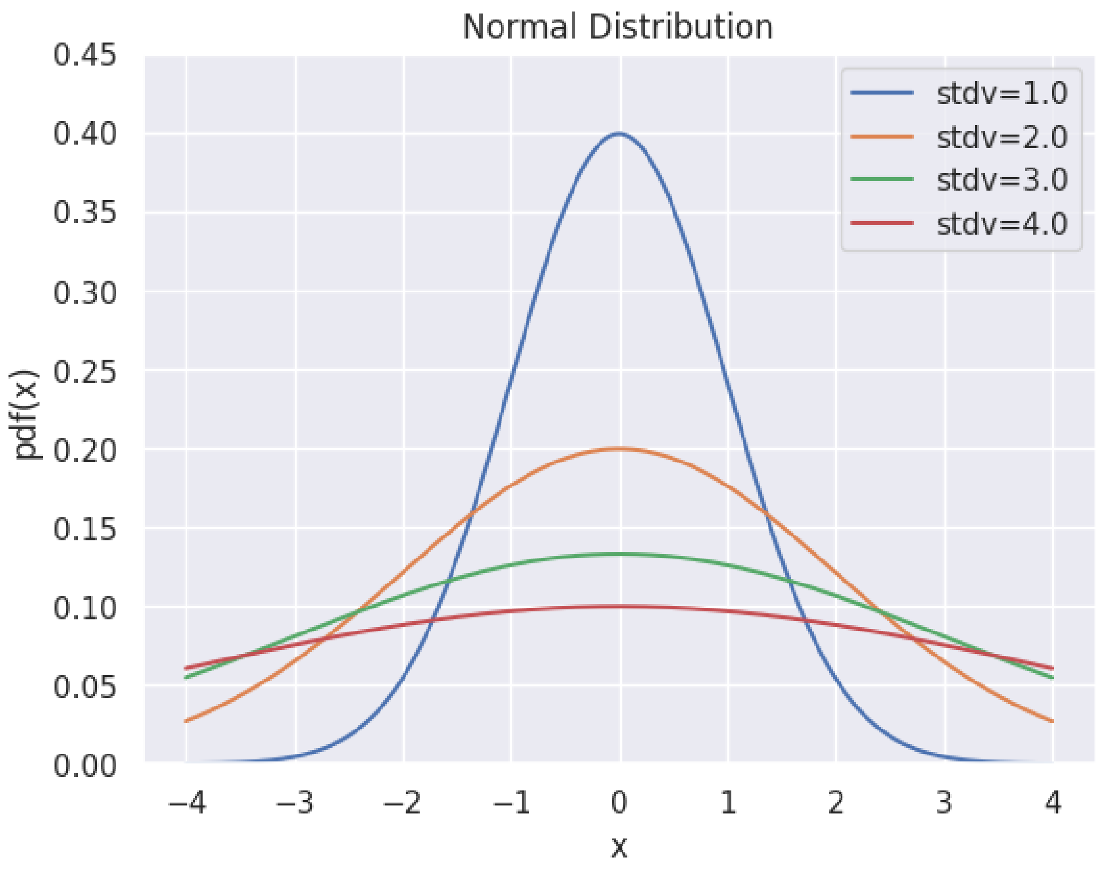
A normal curve is smooth bell-shaped. It is symmetrical about the = and has a maximum point at =.

Normal distribution PDF with different standard deviations

Let's plot the probability distribution functions of a normal distribution where the mean has different standard deviations.

scipy.norm.pdf has keywords, loc and scale. The location (loc) keyword specifies the mean and the scale (scale) keyword specifies the standard deviation.

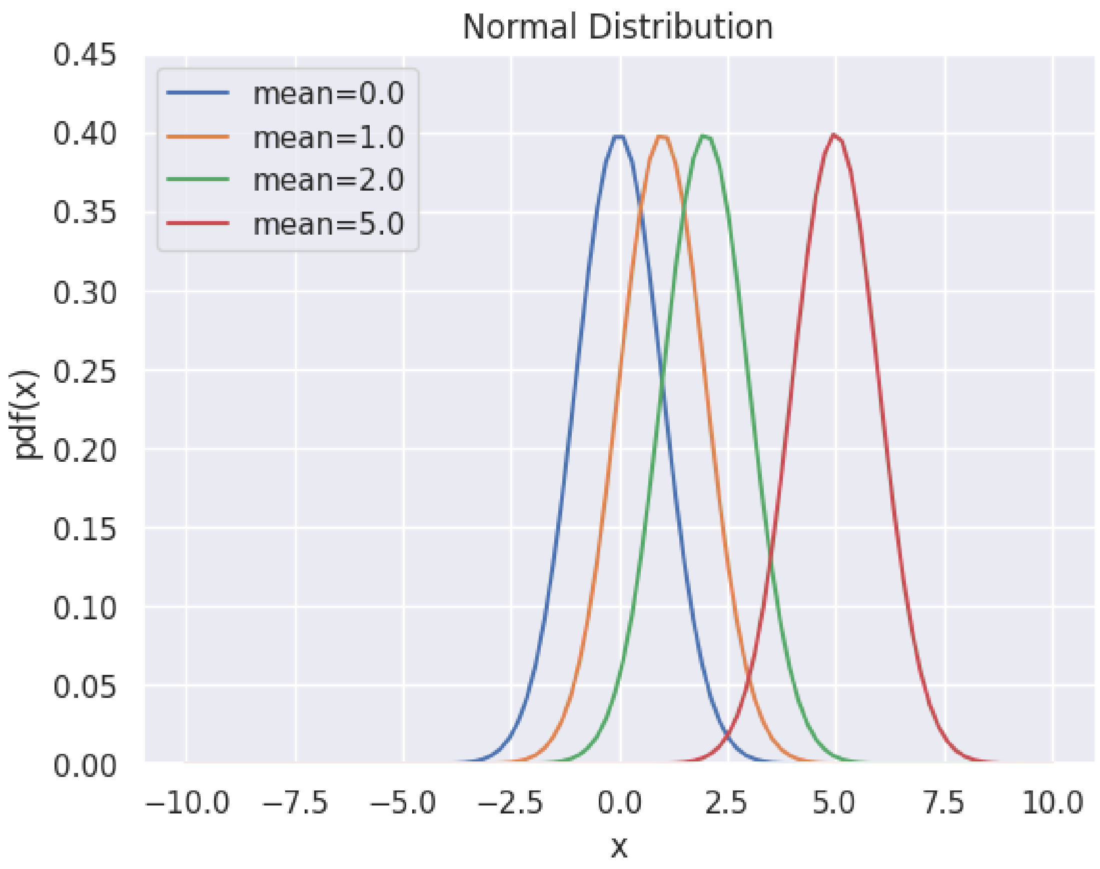
```
fig, ax = plt.subplots()
x = torch.arange(-4,4,0.001)

stdvs = [1.0, 2.0, 3.0, 4.0]
for s in stdvs:
    ax.plot(x, norm.pdf(x,scale=s), label='stdv=%.1f' % s)

ax.set_xlabel('x')
ax.set_ylabel('pdf(x)')
ax.set_title('Normal Distribution')
ax.legend(loc='best', frameon=True)
ax.set_ylim(0,0.45)
ax.grid(True)
```



Normal distribution PDF with different means

Let's plot the probability distribution functions of a normal distribution where the mean has different values.

```
fig, ax = plt.subplots()
x = torch.linspace(-10,10,100)

means = [0.0, 1.0, 2.0, 5.0]
for mean in means:
    ax.plot(x, norm.pdf(x,loc=mean), label='mean=%.1f' % mean)

ax.set_xlabel('x')
ax.set_ylabel('pdf(x)')
ax.set_title('Normal Distribution')
ax.legend(loc='best', frameon=True)
ax.set_ylim(0,0.45)
ax.grid(True)
```



The mean of the distribution determines the location of the center of the graph. As you can see in the above graph, the shape of the graph does not change by changing the mean, but the graph is translated horizontally.

A cumulative normal distribution function

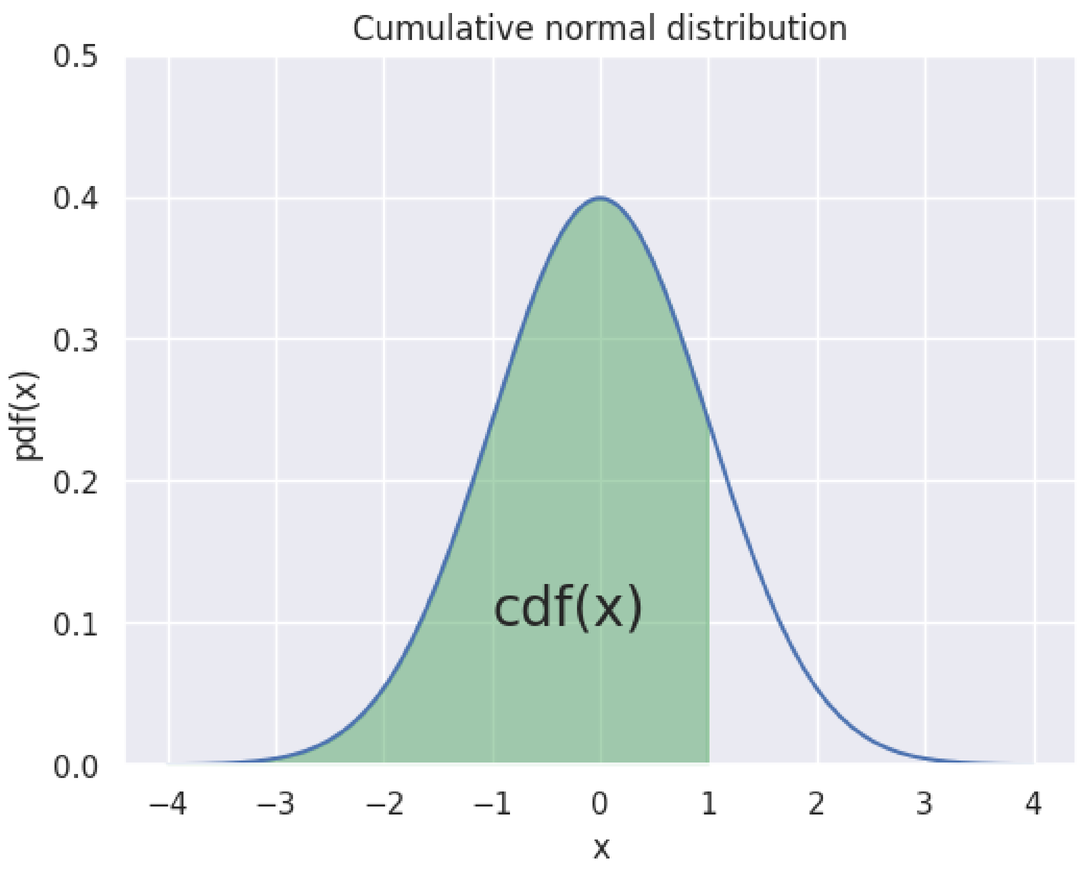
The cumulative distribution function of a random variable X, evaluated at x, is the probability that X will take a value less than or equal to x. Since the normal distribution is a continuous distribution, the shaded area of the curve represents the probability that X is less or equal than x.

```
fig, ax = plt.subplots()
# for distribution curve
x= torch.arange(-4,4,0.001)

ax.plot(x, norm.pdf(x))
ax.set_title("Cumulative normal distribution")
ax.set_xlabel('x')
ax.set_ylabel('pdf(x)')
ax.grid(True)
# for fill_between
px=torch.arange(-4,1,0.01)
ax.set_ylim(0,0.5)
ax.fill_between(px,norm.pdf(px),alpha=0.5, color='g')
# for text
ax.text(-1,0.1,"cdf(x)", fontsize=20)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



Expected Value and Variance

$E(X) = \mu$

$V(X) = \sigma^2$

Problems With Normality

Assuming normality has its own flaws. As an instance, we cannot assume that the stock price follows normal distribution as the price cannot be negative. Therefore the stock price potentially follows a log of the normal distribution to ensure it is never below zero.

We know that the daily returns can be negative, therefore the returns can at times follow a normal distribution. It is not wise to assume that the variable follows a normal distribution without any analysis.

A variable can follow Poisson, Student-t, or Binomial distribution as an instance and falsely assuming that a variable follows normal distribution can lead to inaccurate results.

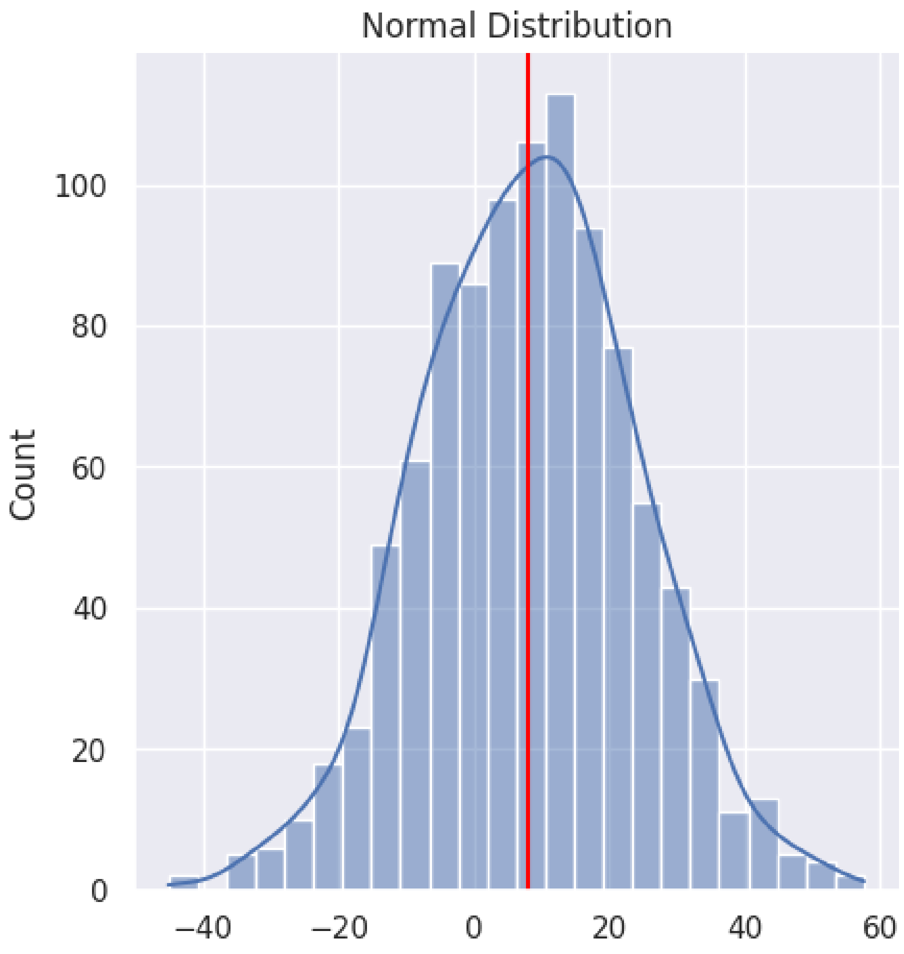
Identify the Normal RV?

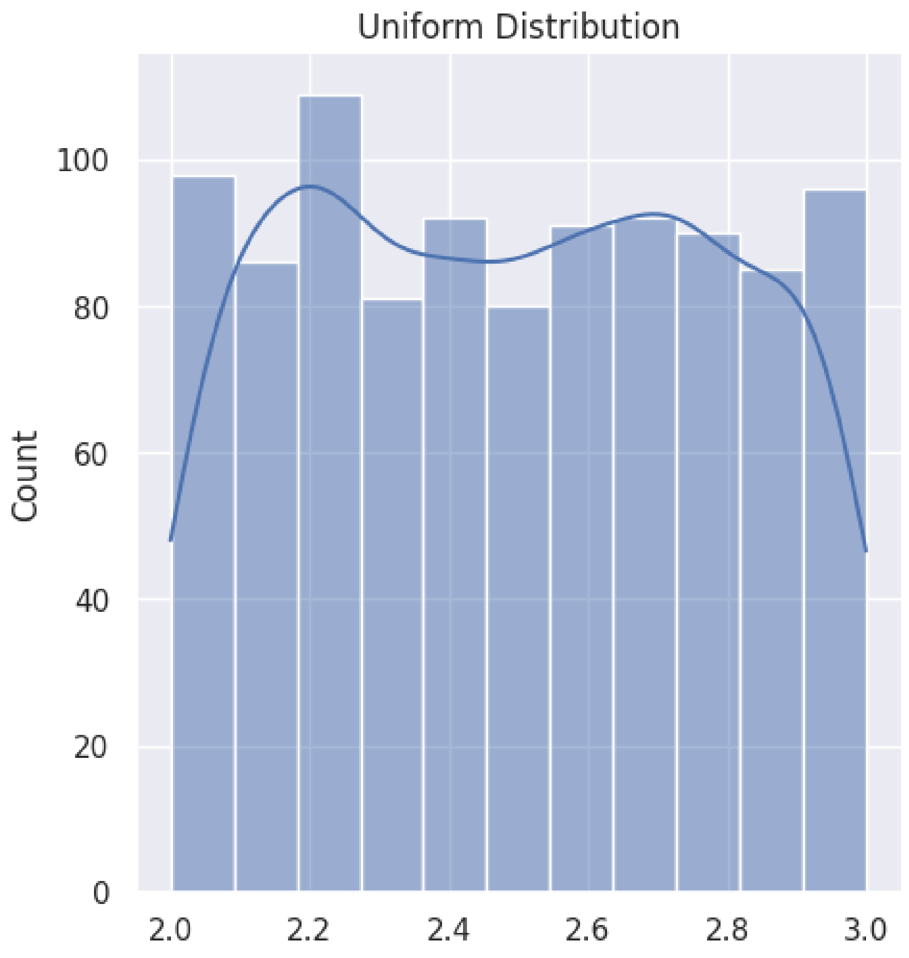
Many methods exist for testing whether a variable has a normal distribution

1. Histogram

The histogram is a data visualization that shows the distribution of a variable. It is a bar graph that shows the frequency of each value in the variable. The histogram is a graphical representation of the distribution of a variable.

```
sample = torch.normal(mean = 8, std = 16, size=(1,1000))
sample2 = torch.distributions.uniform.Uniform(2,3).sample([1,1000])

sns.displot(sample[0], kde=True,).set(title='Normal Distribution')
plt.axvline(torch.mean(sample[0]), color='red', label='mean')

sns.displot(sample2[0], kde=True,).set(title='Uniform Distribution')
plt.show()
```





2. Box Plot

The Box Plot is another visualization technique that can be used for detecting non-normal samples.

The box plot is a graphical representation of the distribution of a variable. It is a graphical representation of the five-number summary of a variable. The five-number summary is the minimum, first quartile, median, third quartile, and maximum of a variable.

3. QQ Plot

QQ Plot stands for Quantile vs Quantile Plot, which is exactly what it does: plotting theoretical quantiles against the actual quantiles of our variable.

The QQ Plot allows us to see deviation of a normal distribution much better than in a Histogram or Box Plot.

1.6.5 Standard Normal Distribution

The normal distribution with parameter values μ = 0 and σ^2 = 1 is called the standard normal distribution.

A rv with the standard normal distribution is denoted by $Z \sim N(0,1)$

If $X \sim N\left(\mu, \sigma^2\right)$ then

$$f_X(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2} \text{ for } -\infty < x < \infty$$

If $Z \sim N(0,1)$ then

$$f_Z(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \text{ for } -\infty < x < \infty$$

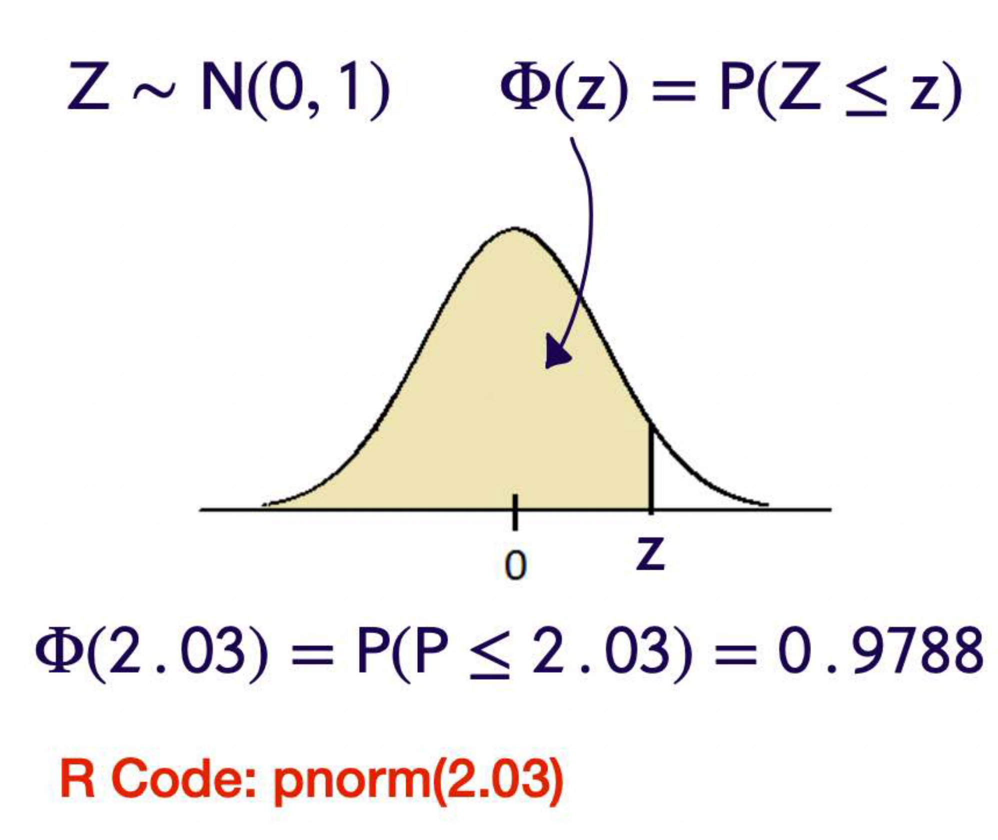
PDF

$f_Z(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \text{ for } -\infty < x < \infty$

Cumulative distribution function

We use special notation to denote the cdf of the standard normal curve

$F(z) = \Phi(z) = P(Z \leq z) = \int_{-\infty}^{z} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx$



Properties

1. The standard normal density function is symmetric about the y axis.
2. The standard normal distribution rarely occurs naturally.
3. Instead, it is a reference distribution from which information about other normal distributions can be obtained via a simple formula.
4. The cdf of the standard normal, $\Phi(z)$ can be found in tables and it can also be computed with a single command in R.
5. As we'll see, sums of standard normal random variables play a large role in statistical analysis.

Proposition

If $X \sim N\left(\mu, \sigma^{2}\right)$, then $\frac{X-\mu}{\sigma} \sim N(0,1)$

$\frac{X-\mu}{\sigma}$ Shifted by μ or (Centered at zero) and scaled by $\frac{1}{\sigma}$ that will give us variance of 1.

$\mathrm{Z} \sim \mathrm{N}(0,1) \Rightarrow \sigma \mathrm{Z}+\mu \sim \mathrm{N}\left(\mu, \sigma^{2}\right)$

Example

Let $X \sim N(2,3)$

$$\begin{aligned} P(X \leq 4.1) &= P\left(\frac{X-\mu}{\sigma} \leq \frac{4.1-2}{\sqrt{3}}\right) \\ &= P(Z \leq 1.21) \end{aligned}$$

Proving this proposition

For any continuous random variable. Suppose we have Y rv, with Desnity function $f_Y(y)$

We know

$P(y \leqslant a) = \int_{-\infty}^{a} f_y(y)dy$

What if

$P(2y \leqslant a)$

= Can't really use the density function until we isolate y =

$P\left(y \leq \frac{a}{2}\right) = \int_{-\infty}^{a/2} f_y(y)dy$

This true for all transformation of Y.

With $P\left(\frac{x-\mu}{\sigma} \leq a\right) = P(x \leq a\sigma + \mu) = \int_{x}^{a\sigma+\mu} \frac{1}{\sqrt{2\pi\sigma}} e^{-(x-\mu)^2/2\sigma^2}$

U subsitution

Let

$u = \frac{x-\mu}{\sigma}$

$du = \frac{1}{\sigma}dx$

SO $= \int_{-\infty}^{a} \frac{1}{\sqrt{2\pi}} e^{-u^2/2} du$ This is density function for N(0,1).

Examples

Find P(X<2) when N(3, 2)?

In norm.cdf, the location (loc) keyword specifies the mean and the scale (scale) keyword specifies the standard deviation.

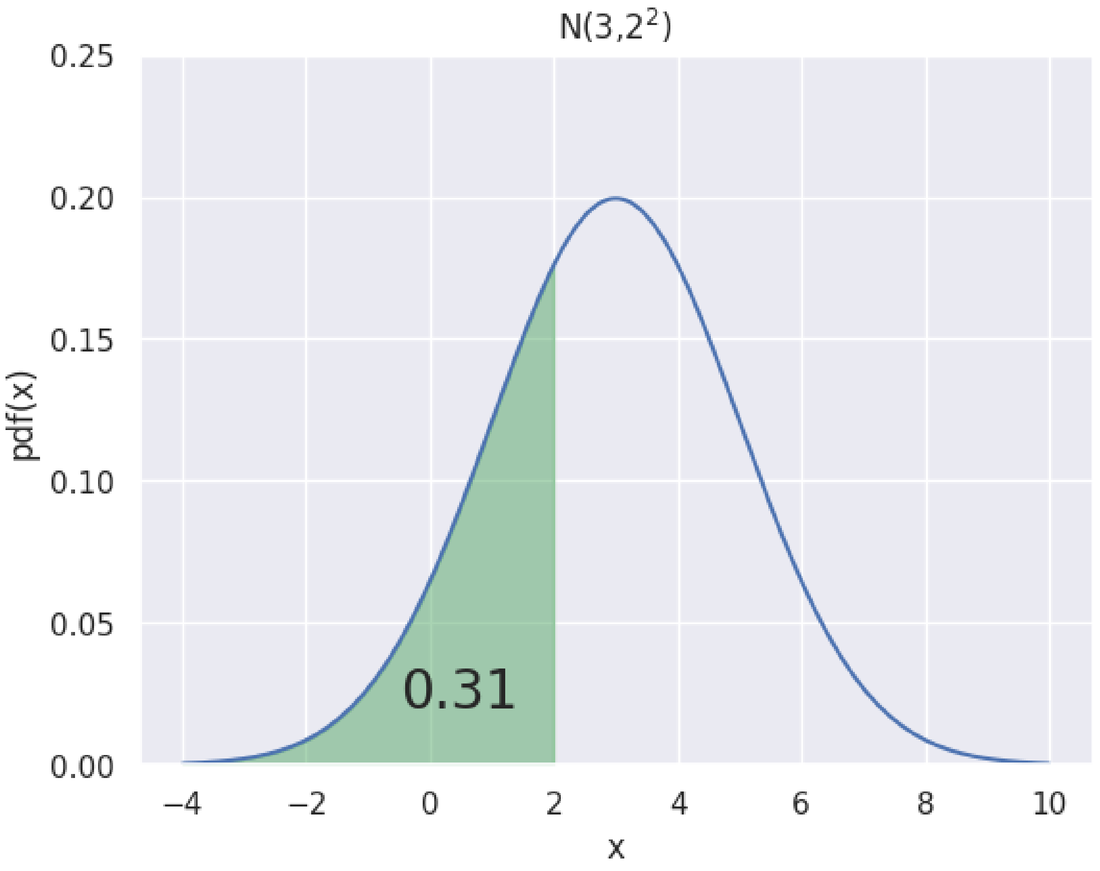
```python
from scipy.stats import norm

val = norm.cdf(x=2, loc=3, scale=2)

print(f"P(X<2) = {val}")


fig, ax = plt.subplots()

x= torch.arange(-4,10,0.001)
ax.plot(x, norm.pdf(x,loc=3,scale=2))
ax.set_title("N(3,$2^2$)")
ax.set_xlabel('x')
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel('pdf(x)')
ax.grid(True)
# for fill_between
px=torch.arange(-4,2,0.01)
ax.set_ylim(0,0.25)
ax.fill_between(px,norm.pdf(px,loc=3,scale=2),alpha=0.5, color='g')
# for text
ax.text(-0.5,0.02,round(val,2), fontsize=20)
plt.show()
```

```
P(X<2) = 0.3085375387259869
```



$$\begin{aligned} P(X \le 2) &= P\left(\frac{X-\mu}{\sigma} \le \frac{2-3}{\sqrt{2}}\right) \\ &= P(Z \le 1.21) \\ &\approx 0.30 \end{aligned}$$

R code: pnorm(1.2)

Find P(X<4.1) when N(2, 3)?

Let $X \sim N(2,3)$. Then

$$\begin{aligned} P(X \leq 4.1) &= P\left(\frac{X-\mu}{\sigma} \leq \frac{4.1-2}{\sqrt{3}}\right) \\ &= P(Z \leq 1.21) \\ &\approx 0.8868 \end{aligned}$$

R Code: pnorm(1.21)

```
z_score <- (4.1 - 2) / sqrt(3)
pnorm(z_score)
```

$$\begin{aligned} & X_1, X_2, \ldots, X_{10} \overset{id}{\sim} N(2,3) \\ & \overline{X} \sim N\left(\mu, \sigma^2/n\right) = N(2, 3/10) \\ & P(\overline{X} \leq 2.3) = P\left(\frac{\overline{X}-\mu_{\overline{X}}}{\sigma_{\overline{X}}} \leq \frac{2.3-2}{\sqrt{3/10}}\right) \\ & \frac{\overline{X}-\mu}{\sigma/\sqrt{n}} = \begin{aligned} &= P(Z \leq 0.5477) \\ &\approx 0.7081 \end{aligned} \end{aligned}$$

Interval between variables

To find the probability of an interval between certain variables, you need to subtract cdf from another cdf.

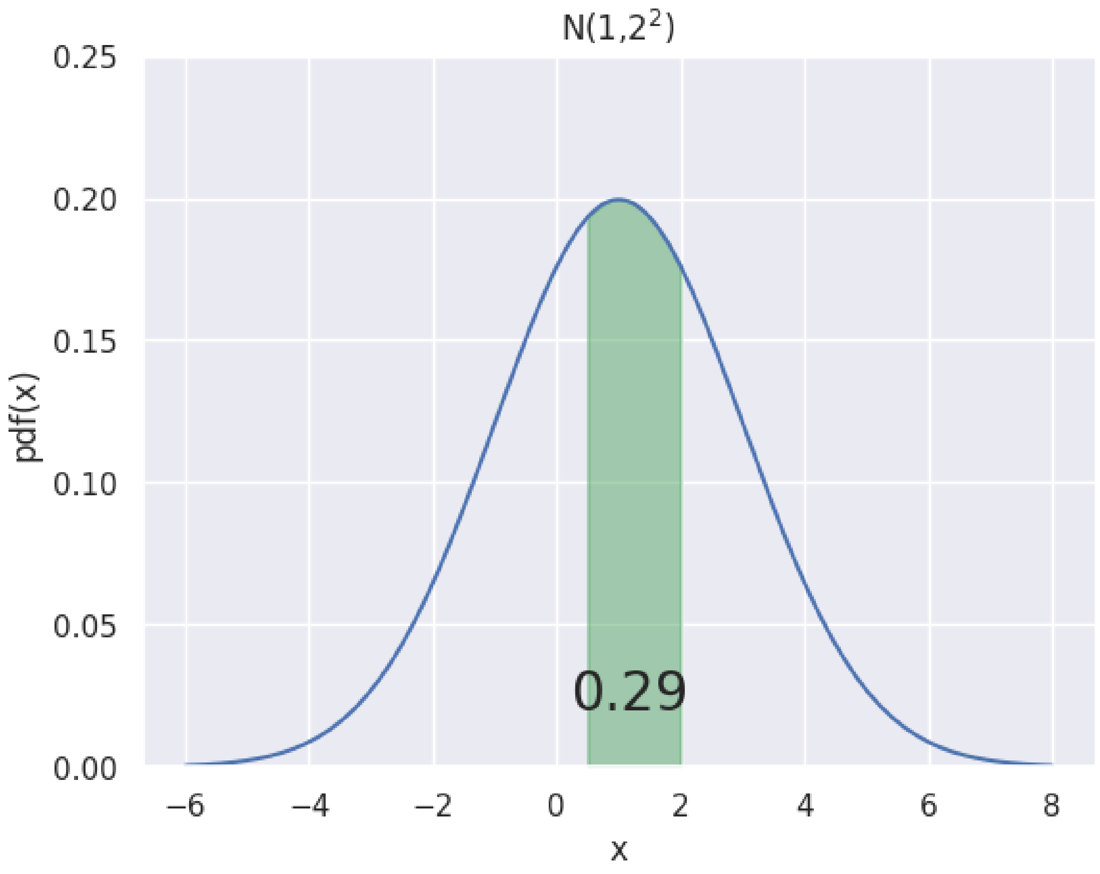
Let's find (0.5<<2) with a mean of 1 and a standard deviation of 2.

```
print(norm(1, 2).cdf(2) - norm(1,2).cdf(0.5))
```

```
0.2901687869569368
```

$$X \sim N\left(1, 2^2\right), P(0.5 < X < 2)$$

```
fig, ax = plt.subplots()

# for distribution curve
x= torch.arange(-6,8,0.001)
ax.plot(x, norm.pdf(x,loc=1,scale=2))
ax.set_title("N(1,$2^2$)")
ax.set_xlabel('x')
ax.set_ylabel('pdf(x)')
ax.grid(True)
px=torch.arange(0.5,2,0.01)
ax.set_ylim(0,0.25)
ax.fill_between(px,norm.pdf(px,loc=1,scale=2),alpha=0.5, color='g')
pro=norm(1, 2).cdf(2) - norm(1,2).cdf(0.5)
ax.text(0.2,0.02,round(pro,2), fontsize=20)
plt.show()
```

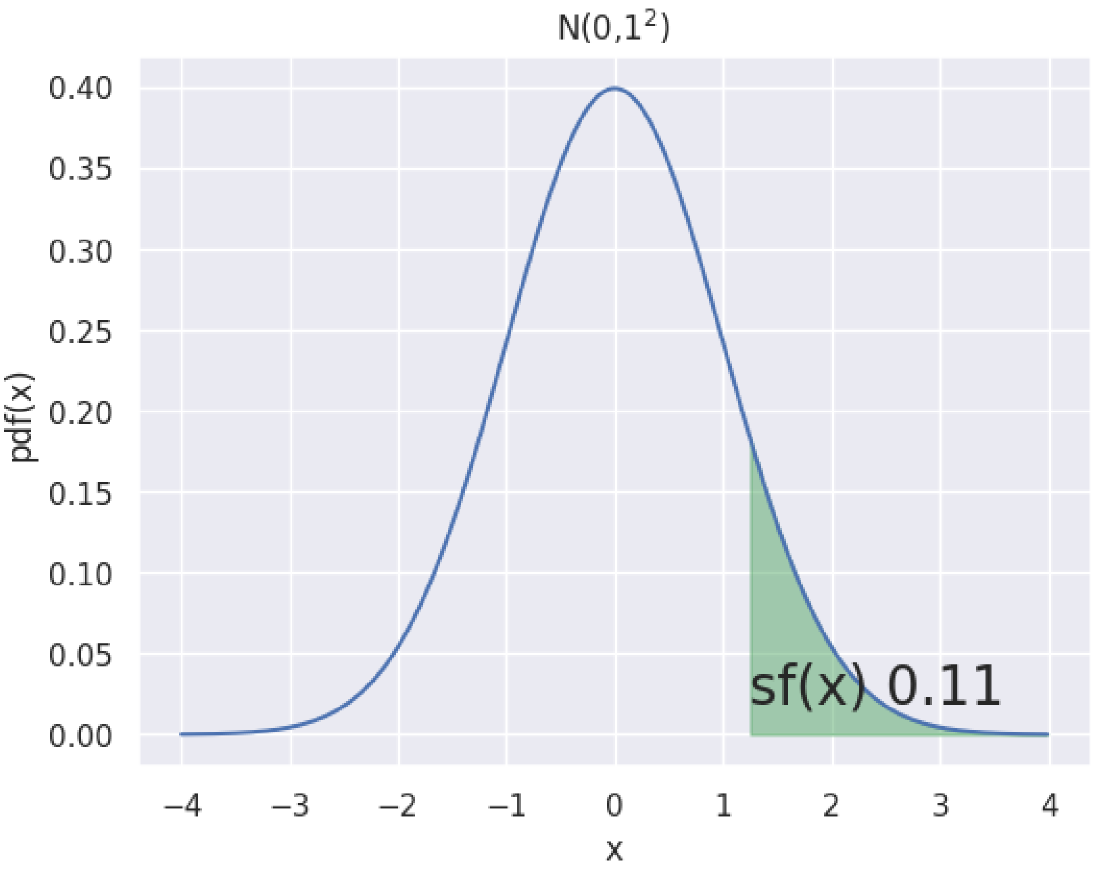


P(Z > 1.25) ?

Let's plot a graph.

```
fig, ax = plt.subplots()
x= torch.arange(-4,4,0.01)
gr4sf=norm.sf(x=1.25, loc=0, scale=1)
print(gr4sf)

ax.plot(x, norm.pdf(x,loc=0,scale=1))
ax.set_title("N(0,$1^2$)")

ax.set_xlabel('x')
ax.set_ylabel('pdf(x)')

ax.grid(True)
px=torch.arange(1.25,4,0.01)
#ax.set_ylim(0,0.15)
ax.fill_between(px,norm.pdf(px,loc=0,scale=1),alpha=0.5, color='g')
ax.text(1.25,0.02,"sf(x) %.2f" %(gr4sf), fontsize=20)
plt.show()
```

```
0.10564977366685535
```



we can use sf which is called the survival function and it returns 1-cdf.

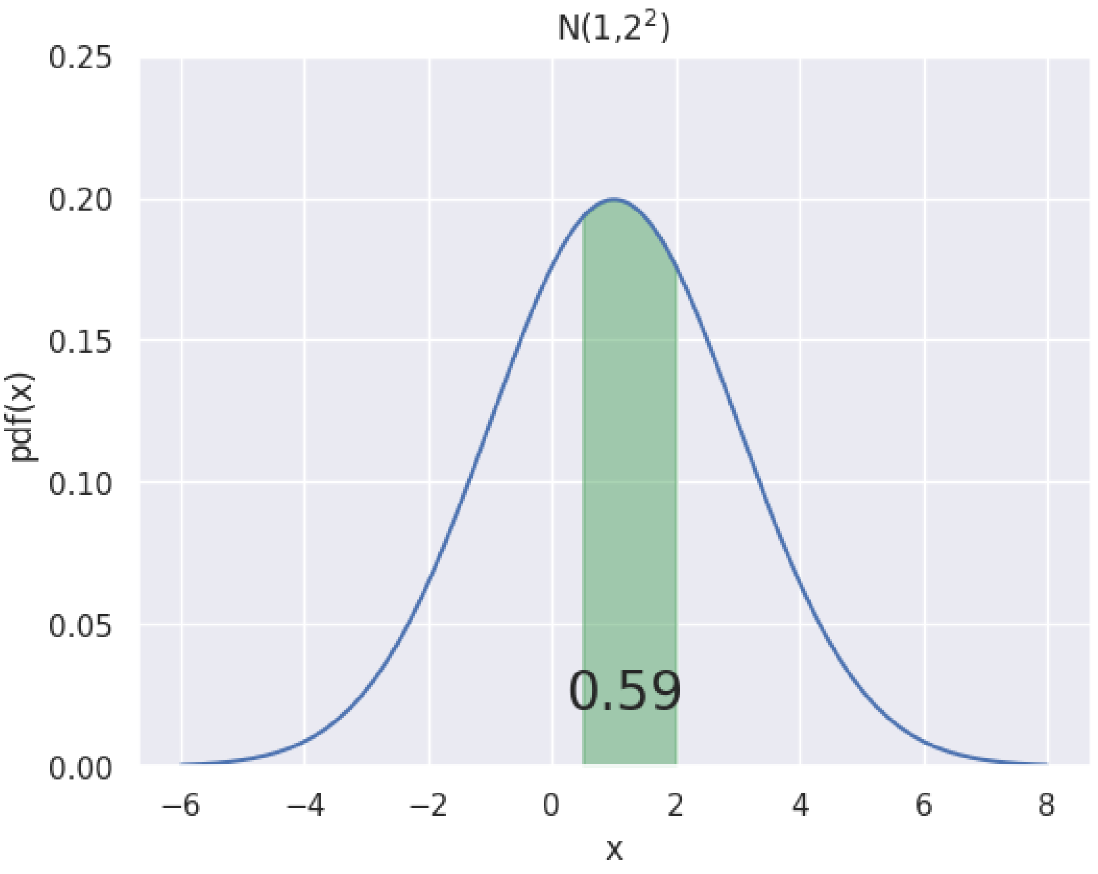
If X = N(1, 4), find P(0 < X < 3.2)?

$$\begin{aligned}
P(0 \leq X \leq 3.2) &= \int_0^{3.2} f_X(x)dx \\
&= P\left(\frac{0-1}{2} \leqslant \frac{x-1}{2} \leqslant \frac{3.2-1}{2}\right) \\
&= P\left(-\frac{1}{2} \leq Z \leq 1.1\right) \\
&= P(z \leq 1.1) - P\left(z < -\frac{1}{2}\right) \\
&= \Phi(1.1) - \Phi\left(-\frac{1}{2}\right) \\
&= .558
\end{aligned}$$

```
print(norm(1, 2).cdf(3.2) - norm(1,2).cdf(0))
```

```
0.5557964003276304
```

```
fig, ax = plt.subplots()

# for distribution curve
x= torch.arange(-6,8,0.001)
ax.plot(x, norm.pdf(x,loc=1,scale=2))
ax.set_title("N(1,$2^2$)")
ax.set_xlabel('x')
ax.set_ylabel('pdf(x)')
ax.grid(True)
px=torch.arange(0.5,2,0.01)
ax.set_ylim(0,0.25)
ax.fill_between(px,norm.pdf(px,loc=1,scale=2),alpha=0.5, color='g')
pro=norm(1, 2).cdf(3.5) - norm(1,2).cdf(0)
ax.text(0.2,0.02,round(pro,2), fontsize=20)
plt.show()
```



1.6.6 Gamma Distribution

The gamma distribution term is mostly used as a distribution which is defined as two parameters – shape parameter and inverse scale parameter, having continuous probability distributions. Its importance is largely due to its relation to exponential and normal distributions.

Gamma distributions have two free parameters, named as alpha () and beta (), where;

- = Shape parameter
- = Rate parameter (the reciprocal of the scale parameter)

The scale parameter is used only to scale the distribution. This can be understood by remarking that wherever the random variable x appears in the probability density, then it is divided by . Since the scale parameter provides the dimensional data, it is seldom useful to work with the "standard" gamma distribution, i.e., with = 1.

Gamma function:

The gamma function [10], shown by $\Gamma(x)$, is an extension of the factorial function to real (and complex) numbers. Specifically, if $n \in \{1, 2, 3, \ldots\}$, then

$$\Gamma(n) = (n-1)!$$

Probability Density Function

$$f(x) = \frac{1}{\Gamma(\alpha)}\beta^{\alpha}x^{\alpha-1}e^{-\beta x}$$

Mean

$$\begin{aligned}\mu = E[X] &= \int_{-\infty}^{\infty} x f(x) dx \\ &= \int_{0}^{\infty} x \frac{1}{\Gamma(\alpha)}\beta^{\alpha}x^{\alpha-1}e^{-\beta x} dx \\ &= \frac{\alpha}{\beta}\end{aligned}$$

Variance

$$\begin{aligned}\sigma^2 = \mathrm{Var}[X] &= E\left[(X-\mu)^2\right] \\ &= E\left[X^2\right] - (E[X])^2 \\ &= \cdots = \frac{\alpha}{\beta^2}\end{aligned}$$

1.6.7 Chi-squared Distribution

One Parameter:

- degrees of freedom: $n \geq 1$ (n is an integer) $X \sim \chi^2(n)$ is defined as $\Gamma\left(\frac{n}{2}, \frac{1}{2}\right)$

mean

$$\mu = E[X] = n$$

variance

$$\sigma^2 = \text{Var}[X] = 2n$$

1.6.8 T-distribution

Let $Z \sim N(0,1)$ and $W \sim \chi^2(n)$ be independent random variables. Define

$$T = \frac{Z}{\sqrt{W/n}}$$

the t-distribution Write $X \sim t(n)$ One Parameter: degrees of freedom: $n \geq 1$ (n is an integer) The pdf:

$$f(x) = \frac{\Gamma\left(\frac{n+1}{2}\right)}{\sqrt{\pi n}\Gamma\left(\frac{n}{2}\right)}\left(1 + \frac{x^2}{n}\right)^{-(n+1)/2} \\ -\infty < x < \infty$$

1.7 Joint Distributions

1.7.1 Discrete Definition

Given two discrete random variables, X and Y , p(x, y) = P(X = x, Y = y) is the joint probability mass function for X and Y .

Important property X and Y are independent random variables if P(X = x, Y = y) = P(X = x)P(Y = y) for all possible values of x and y.

$f(x,y) = P(X = x\, and\, Y = y) = P(X = x, Y = y)$

- Sum of all marginal probabilities is equal to 1. (P(y=0) + P(y=100) = P(y = 200) = 1)
- Sum of all joint probabilities is equal to 1.

Marginal Probabilities

Example

An insurance agency services customers who have both a homeowner's policy and an automobile policy. For each type of policy, a deductible amount must be specified. For an automobile policy, the choices are $100 or $250 and for the homeowner's policy, the choices are $0, $100, or $200.

`Recall` Two events are independent if P(A and B) = P(A)P(B) for all possible values of A and B.

P(x=100,y=100) = .1
P(x=100) p(y=200) = (.5)(.25) =.125

X and y are not independent.

1.7.2 Continuous Definition

Definition: If X and Y are continuous random variables, then f(x, y) is the joint probability density function for X and Y if $P(a \leq X \leq b, c \leq Y \leq d)=\int_a^b \int_c^d f(x, y) d x d y$ for all possible $a, b, c$, and $d$ Important property: $X$ and $Y$ are independent random variables if $f(x, y)=f(x) f(y)$ for all possible values of $x$ and $y$.

Example

Example: Suppose a room is lit with two light bulbs. Let X_1 be the lifetime of the first bulb and X_2 be the lifetime of the second bulb. Suppose $X_1 \sim Exp\left(\lambda_1=1/2000\right)$ and $X_2 \sim Exp\left(\lambda_2=1/3000\right)$. If we assume the lifetimes of the light bulbs are independent of each other, find the probability that the room is dark after 4000 hours.

$E\left(X_1\right)=\frac{1}{\lambda_1}=2000\text{hrs}, E\left(X_2\right)=\frac{1}{\lambda_2}=3000\text{hrs}.$

Light bulbs function independently,so

$$\begin{aligned} P\left(X_1 \leq 4000, X_2 \leq 4000\right) &= P\left(X_1 \leq 4000\right) P\left(X_2 \leq 4000\right) \\ &= \left(\int_0^{4000} \lambda_1 e^{-\lambda_1 x_1} d x_1\right)\left(\int_0^{4000} \lambda_2 e^{-\lambda_2 x_2} d x_2\right) \\ &= \left.\left(-e^{-\lambda_1 x_1}\right)\right|_0^{4000}\left.\left(-e^{-\lambda_2 x_2}\right)\right|_0^{4000} \\ = \left(1-e^{-4000/2000}\right)\left(1-e^{-4000/3000}\right) &= \left(1-e^{-2}\right)\left(1-e^{-4/3}\right) \\ &\simeq .6368 \end{aligned}$$

1.8 Covariance and Correlation

1.8.1 Covariance

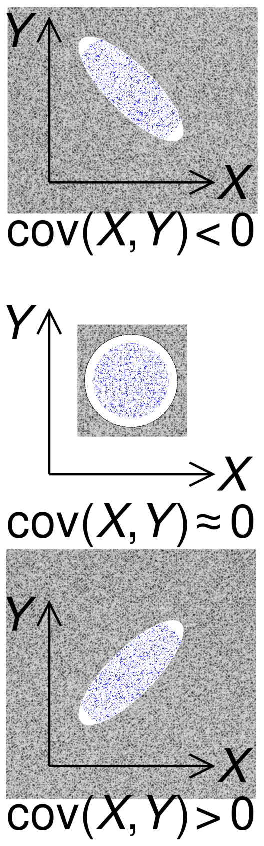
The covariance between two rv's, X and Y, is defined as

$\operatorname{Cov}(X, Y)=E[(X-E(X))(Y-E(Y))]=E[(X-\mu_x)(Y-\mu_y)]$

$$\operatorname{Cov}(X, Y)=\left\{\begin{array}{c} \sum_x \sum_y\left(x-\mu_X\right)\left(y-\mu_Y\right) P(X=x, Y=y) \\ \int_{-\infty}^{\infty} \int_{-\infty}^{\infty}\left(x-\mu_X\right)\left(y-\mu_Y\right) f(x, y) d x d y \end{array}\right.$$

The covariance depends on both the set of possible pairs and the probabilities for those pairs.

Image from Wikipedia



- If both variables tend to deviate in the same direction (both go above their means or below their means at the same time), then the covariance will be positive.
- If the opposite is true, the covariance will be negative.
- If X and Y are not strongly (linearly) related, the covariance will be near 0.

Computational formula for Covariance

$\mathrm{Cov}(X, Y) = E[XY] - E[X]E[Y]$

1.8.2 Correlation Coefficient

The correlation Coefficient of X and Y , denoted by Cor(X, Y) Represented by the Greek letter ‘” (rho)

$Cor(X, Y) = \rho_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y}$

It represents a “scaled” covariance. The correlation is always between -1 and 1.

1.8.3 Transformations of Distributions

Discrete Distributions

Suppose that   (,) What is the distribution of Y = n-X?

$f(x) = P(X = x) = \binom{n}{x} p^x (1-p)^{n-x} \cdot I_{\{1,2,3,\ldots\}}(x)$

Just do it:

$$P(Y = y) = P(n - X = y) = P(X = n - y)$$
$$= \binom{n}{n-y} p^x (1-p)^{n-(n-y)} \cdot I_{\{0,1,2,3,\ldots\}}(n-y)$$
$$= \binom{n}{y} p^n - y(1-p)^y \cdot I_{\{0,1,2,3,\ldots\}}(y) = (,)$$

Continuous Distributions

Invertible functions

In the most general sense, are functions that "reverse" each other. For example, if f takes a to b, then the inverse, f^{-1} must take b to a. a function is invertible only if each input has a unique output. That is, each output is paired with exactly one input. That way, when the mapping is reversed, it will still be a function!

For X discrete or continuous, the cumulative distribution function (cdf) Is denoted by F(x) and is defined by

$F(X) = P(X < x)$

1.9 Estimators and Sampling Distributions

We have learned many different distributions for random variables and all of those distributions had parameters: the numbers that you provide as input when you define a random variable.

What if we don't know the values of the parameters. What if instead of knowing the random variables, we have a lot of examples of data generated with the same underlying distribution? In this chapter we are going to learn formal ways of estimating parameters from data.

These ideas are critical for artificial intelligence. Almost all modern machine learning algorithms work like this

1. specify a probabilistic model that has parameters.
2. Learn the value of those parameters from data.

Estimate the model parameters.

- Maximum Likelihood Estimation (MLE)
- Maximum A Posteriori (MAP).

Both of these schools of thought assume that your data are independent and identically distributed (IID) samples.

1.9.1 Random Sample

A collection of random variables is independent and identically distributed if each random variable has the same probability distribution as the others and all are mutually independent.

Random Sample = $X_1, X_2, X_3, ..., X_n$

Suppose that $X_1, X_2, X_3, ..., X_n$ is a random sample from the gamma distribution with parameters $alpha$ and β.

$$X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \Gamma(\alpha, \beta)$$

E.g

A good example is a succession of throws of a fair coin: The coin has no memory, so all the throws are “independent”. And every throw is 50:50 (heads:tails), so the coin is and stays fair - the distribution from which every throw is drawn, so to speak, is and stays the same: “identically distributed”.

Independent and identically distributed random variables (IID)

Random Sample == IID

Note: What are biased and unbiased estimators? A biased estimator is one that deviates from the true population value. An unbiased estimator is one that does not deviate from the true population parameter.

1.9.2 Parameters

Before we dive into parameter estimation, first let’s revisit the concept of parameters. Given a model, the parameters are the numbers that yield the actual distribution.

- In the case of a Bernoulli random variable, the single parameter was the value p.
- In the case of a Uniform random variable, the parameters are the a and b values that define the min and max value.

we are going to use the notation θ to be a vector of all the parameters.

Distribution	Parameters
Bernoulli(p)	$\theta = p$
Poisson()	$\theta = \lambda$
Uniform(a,b)	$\theta = (a, b)$
Normal	$\theta = (\mu, \sigma)$
$Y = wX + b$	$\theta = (w, b)$

1.9.3 Sampling Distributions

θ will denote a generic parameter.

E.g

$\theta = \mu, \theta = p, \theta = \lambda, \theta = (\alpha, \beta)$

Estimator

$\hat{\theta}$ = a Random variable,

$\hat{\theta} = \bar{X}$

Estimate

$\hat{\theta}$ = a observed number

$\hat{\theta} = \bar{x} = 42.5$

- We want our estimator of to be correct "on average.
- $\bar{X}$ is a random variable with its owo distribution and its own mean or expected value.

We would like sample mean [] $=$ to be close to the true mean or population mean .

Important:

- If this is true, we say that¯is an unbiased estimator of μ.
- In general, $\bar{\theta}$ is an unbiased estimator of θ. if $E[\bar{\theta}] = \theta$.

That's is really good thing.

Mean

Let X1, X2, . . . , Xn be random sample from any distribution with mean μ.

That is $E[X_i] = \mu$ for i = 1,2,3,. . . , n. Then

$$E[\bar{X}] = E\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n}\sum_{i=1}^{n} E\left[X_i\right]$$
$$= \frac{1}{n}\sum_{\mathrm{i=1}}^{\mathrm{n}} \mu = \frac{1}{\mathrm{n}}(\mu + \mu + \cdots + \mu) = \frac{1}{\mathrm{n}}\mathrm{n}\mu = \mu$$

We have shown that, no matter what distribution we are working with, if the mean is μ , $\bar{X}$ is an unbiased estimator for μ.

Attention: We have shown that, no matter what distribution we are working with, if the mean μ is , $\bar{X}$ is an unbiased estimator for μ .

Let X1, X2, . . . , Xn be random sample from any (rate = λ)

Let $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ is the sample mean. We know, for the exponential distribution, that $E[X_i] = \frac{1}{\lambda}$.

Then $E[\bar{X}] = \frac{1}{\lambda}$

Variance

Let X1, X2, …, Xn be random sample from any distribution with mean μ and variance σ^2.

- We already know that $\bar{X}$ is an unbiased estimator for μ .
- What can we say about the variance of $\bar{X}$?

$Var[\bar{X}] = Var\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] == \frac{1}{n^2}Var\left[\sum_{i=1}^{n} X_i\right] == \frac{1}{n^2}\sum_{i=1}^{n} Var\left[X_i\right]$

$= \frac{1}{n^2}\sum_{i=1}^{n}\sigma^2 = \frac{1}{n^2}n\sigma^2 = \frac{\sigma^2}{\mathrm{n}}$

1.10 Moments Generating Functions

We are still, believe it or not, trying to estimate things from a larger population based on a sample. For example, sample mean, or maybe the sum of the values in the sample etc. Any function of your data is known as a statistic. And we're going to use them to estimate other things. And in order to figure out how well we're doing, we're going to **need to know often the distributions of some of these statistics**.

1.10.1 Distributions of sums

A lot of them depend on sums, so we're going to start out by talking about the distribution of sums of random variables.

Suppose That,

$$X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} Bernoulli(p)$$
$$\text{What is the distribution of } Y = \sum_{i=1}^{n} X_i?$$
$$\text{Sum of Bernoulli rv is equal to bin(n,p)}$$
$$Y = \sum_{i=1}^{n} X_i \sim bin(n, p)$$

Each X_i take value success (P) and failure (1-P). So summing all X_i is equal to sum of all success gives the value of Y. Which is binomial distribution.

Caution: Not all random variables are so easily interpreted by methods of Distributions of sums. So we need a tool.

1.10.2 Moment generating functions

The moments generating functions are the functions that generate the moments of a random variable. The expected values $E(X), E\left(X^2\right), E\left(X^3\right), \ldots E\left(X^r\right)$ are called moments.

- Mean $\mu = E(X)$
- Variance $\sigma^2 = Var(X) = E\left(X^2\right) - \mu^2$

which are functions of moments. moment-generating functions can sometimes make finding the mean and variance of a random variable simpler.

Let X be a random variable. It's moment generating function (mgf) is denoted and defined as

Continuous Random Variables
$M_X(t) = E\left[e^{tX}\right] = \int_{-\infty}^{\infty} e^{tx} f_X(x) dx$

Discrete Random Variables
$M_X(t) = E\left[e^{tX}\right] = \sum_x e^{tx} f_x(x)$

where $f_X(x)$ is the distribution of X.

Properties

- Moment generating functions also **uniquely identify distributions**.

1.10.3 MGT of Famous Distributions

Bernoulli(p)

$$\begin{aligned} M_X(t) = E\left[e^{tX}\right] &= \sum_x e^{tx} f_X(x) = \sum_x e^{tx} P(X = x) \\ &= e^{t\cdot 0} P(X = 0) + e^{t\cdot 1} P(X = 1) \\ &= 1 \cdot (1 - p) + e^t \cdot p \\ &= 1 - p + pe^t \end{aligned}$$

Binomial(n,p)

$X \sim bin(n, p)$

$$M_x(t) = \sum_{x=0}^{n} e^{tx} \binom{n}{x} p^x (1 - p)^{n-x}$$
$$M_x(t) = \sum_{x=0}^{n} e^{tx} \binom{n}{x} (pe^t)^x (1 - p)^{n-x}$$

Binomial Theorem
$(a + b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$

$$M_X(t) = (1 - p + pe^t)^n$$

1.10.4 Finding Distributions

A moment-generating function uniquely determines the probability distribution of a random variable. if two random variables have the same moment-generating function, then they must have the same probability distribution.

Important feature of MGF

Some distribution with $X_1, X_2, \ldots, X_n$ iid and $Y = \sum_{i=1}^{n} X_i$.
$M_Y(t) = [M_{X_1}(t)]^n$

We have just seen that the moment generating function of the sum. Is the moment generating function of one of them raised to the nth power.

Key points

- sum of n iid Bernoulli(p) random variables is bin(n, p)
- sum of n iid exp(rate =lambda) random variables is Gamma(n, lambda)
- sum of m iid bin(n,p) is bin(nm,p)
- sum of n iid Gamma(alpha, beta) is Gamma(n alpha, beta)
- sum of n iid $N\left(\mu, \sigma^2\right) isN\left(n\mu, n\sigma^2\right)$.
- sum of $n$ independent normal random variable with $\mathrm{X_i} \sim \mathrm{N}\left(\mu_{\mathrm{i}}, \sigma_{\mathrm{i}}^2\right)$ is $\mathrm{N}\left(\sum_{\mathrm{i}=1}^{\mathrm{n}} \mu_{\mathrm{i}}, \sum_{\mathrm{i}=1}^{\mathrm{n}} \sigma_{\mathrm{i}}^2\right)$

1.10.5 Method of Moments Estimators(MMEs)

Method of moments means you set sample moments equal to population/theoretical moments.

It totally makes sense if you're trying to estimate the mean or average out there in the entire population. That you should use the sample mean or sample average of the values in the sample, but what about parameters with not such an obvious interpretation?

Idea: Equate population and sample moments and solve for the unknown parameters.

Suppose that $X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \Gamma(\alpha, \beta)$
How can we estimate ?
We could estimate the true mean α/β with the sample mean $\bar{X}$, but we still can't get at if we don't know .

Attention: Recall that the "moments" of a distribution are defined as [], [], [ ], … These are distribution or "population" moments

- $\mu = E[X]$ is a probability weighted average of the values in the population.
- $\bar{X}$ is the average of the values in the sample.

It was natural for us to think about estimating $mu$ with the average in our sample.

- $\mathrm{E}\left[\mathrm{X}^2\right]$ is a probability weighted average of the squares of the values in the population.

It is intuitively nice to estimate it with the average of the squared values in the sample:

$$\frac{1}{n}\sum_{i=1}^{n} X_i^2$$

$$\text{The kth population moments:}$$
$$\mu_\mathrm{k} = \mathrm{E}\left[\mathrm{X}^\mathrm{k}\right] \quad \mathrm{k} = 1,2,3,\ldots$$
$$\text{The kth population moments:}$$
$$\mu_\mathrm{k} = \mathrm{E}\left[X^\mathrm{k}\right] \quad \mathrm{k} = 1,2,3,\ldots$$
$$\text{The kth sample moments:}$$
$$M_k = \frac{1}{n}\sum_{i=1}^{n} X_i^k \quad k = 1,2,3,\ldots$$

Eg

$$X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \exp(\text{ rate } = \lambda)$$
$$\text{First population moment:}$$
$$\mu_1 = \mu = \mathrm{E}[\mathrm{X}] = \frac{1}{\lambda}$$
$$\text{First sample moment:}$$
$$M_1 = \frac{1}{n}\sum_{i=1}^{n} X_i = \bar{X}$$
$$\text{Equate:} \frac{1}{\lambda} = \bar{x}$$
$$\text{Solve for the unknown parameter...} \lambda = \frac{1}{\bar{x}}$$
$$\text{The MME is } \hat{\lambda} = \frac{1}{\bar{x}}$$

1.11 Maximum Likelihood Estimation

1.11.1 Idea

Choose the value in the parameter space that makes the observed data "most likely".

Suppose that we flip a biased coin which has the probability of getting "Heads" as either 0.2, 0.3, or 0.8. Suppose that we flip the coin 20 times and see the results:

Sample Space: H, H, T, H, H, H, H, T, H, H, H, H, H, T, H, H, H, H, H , H

`Which of 0.2, 0.3, or 0.8 seems "most likely"?`

What if we only flip the coin twice? For i=1,2, let $X_i = \begin{cases} 1 & \text{if we get "Heads" on the ith flip} \\ 0, & \text{if we get "Tails" on the ith flip} \end{cases}$

Let p=P("Heads" on any one flip) **Then** X1, X2 Bernoulli(P) iid **where** $\{\ .\ ,\ .\ ,\ .\ \}$

Joint pmf Due to independence of the variables, we can write the joint pmf as

$$\begin{aligned} f\left(x_1, x_2\right) &= P\left(X_1=x_1, X_2=x_2\right) \\ &= P\left(X_1=x_1\right) \cdot P\left(X_2=x_2\right) \\ = p^{x_1}(1-p)^{1-x_1} \mathrm{I}_{\{0,1\}}\left(\mathrm{x}_1\right) &\cdot \mathrm{p}^{\mathrm{x}_2}(1-p)^{1-\mathrm{x}_2} \mathrm{I}_{\{0,1\}}\left(\mathrm{x}_2\right) \\ \text{if p=0.2 and (0,0)} &= 0.2^0 \times(1-0.2)^0 \times 0.2^0 \times(1-0.2)^0=0.64 \\ \text{if p=0.8 and (0,1)} &= 0.8^0 \times(1-0.8)^0 \times 0.8^1 \times(1-0.8)^1=0.16 \end{aligned}$$

Tabulated values of the joint pmf

- When we observe the data to be (0,0) i.e. ("Tails", "Tails"), the value of p that gives the highest joint probability `(0.64) is 0.2`.
- When we observe the data to be (0,1) or (1,0) i.e. ("Tails", "Heads") or ("Heads", "Tails"), the value of p that gives the highest joint probability `(0.21) is 0.3`.
- When we observe the data to be (1,1) i.e. ("Heads", "Heads"), the value of p that gives the highest joint probability `(0.64) is 0.8`.

The maximum likelihood estimator for p is:

$$\widehat{p}=\begin{cases} 0.2 & \text{, if } \left(x_1, x_2\right)=(0,0) \\ 0.3 & \text{, if } \left(x_1, x_2\right)=(0,1) \text{ or } (1,0) \\ 0.8 & \text{, if } \left(x_1, x_2\right)=(1,1) \end{cases}$$

1.11.2 Introduction

Given data $X_1, X_2 \ldots X_n$, a random sample (iid) from a distribution with unknown parameter , we want to find the value of in the parameter space that maximizes our probability of observing that data.

For Discrete…

If $X_1, X_2 \ldots X_n$ are discrete, we can look at $P\left(X_1=x_1, X_2=x_2, \ldots, X_n=x_n\right)$ as a function of , and find the that maximizes it. This is the joint pmf for $X_1, X_2 \ldots X_n$.

For Continuous…

If $X_1, X_2 \ldots X_n$ are continuous is to maximize the **joint pdf** with respect to .

For Discrete…

If $X_1, X_2 \ldots X_n$ are discrete, we can look at $P\left(X_1=x_1, X_2=x_2, \ldots, X_n=x_n\right)$ as a function of , and find the that maximizes it. This is the joint pmf for $X_1, X_2 \ldots X_n$.

For Continuous…

If $X_1, X_2 \ldots X_n$ are continuous is to maximize the **joint pdf** with respect to .

The pmf/pdf for any one of is denoted by `f(x)`. We will emphasize the dependence of f on a parameter by writing it as

$$f(x)=f(x;\theta)$$

The joint pmf/pdf for all n of them is

$$f\left(x_1, x_2, \ldots, x_n ; \theta\right)=\prod_{i=1}^n f\left(x_i ; \boldsymbol{\theta}\right)$$

$$f(\vec{x} ; \boldsymbol{\theta})=\prod_{i=1}^n f\left(x_i ; \boldsymbol{\theta}\right)$$

- The data (the x's) are fixed.
- Think of the x's as fixed and the joint pdf as a function of .

Given the joint PDF, the data, the Xs are fixed, and we think of it as a function of theta and we want to find the value of theta that maximizes the joint probability density function or probability mass function.

1.11.3 Likelihood function

If we think of this as a function of theta, and the x's as fixed, we're going to rename the joint PDF. We're going to call it a likelihood function and write it as a capital L of theta L().

Attention: Because I can multiply or divide my likelihood by a constant and not change where the maximum occurs, then we can actually define the likelihood to be anything proportional to the joint pdf. So we can throw out multiplicative constants, including multiplicative constants that involve Xs.

1.11.4 Bernoulli distribution

$$X_1, X_2, \ldots, X_n \stackrel{\text{iid}}{\sim} \text{ Bernoulli }(p)$$

The pmf for one of them is $f(x;p)=p^x(1-p)^{1-x}I_{\{0,1\}}(x)$
The joint pmf for all of them is

$$f(\vec{x} ; p)=\prod_{i=1}^n f\left(x_i ; p\right)=\prod_{i=1}^n p^{x_i}(1-p)^{1-x_i} I_{\{0,1\}}\left(x_i\right)$$

The joint probability mass function we'll get by multiplying the individual ones together, because these guys are IID independent and identically distributed. Now, fix the Xs. Those are stuck, fixed, not moving, and *think of this as a function of p*. The values of p that are allowed, the parameter space for this model, are all values of p between 0 and 1.

For example I have p^X_1 times p^X_2 times p^X_3 and that's going to be p to the sum of the Xs, and I've got 1 minus p^1 minus X_1, 1 minus p^1 minus X_2. If I add up those exponents, I'm going to get an exponent of n minus the sum of the Xs, and I do have a product of indicators.

$$=p^{\sum_{i=1}^n x_i}(1-p)^{n-\sum_{i=1}^n x_i} \prod_{i=1}^n I_{\{0,1\}}\left(x_i\right)$$

Drop the indicator stuff, so that is a multiplicative constant which is constant with respect to p. I think I'm going to drop it. Why not make it simpler?

$$\text{A likelihood is } L(p)=p^{\sum_{i=1}^n x_i}(1-p)^{n-\sum_{i=1}^n x_i}$$

Log-likelihood

It is almost always easier to maximize the log-likelihood function due to properties of Logarithms.

$$ln(uv) = ln(u) + ln(v) \text{ and } ln(n)^V = v \times ln(n)$$

Important: The log function is an increasing function. So the log of the likelihood is going to have different values than the likelihood, but because log is increasing, this is not going to mess up the location of the maximum.

$$L(p) = \log\left(\prod_{i=1}^{n} p^{x_i}(1-p)^{1-x_i} I_{\{0,1\}}(x_i)\right)$$

$$\ell(p) = \sum_{i=1}^{n} x_i \ln p + \left(n - \sum_{i=1}^{n} x_i\right)\ln(1-p)$$

```
I want to maximize it with respect to p, so I'm going to take a derivative with respect
to p and set it equal to 0.
```

$$\frac{\partial}{\partial p} l(p) = \frac{\sum_{i=1}^{n} x_i}{p} - \frac{n - \sum_{i=1}^{n} x_i}{1-p} \overset{\text{set}}{=} 0$$

$$p(1-p)\left[\frac{\sum_{i=1}^{n} x_i}{p} - \frac{n - \sum_{i=1}^{n} x_i}{1-p}\right] = p(1-p)\cdot 0$$

$$(1-p)\sum_{i=1}^{n} x_i - p\left(n - \sum_{i=1}^{n} x_i\right) = 0$$

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

This is our coin example again. But we have n flips, and we have the Bernoulli's ones and zeros for heads and tails, and the value of p is unknown, it's somewhere between 0 and 1. We're no longer restricted to 0.2, 0.3, and 0.8. The maximum likelihood estimator, is the sample mean of the ones and zeros. If you add up the ones and zeros, and divide by n, you're really computing the proportion of ones in your sample. You're really computing the proportion of times you see heads in your sample. This maximum likelihood estimator, at least, in this case, makes a lot of sense.

$$\hat{p} = \frac{\sum_{i=1}^{n} X_i}{n} = \bar{X}$$

Q/A

Is maximum likelihood estimator Bernoulli unbiased?

the maximum likelihood estimator of is a *biased estimator*. Recall that if X_i is a Bernoulli random variable with parameter P, then $E[X_i] = p$.

$$E(\hat{p}) = E\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) = \frac{1}{n}\sum_{i=1}^{n} E(X_i) = \frac{1}{n}\sum_{i=1}^{n} p = \frac{1}{n}(np) = p$$

1.11.5 Exponential distribution

$$X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \text{ Exponential } (rate = \lambda)$$

The pmf for one of them is $f(x; p) = \lambda e^{-\lambda x} I_{(0,\infty)}(x)$
The joint pmf for all of them is

$$f(\vec{x}; \lambda) = \prod_{i=1}^{n} f(x_i; \lambda) = \prod_{i=1}^{n} \lambda e^{-\lambda x_i} I_{(0,\infty)}(x_i)$$

$$f(\vec{x}; p) = \lambda^n e^{-\lambda \sum_{i=1}^{n} x_i} \prod_{i=1}^{n} I_{(0,\infty)}(x_i)$$

The parameter space, the Lambdas that are allowed are everything from 0 to infinity.
At this point, I can drop constants of proportionality. Again, I'm going to drop that indicator.

$$\text{A likelihood is} = L(\lambda) = \lambda^n e^{-\lambda \sum_{i=1}^{n} x_i}$$

$$\text{The log-likelihood is} = \ell(\lambda) = n \ln \lambda - \lambda \sum_{i=1}^{n} x_i$$

`Our goal is to maximize this as a function of Lambda.`

$$\frac{\partial}{\partial \lambda} \ell(\lambda) = \frac{n}{\lambda} - \sum_{i=1}^{n} x_i \overset{\text{set}}{=} 0$$

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

I want to make everything capital, and throw a hat on it. Here is our first continuous maximum likelihood estimator for Theta or Lambda.

The maximum likelihood estimator for λ is

$$\hat{\lambda} = \frac{n}{\sum_{i=1}^{n} X_i} = \frac{1}{\bar{X}}$$

Warning: Same as method of moments. Biased!

This is exactly what we got with method of moments. Because if Lambda is the rate of this distribution, the true distribution mean is 1 over Lambda. If you equate that to the sample mean x bar and solve for Lambda, in the method of moments case, we got 1 over x bar. We weren't that happy about it because it was a biased estimator. I'm trying to convince you that MLEs are everything. But they're not unbiased.

1.11.6 Normal distribution

`MLEs for Multiple and Support Parameters`

We're going to to consider two cases

- One is when theta is higher dimensional, so theta might be the vector of mu and sigma squared.
- Other cases when the parameter is in the indicator.

$$X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} N(\mu, \sigma^2)$$

The pdf for one of them is $\mathrm{f}\left(\mathrm{x}; \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{-\frac{1}{2\sigma^2}(\mathrm{x}-\mu)^2}$
The joint pdf for all of them is

$$f(\vec{x}; \mu, \sigma^2) = \prod_{i=1}^{n} f\left(x_i; \mu, \sigma^2\right) = \left(2\pi\sigma^2\right)^{-\mathrm{n}/2} \mathrm{e}^{-\frac{1}{2\sigma^2}\sum_{\mathrm{i}=1}^{\mathrm{n}}(\mathrm{x_i}-\mu)^2}$$

The parameter space : $-\infty < \mu < \infty, \quad \sigma^2 > 0$

$$\text{A likelihood is } \mathrm{L}\left(\mu, \sigma^2\right) = \left(2\pi\sigma^2\right)^{-\mathrm{n}/2} \mathrm{e}^{-\frac{1}{2\sigma^2}\sum_{\mathrm{i}=1}^{\mathrm{n}}(\mathrm{x_i}-\mu)^2}$$
$$\text{The log-likelihood is } \ell\left(\mu, \sigma^2\right) = -\frac{\mathrm{n}}{2}\ln\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(\mathrm{x_i}-\mu)^2$$
$$\ell\left(\mu, \sigma^2\right) = -\frac{\mathrm{n}}{2}\ln\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{\mathrm{i}=1}^{\mathrm{n}}(\mathrm{x_i}-\mu)^2$$
$$\frac{\partial}{\partial\mu}\ell\left(\mu, \sigma^2\right) \overset{\text{set}}{=} 0$$
$$\frac{\partial}{\partial\sigma^2}\ell\left(\mu, \sigma^2\right) \overset{\text{set}}{=} 0$$

Solve for and simultaneously

1.11.7 The Invariance Property

1.11.8 Evaluation

Comparing the quality of different estimators

Variance, MSE, and Bias

Mean Squared Error

Let $\hat{\theta}$ be an estimator of a parameter θ. The mean squared error of $\hat{\theta}$ is denoted and defined by

$$MSE(\hat{\theta}) = E[(\hat{\theta} - \theta)^2]$$

Note: If $\hat{\theta}$ is an unbiased estimator of θ, its mean squared error is simply the variance of θ

Bias

The bias of $\hat{\theta}$ is denoted and defined by

$$B(\hat{\theta}) = E[\hat{\theta}] - \theta$$

An unbiased estimator has a bias of zero.

$$MSE(\widehat{\theta}) = E\left[(\widehat{\theta} - \theta)^2\right]$$
$$= E\left[(\widehat{\theta} - E[\hat{\theta}] + E[\hat{\theta}] - \theta)^2\right]$$
$$= E\left[((\hat{\theta} - E[\hat{\theta}]) + B[\hat{\theta}])^2\right]$$

$$MSE(\hat{\theta}) = Var[\hat{\theta}] + (B[\hat{\theta}])^2$$

Practise

Let $X_1, X_2, \ldots, X_n$ be a random sample from the Poisson distribution with parameter $\lambda > 0$. Let $\hat{\theta}$ be the MLE for λ. What is the mean-squared error of $\widehat{\lambda}$ as an estimator of λ ?

$$x \sim \text{ poisson (} \lambda \text{) and MLE of } \lambda = \bar{X}$$
$$\text{Proof}$$
$$E(\bar{X}) = E[\sum_{i=1}^{n} X_i] = \frac{1}{n}E\left[\sum_{n=1}^{n} x_i\right]$$
$$E(\bar{X}) = \frac{n}{n}E\left[X_i\right] = \lambda$$

$$MSE(\hat{x}) = Var(\hat{\lambda}) + (Bias(\lambda))^2$$
$$= Var(\hat{\lambda}) + 0$$
$$= Var(\frac{1}{n}\sum_{i=1}^{n} x_i)$$
$$= \frac{n}{n^2} \times \lambda = \frac{\lambda}{n}$$

1.11.9 MLE Properties

1.12 Confidence Interval

A 95% confidence for the mean is given by (-2.14,3.07).

There’s no probability, where does the probability come in?

it comes in from random sampling. The fact that if you select a random sample and compute your estimator and if you got to do it again or if someone else did it, they’re going to get a different estimator based on the *randomness of the sample*.

- Collect your sample.
- Estimate the parameter.
- Return a confidence interval.

If you did this again, you would not get the same results! Different samples and different confidence intervals.

Multiple samples give multiple confidence intervals. `95% of them will correctly capture  and 5% miss it.`

1.13 Hypothesis Testing

Statistical inference is the process of learning about characteristics of a population based on what is observed in a relatively small sample from that population. A sample will never give us the entire picture though, and we are bound to make incorrect decisions from time to time.

We will learn how to derive and interpret appropriate tests to manage this error and how to evaluate when one test is better than another. we will learn how to construct and perform principled hypothesis tests for a wide range of problems and applications when they are not.

1.13.1 What is Hypothesis

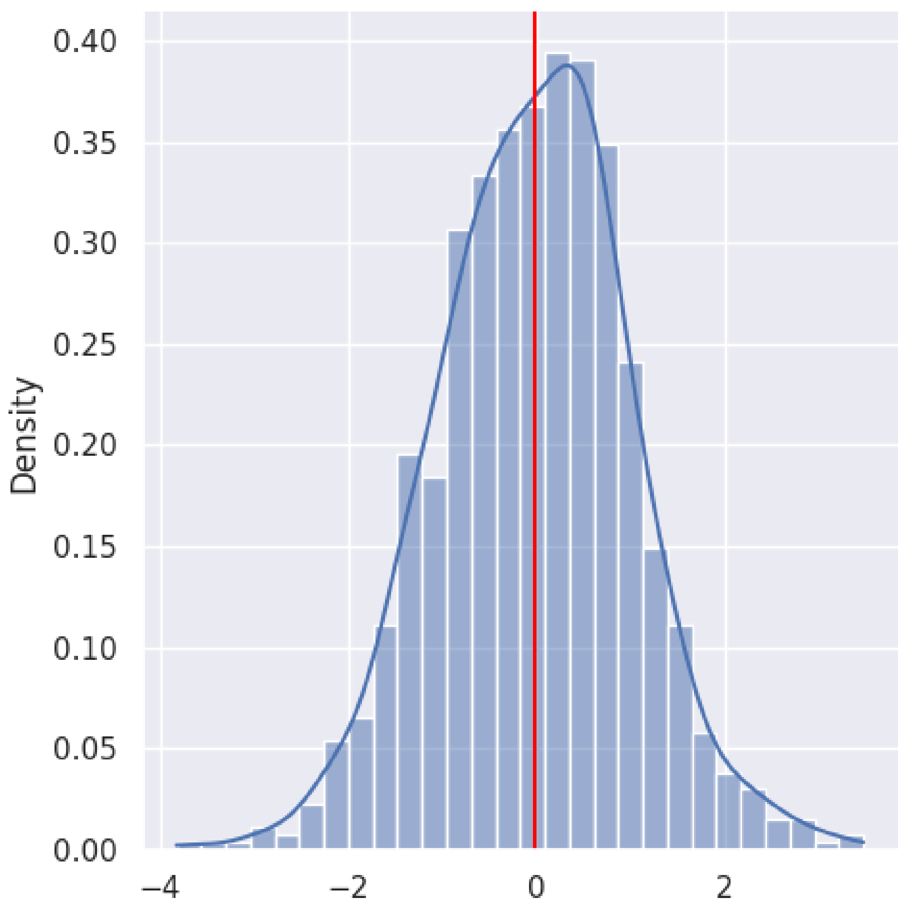
- Hypothesis testing is an act in statistics whereby an analyst tests an assumption regarding a population parameter.
- Hypothesis testing is a formal procedure for investigating our ideas about the world using statistics. It is most often used by scientists to test specific predictions, called hypotheses, that arise from theories.

Note: Due to random samples and randomness in the problem, we can different errors in our hypothesis testing. These errors are called Type I and Type II errors.

1.13.2 Type of hypothesis testing

Let $X_1, X_2, \ldots, X_n$ be a *random sample* from the normal distribution with mean μ and variance σ^2

```python
import torch
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm


sns.set_theme(style="darkgrid")
sample = torch.normal(mean = 0, std = 1, size=(1,1000))

sns.displot(sample[0], kde=True, stat = 'density',)
plt.axvline(torch.mean(sample[0]), color='red', label='mean')
plt.show()
```



Example of random sample after it is observed:

$$2.73, 1.14, 3.98, 2.15, 5, 85, 1.97, 2.54, 2.03$$

Based on what you are seeing, do you believe that the true population mean μ is

$$\mu <= 3 \text{ or } \mu > 3$$
$$\text{The sample mean is } \overline{\mathrm{x}} = 2.799$$

This is below 3 , but can we say that $\mu < 3$?

This seems awfully dependent on the random sample we happened to get! Let's try to work with the most generic random sample of size 8:

$$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8$$

Let $\mathrm{X}_1, \mathrm{X}_2, \ldots, \mathrm{X}_\mathrm{n}$ be a random sample of size n from the $\mathrm{N}\left(\mu, \sigma^2\right)$ distribution.

$$\mathrm{X}_1, \mathrm{X}_2, \ldots, \mathrm{X}_\mathrm{n} \overset{\mathrm{iid}}{\sim} \mathrm{N}\left(\mu, \sigma^2\right)$$

The Sample mean is

$$\bar{x} = \frac{1}{n} \sum_{i=11}^{n} X_i$$

- We're going to tend to think that $\mu < 3$ when $\bar{X}$ is "significantly" smaller than 3.

- We're going to tend to think that $\mu > 3$ when $\bar{X}$ is "significantly" larger than 3.
- We're never going to observe $\bar{X} = 3$, but we may be able to be convinced that $\mu = 3$ if $\bar{X}$ is not too far away.

How do we formalize this stuff, We use hypothesis testing

Notation

$\mathrm{H}_0 : \mu \leq 3$ <- Null hypothesis
$\mathrm{H}_1 : \mu > 3$ Alternate hypothesis

Null hypothesis

The null hypothesis is a hypothesis that is assumed to be true. We denote it with an H_0.

Alternate hypothesis

The alternate hypothesis is what we are out to show. The alternative hypothesis is a hypothesis that we are looking for evidence for or **out to show**. We denote it with an H_1.

Note: Some people use the notation H_a here

Conclusion is either:
Reject H_0 OR Fail to Reject H_0

simple hypothesis

A simple hypothesis is one that completely specifies the distribution. Do you know the exact distribution.

composite hypothesis

You don't know the exact distribution.
Means you know the distribution is normal but you don't know the mean and variance.

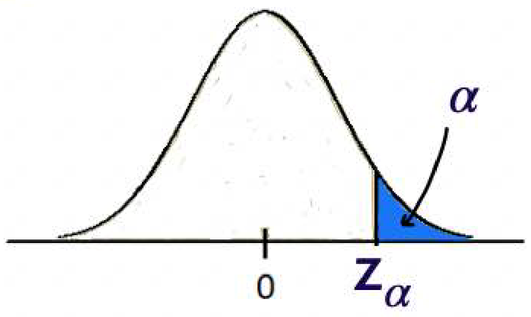
Critical values

Critical values for distributions are numbers that cut off specified areas under pdfs. For the N(0, 1) distribution, we will use the notation z_α to denote the value that cuts off area α to the right as depicted here.

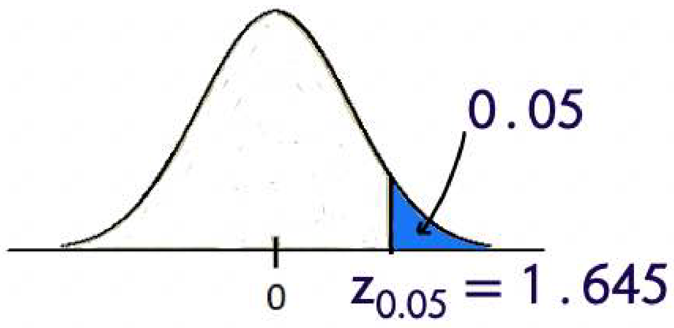
- values that cut off specified areas under pdfs
- for the N(0,1) distribution, we will use the notation

$\boxed{z_\alpha}$

to be the value that cuts off area α to the right



Example:



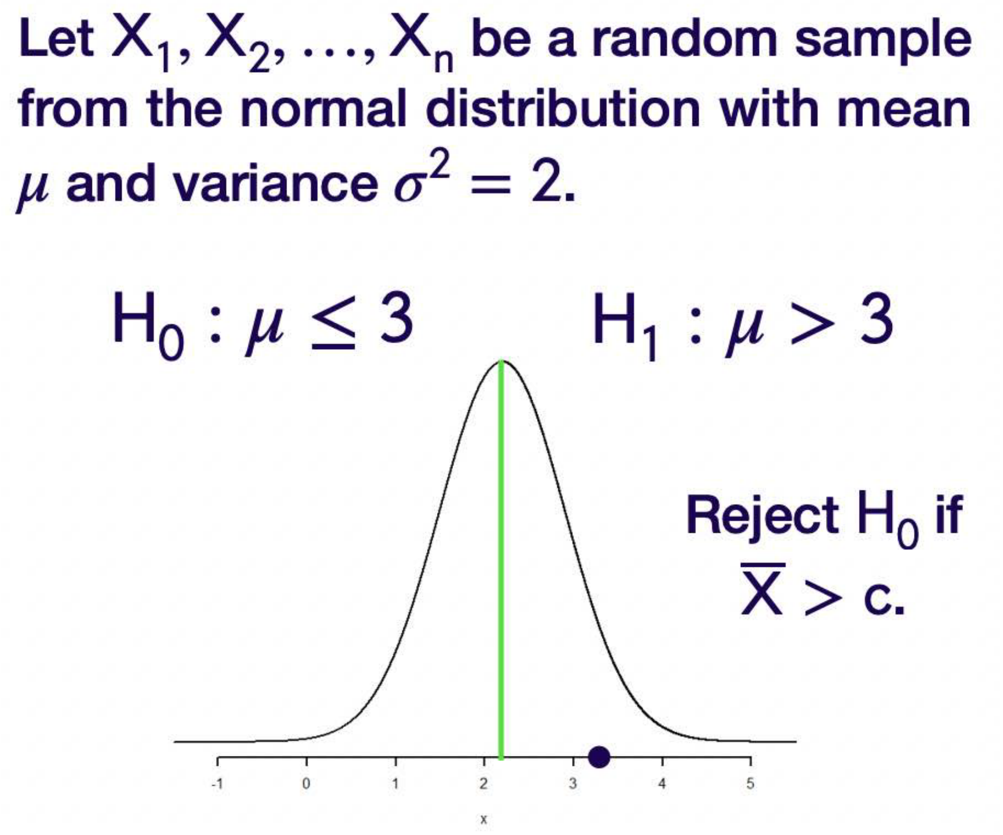
R Code: qnorm(0.95)

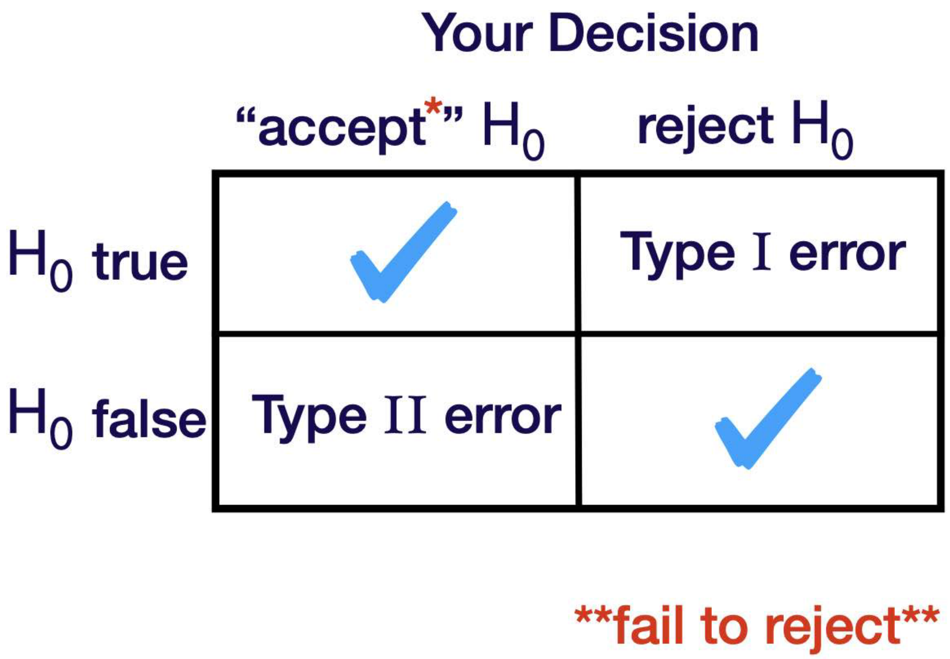
1.13.3 Errors in Hypothesis Testing

Let $X_1, X_2, \ldots, X_n$ be a random sample from the normal distribution with mean μ and variance $\sigma^2 = 2$

$$H_0 : \mu \leq 3 \quad H_1 : \mu > 3$$

Idea: Look at $\bar{X}$ and reject H_0 in favor of H_1 if $\overline{X}$ is "large".
i.e. Look at $\bar{X}$ and reject H_0 in favor of H_1 if $\overline{X} > c$ for some value c.





Your Decision

	"accept*" H_0	reject H_0
H_0 true	✓	Type I error
H_0 false	Type II error	✓

fail to reject

You are a potato chip manufacturer and you want to ensure that the mean amount in 15 ounce bags is at least 15 ounces. $\mathrm{H}_0 : \mu \leq 15 \quad \mathrm{H}_1 : \mu > 15$

Type I Error

The true mean is ≤ 15 but you concluded i was > 15. You are going to save some money because you won't be adding chips but you are risking a lawsuit!

Type II Error

The true mean is > 15 but you concluded it was ≤ 15 . You are going to be spending money increasing the amount of chips when you didn't have to.

1.13.4 Developing a Test

Let $X_1, X_2, \ldots, X_n$ be a random sample from the normal distribution with mean μ and known variance σ^2.

Consider testing the simple versus simple hypotheses

$$H_0 : \mu = 5$$
$$H_1 : \mu = 3$$

level of significance

Let $\alpha = P$ (Type I Error)
$= P($ Reject H_0 when it's true $)$
$= P($ Reject H_0 when $\mu = 5)$

α is called the level of significance of the test. It is also sometimes referred to as the size of the test.

$$\begin{aligned} \alpha &= \max P(\text{ Type I Error }) \\ &= \max_{\mu \in H_0} P(\text{ Reject } H_0; \mu) \\ \beta &= \max P(\text{ Type II Error }) \\ &= \max_{\mu \in H_1} P(\text{ Fail to Reject } H_0; \mu) \end{aligned}$$

Power of the test

$1 - \beta$ is known as the power of the test

$$\begin{aligned} 1 - \beta &= 1 - \max_{\mu \in H_1} P(\text{ Fail to Reject } H_0; \mu) \\ &= \min_{\mu \in H_1} (1 - P(\text{ Fail to Reject } H_0; \mu)) \\ &= \min_{\mu \in H_1} P(\text{ Reject } H_0; \mu) \quad \text{High power is good!} \end{aligned}$$

Step One

Choose an estimator for .

$$\widehat{\mu} = \bar{X}$$

Step Two

Choose a test statistic or Give the "form" of the test.

- We are looking for evidence that H_1 is true.
- The $N\left(3, \sigma^2\right)$ distribution takes on values from $-\infty$ to ∞.
- $\overline{X} \sim N\left(\mu, \sigma^2/n\right) \Rightarrow \overline{X}$ also takes on values from $-\infty$ to ∞.
- It is entirely possible that $\bar{X}$ is very large even if the mean of its distribution is 3.
- However, if $\bar{X}$ is very large, it will start to seem more likely that μ is larger than 3.
- Eventually, a population mean of 5 will seem more likely than a population mean of 3.

Reject H_0, in favor of H_1, if $\overline{X} < c$ for some c to be determined.

Step Three

Find c.

- If c is too large, we are making it difficult to reject H_0. We are more likely to fail to reject when it should be rejected.
- If c is too small, we are making it to easy to reject H_0. We are more likely reject when it should not be rejected.

This is where α comes in.

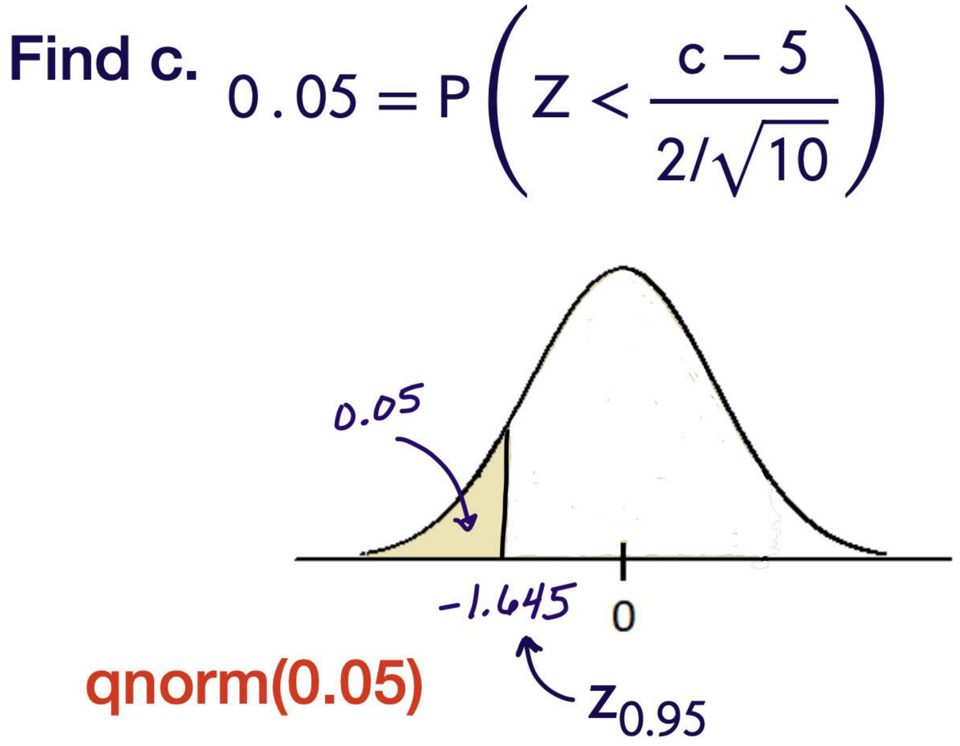
$$\begin{aligned}\alpha &= P(TypeIError)\\ &= P(\text{Reject } H_0 \text{ when true})\\ &= P(\overline{X} < c \text{ when } \mu = 3)\end{aligned}$$

Step Four

Give a conclusion!

$0.05 = P(\text{ Type I Error})$
$= P\text{ (Reject } H_0 \text{ when true)}$
$= P(\overline{X} < \text{ c when } \mu = 5)$

$= P\left(\frac{\overline{X}-\mu_0}{\sigma/\sqrt{n}} < \frac{c-5}{2/\sqrt{10}} \text{ when } \mu = 5\right)$



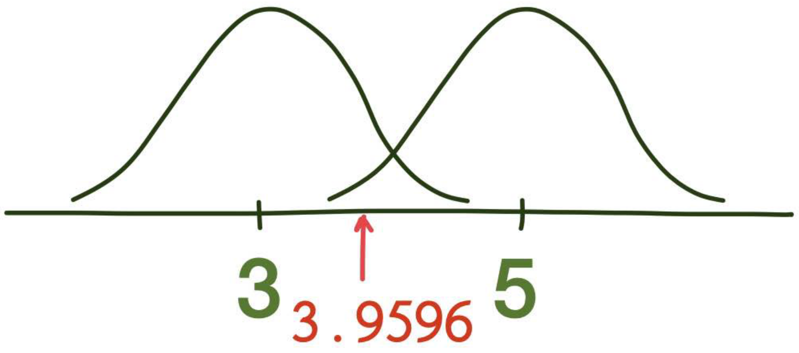
$$0.05 = P\left(Z < \frac{c-5}{2/\sqrt{10}}\right)$$

$$\Rightarrow \quad \frac{c-5}{2/\sqrt{10}} = -1.645$$

$$\Rightarrow \quad c = 3.9596$$

Reject H_0, in favor of H_1, if

$$\overline{X} < 3.9596$$



Formula

Let $X_1, X_2, \ldots, X_n$ be a random sample from the normal distribution with mean μ and known variance σ^2.

Consider testing the simple versus simple hypotheses

$$H_0 : \mu = \mu_0 \quad H_1 : \mu = \mu_1$$

where μ_0 and μ_1 are fixed and known.

$$\begin{aligned} H_0 &: \mu = \mu_0 \\ H_1 &: \mu = \mu_1 \\ \mu_0 &< \mu_1 \\ \text{Reject H0, in favor of H1 if} \end{aligned}$$

$$\overline{X} > \mu_0 + z_\alpha \frac{\sigma}{\sqrt{n}}$$

$$\begin{aligned} H_0 &: \mu = \mu_0 \\ H_1 &: \mu = \mu_1 \\ \mu_0 &> \mu_1 \\ \text{Reject H0, in favor of H1 if} \end{aligned}$$

$$\overline{X} < \mu_0 + z_{1-\alpha} \frac{\sigma}{\sqrt{n}}$$

Type II Error

$$\begin{aligned} H_0 &: \mu = \mu_0 \\ H_1 &: \mu = \mu_1 \\ \mu_0 &< \mu_1 \end{aligned}$$

$$\begin{aligned} \beta = P(&\text{ Type II Error }) \\ =P&\text{ (Fail to Reject } H_0 \text{ when false)} \\ =P&\left(\overline{X} \le \mu_0 + z_\alpha \frac{\sigma}{\sqrt{n}} \text{ when } \mu = \mu_1\right) \\ =P&\left(\overline{X} \le \mu_0 + z_\alpha \frac{\sigma}{\sqrt{n}}; \mu_1\right) \\ \beta = P&\left(\left(\frac{\overline{X} - \mu_1}{\sigma/\sqrt{n}}\right) \le \frac{\mu_0 + z_\alpha \frac{\sigma}{\sqrt{n}} - \mu_1}{\sigma/\sqrt{n}}; \mu_1\right) \\ = P&\left(Z \le \frac{\mu_0 + z_\alpha \frac{\sigma}{\sqrt{n}} - \mu_1}{\sigma/\sqrt{n}}\right) \end{aligned}$$

1.13.5 Composite vs Composite Hypothesis

$$
\begin{aligned}
& X_1, X_2, \ldots, X_n \sim N\left(\mu, \sigma^2\right), \sigma^2 \text{ known} \\
& H_0: \mu \leq \mu_0 \quad \text{vs} \quad H_1: \mu > \mu_0
\end{aligned}
$$

- Step One Choose an estimator for
- Step Two Choose a test statistic: Reject H_0 , in favor of H_1 if¯> c, where c is to be determined.
- Step Three Find c.

1.13.6 One-Tailed Tests

Let $X_1, X_2, \ldots, X_n$ be a random sample from the normal distribution with mean μ and known variance σ^2. Consider testing the hypotheses

$$H_0: \mu \geq \mu_0 \quad H_1: \mu < \mu_0$$

where μ_0 is fixed and known.

Step One

Choose an estimator for .

$$\widehat{\mu} = \bar{X}$$

Step Two

Choose a test statistic or Give the "form" of the test.

Reject H_0, in favor of H_1, if $\overline{X} < c$ for some c to be determined.

Step Three

Find c.

$$
\begin{aligned}
\alpha &= \max_{\mu \geq \mu_0} P(\text{ Type I Error }) \\
&= \max_{\mu \geq \mu_0} P\left(\text{ Reject } H_0; \mu\right) \\
&= \max_{\mu \geq \mu_0} P(\overline{X} < c; \mu)
\end{aligned}
$$

$$
\begin{aligned}
\alpha &= \max_{\mu \geq \mu_0} P(\overline{X} < c; \mu) \\
&= \max_{\mu \geq \mu_0} P\left(Z < \frac{c-\mu}{\sigma/\sqrt{n}}\right) \\
&= \max_{\mu \geq \mu_0} \Phi\left(\frac{c-\mu}{\sigma/\sqrt{n}}\right)
\end{aligned}
$$

$$
\begin{aligned}
\alpha &= \max_{\mu \geq \mu_0} P(\overline{X} < c; \mu) \\
&= \max_{\mu \geq \mu_0} P\left(Z < \frac{c-\mu}{\sigma/\sqrt{n}}\right) \\
&= \max_{\mu \geq \mu_0} \Phi\left(\frac{c-\mu}{\sigma/\sqrt{n}}\right) \\
& \text{decreasing in } \mu
\end{aligned}
$$

Step four

Reject H_0, in favor of H_1, if \$$\overline{X} < \mu_0 + z_{1-\alpha}\frac{\sigma}{\sqrt{n}}$\$

Example

In 2019, the average health care annual premium for a family of 4 in the United States, was reported to be $\$6,015$.

In a more recent survey, 100 randomly sampled families of 4 reported an average annual health care premium of $\$6,537$. Can we say that the true average is currently greater than $\$6,015$ for all families of 4?

Assume that annual health care premiums are normally distributed with a standard deviation of $\$814$. Let μ be the true average for all families of 4.

Step Zero

Set up the hypotheses.

$$H_0 : \mu = 6015 \quad H_1 : \mu > 6015$$

Decide on a level of significance. $\alpha = 0.10$

Step One

Choose an estimator for μ.

$$\hat{\mu} = \bar{X}$$

Step Two

Give the form of the test. Reject H_0, in favor of H_1, if

$$\bar{X} > c$$

for some c to be determined.

Step Three

Find c.

$$\begin{aligned}\alpha &= \max_{\mu=\mu_0} P(\text{ Type I Error; } \mu) \\ &= P(\text{ Type I Error; } \mu_0)\end{aligned}$$

$$\begin{aligned}\alpha &= P(\text{ Reject } H_0; \mu_0) \text{ when} \\ &= P\left(\overline{X} > c; \mu_0\right) \quad \text{ it true!,} \\ &= P\left(\frac{\overline{X}-\mu_0}{\sigma/\sqrt{n}} > \frac{c-6015}{814/\sqrt{100}}; \mu_0\right) \\ &= P\left(Z > \frac{c-6015}{814/\sqrt{100}}\right) \\ &\frac{c-6015}{814/\sqrt{100}} = 1.28\end{aligned}$$

Step Four

Conclusion. Reject H_0, in favor of H_1, if

$$\bar{X} > 6119.19$$

From the data, where $\bar{x} = 6537$, we reject H_0 in favor of H_1.
The data suggests that the true mean annual health care premium is greater than $\$6015$.

1.13.7 Hypothesis Testing with P-Values

Recall that p-values are defined as the following: A p-value is the probability that we observe a test statistic at least as extreme as the one we calculated, assuming the null hypothesis is true. It isn't immediately obvious what that definition means, so let's look at some examples to really get an idea of what p-values are, and how they work.

Let's start very simple and say we have 5 data points: x = <1, 2, 3, 4, 5>. Let's also assume the data were generated from some normal distribution with a known variance σ but an unknown mean μ_0. What would be a good guess for the true mean? We know that this data could come from *any* normal distribution, so let's make two wild guesses:

1. The true mean is 100.
2. The true mean is 3.

Intuitively, we know that 3 is the better guess. But how do we actually determine which of these guesses is more likely? By looking at the data and asking "how likely was the data to occur, assuming the guess is true?"

1. What is the probability that we observed x=<1,2,3,4,5> assuming the mean is 100? Probabiliy pretty low. And because the p-value is low, we "reject the null hypothesis" that $\mu_0 = 100$.
2. What is the probability that we observed x=<1,2,3,4,5> assuming the mean is 3? Seems reasonable. However, something to be careful of is that p-values do not **prove** anything. Just because it is probable for the true mean to be 3, does not mean we know the true mean is 3. If we have a high p-value, we "fail to reject the null hypothesis" that $\mu_0 = 3$.

What do "low" and "high" mean? That is where your significance level α comes back into play. We consider a p-value low if the p-value is less than α, and high if it is greater than α.

Example

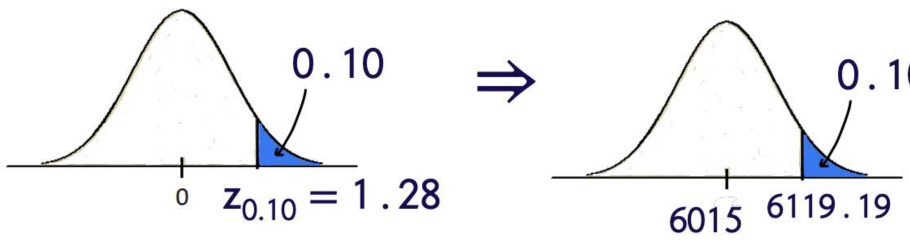
From the above example.

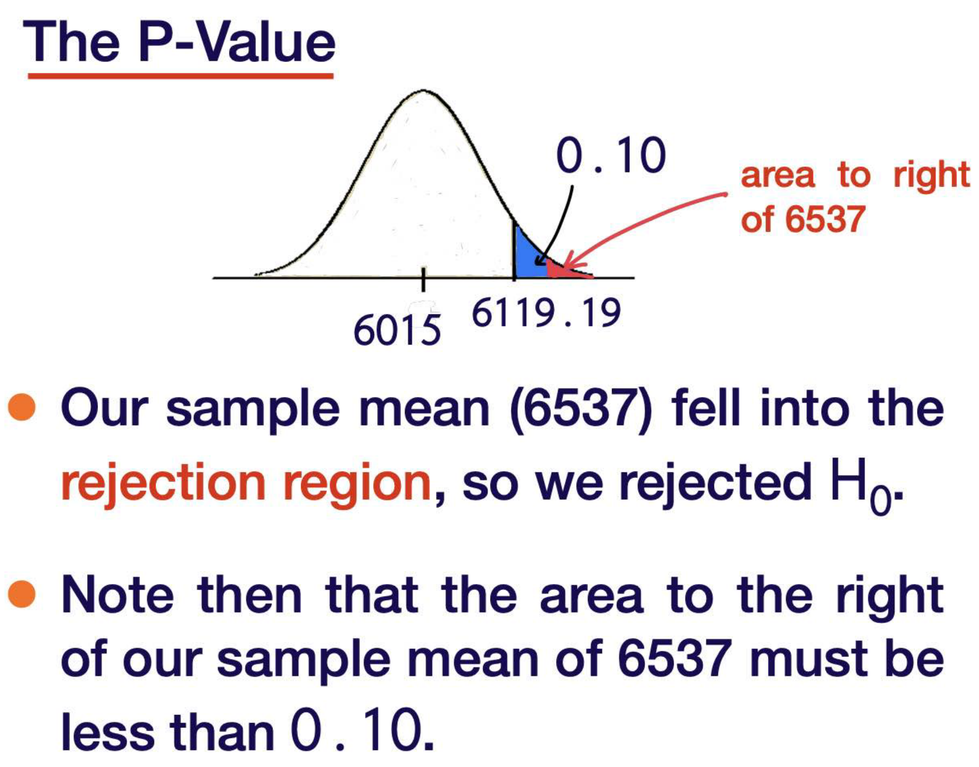
Our size 0.10 test said to

$$\text{Reject } H_0 \text{ if } \overline{X} > 6119.19$$

Our observed sample mean was

$$\bar{x} = 6537$$




0.10
area to right of 6537
6015
6119.19

The P-Value

- Our sample mean (6537) fell into the rejection region, so we rejected H_0.
- Note then that the area to the right of our sample mean of 6537 must be less than 0.10.

- This is the $N\left(6015, 814^2/100\right)$ pdf.
- The red area is $P(\overline{X} > 6537)$.

$$\begin{aligned} &P(\overline{X} > 6537) \\ &\quad = P\left(\frac{\overline{X}-\mu_0}{\sigma/\sqrt{n}} > \frac{6537-6015}{814/\sqrt{100}}\right) \\ &= P(Z > 6.4127) \\ &\approx 0.00000001 \qquad \begin{array}{l}\text{Super small} \\ \text{and way out} \\ \text{"in the tail".}\end{array} \end{aligned}$$

- The P-Value is the area to the right (in this case) of the test statistic $\bar{X}$.
- The P-value being less than 0.10 puts $\bar{X}$ in the rejection region.
- The P-value is also less than 0.05 and 0.01.
- It looks like we will reject H_0 for the most typical values of α.

1.13.8 Power Functions

Let $X_1, X_2, \ldots, X_n$ be a random sample from any distribution with unknown parameter θ which takes values in a parameter space Θ

We ultimately want to test

$$\begin{aligned} H_0 &: \theta \in \Theta_0 \\ H_1 &: \theta \in \Theta \backslash \Theta_0 \end{aligned}$$

where Θ_0 is some subset of Θ.

So in other words, if the null hypothesis was for you to test for an exponential distribution, whether lambda was between 0 and 2, the complement of that is not the rest of the real number line because the space is only non-negative values. So the complement of the interval from 0 to 2 in that space is 2 to infinity.

$\gamma(\theta) = P(\text{ Reject } H_0 \text{ when the parameter is } \theta)$

$$\gamma(\theta) = P(\text{ Reject } H_0; \theta)$$

θ is an argument that can be anywhere in the parameter space Θ. it could be a θ from H_0 it could be a θ from H_1

$$\begin{aligned} \alpha &= \max P(\text{ Reject } H_0 \text{ when true }) \\ &= \max_{\theta \in \Theta_0} P(\text{ Reject } H_0; \theta) \\ &= \max_{\theta \in \Theta_0} \gamma(\theta) \longleftrightarrow \underline{\begin{array}{l} \text{Other notation} \\ \text{is } \max_{\theta \in H_0} \end{array}} \end{aligned}$$

1.13.9 Two Tailed Tests

Let $X_1, X_2, \ldots, X_n$ be a random sample from the normal distribution with mean μ and known variance σ^2.

Derive a hypothesis test of size α for testing

$$\begin{aligned} H_0 &: \mu = \mu_0 \\ H_1 &: \mu \neq \mu_0 \end{aligned}$$

We will look at the sample mean $\bar{X}$ . . . . . . and reject if it is either too high or too low.

Step One

Choose an estimator for .

$$\widehat{\mu} = \bar{X}$$

Step Two

Choose a test statistic or Give the "form" of the test.

Reject H_0, in favor of H_1 if either $\overline{X} < c$ or $\bar{X} > d$ for some c and d to be determined.

Easier to make it symmetric! Reject H_0, in favor of H_1 if either

$$\begin{aligned} \overline{X} &> \mu_0 + c \\ \overline{X} &< \mu_0 - c \end{aligned}$$

for some c to be determined.

Step Three

Find c.

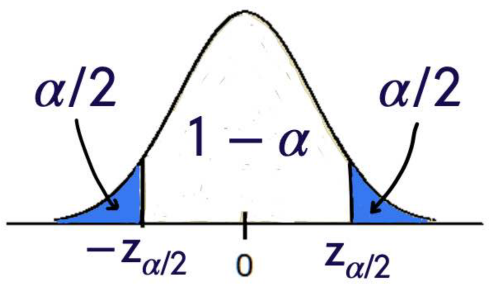
$$\begin{aligned}\alpha &= \max_{\mu=\mu_0} P(\text{ Type I Error })\\ &= \max_{\mu=\mu_0} P\left(\text{ Reject } H_0; \mu\right)\\ &= P\left(\text{ Reject } H_0; \mu_0\right)\end{aligned}$$

$$\begin{aligned}\alpha &= P\left(\overline{X} < \mu_0 - c \text{ or } \overline{X} > \mu_0 + c; \mu_0\right)\\ &= 1 - P\left(\mu_0 - c \leq \overline{X} \leq \mu_0 + c; \mu_0\right)\\ \alpha &= 1 - P\left(\frac{-c}{\sigma/\sqrt{n}} \leq Z \leq \frac{c}{\sigma/\sqrt{n}}\right)\\ 1-\alpha &= P\left(\frac{-c}{\sigma/\sqrt{n}} \leq Z \leq \frac{c}{\sigma/\sqrt{n}}\right)\end{aligned}$$

$$\begin{aligned}\frac{c}{\sigma/\sqrt{n}} &= z_{\alpha/2}\\ c &= z_{\alpha/2}\frac{\sigma}{\sqrt{n}}\end{aligned}$$

$$\alpha = 1 - P\left(\frac{-c}{\sigma/\sqrt{n}} \leq Z \leq \frac{c}{\sigma/\sqrt{n}}\right)$$

$$1-\alpha = P\left(\frac{-c}{\sigma/\sqrt{n}} \leq Z \leq \frac{c}{\sigma/\sqrt{n}}\right)$$



$$\frac{c}{\sigma/\sqrt{n}} = z_{\alpha/2}$$

Step Four

Conclusion

Reject H_0, in favor of H_1, if

$$\overline{X} > \mu_0 + z_{\alpha/2}\frac{\sigma}{\sqrt{n}}$$
$$\overline{X} < \mu_0 - z_{\alpha/2}\frac{\sigma}{\sqrt{n}}$$

Example

In 2019, the average health care annual premium for a family of 4 in the United States, was reported to be $\$6,015$.

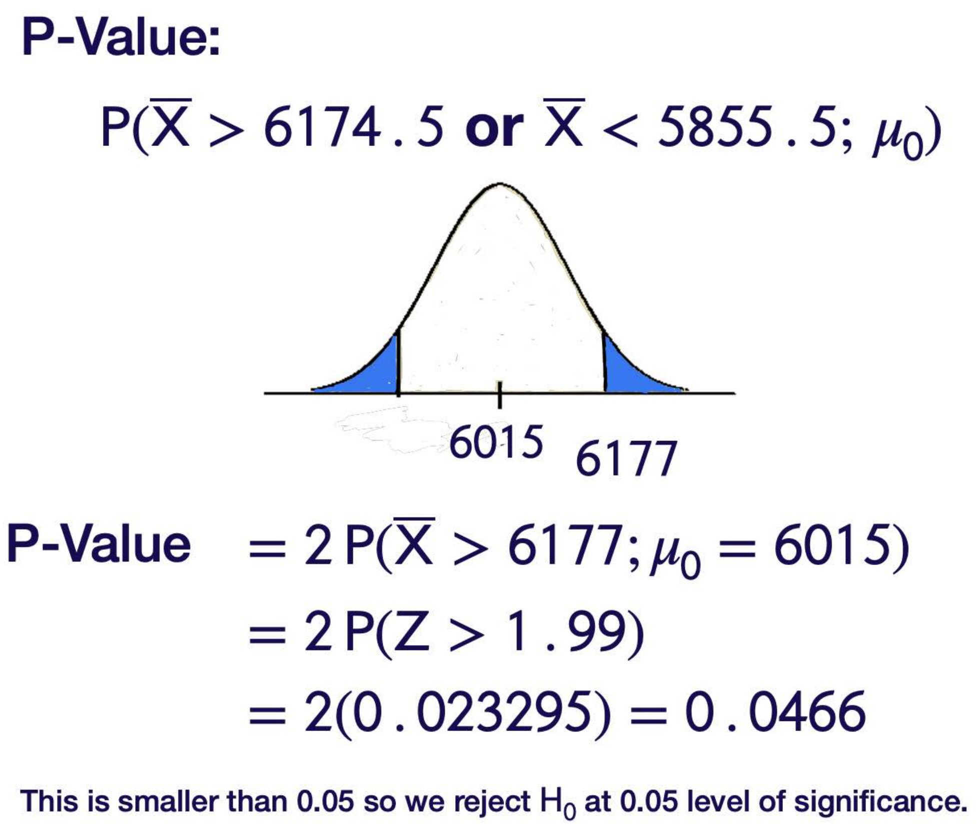
In a more recent survey, 100 randomly sampled families of 4 reported an average annual health care premium of $\$6,177$. Can we say that the true average, for all families of 4 , is currently different than the sample mean from 2019? $\$\sigma = 814 \quad$ Use $\alpha = 0.05\$$

Assume that annual health care premiums are normally distributed with a standard deviation of $\$814$. Let μ be the true average for all families of 4. Hypotheses:

$$H_0 : \mu = 6015$$
$$H_1 : \mu \neq 6015$$

$$\bar{x} = 6177 \quad \sigma = 814 \quad n = 100$$
$$z_{\alpha/2} = z_{0.025} = 1.96$$
$$\text{In R: qnorm(0.975)}$$
$$6015 + 1.96\frac{814}{\sqrt{100}} = 6174.5$$
$$6015 - 1.96\frac{814}{\sqrt{100}} = 5855.5$$

We reject H_0, in favor of H_1. The data suggests that the true current average, for all families of 4 , is different than it was in 2019.



P-Value:

$P(\overline{X} > 6174.5 \text{ or } \overline{X} < 5855.5;\ \mu_0)$

P-Value $= 2\,P(\overline{X} > 6177; \mu_0 = 6015)$

$= 2\,P(Z > 1.99)$

$= 2(0.023295) = 0.0466$

This is smaller than 0.05 so we reject H_0 at 0.05 level of significance.

1.13.10 Hypothesis Tests for Proportions

A random sample of 500 people in a certain country which is about to have a national election were asked whether they preferred "Candidate A" or "Candidate B". From this sample, 320 people responded that they preferred Candidate A.

Let p be the true proportion of the people in the country who prefer Candidate A.

Test the hypotheses $H_0 : p \leq 0.65$ versus $H_1 : p > 0.65$ Use level of significance 0.10. We have an estimate

$$\hat{p} = \frac{320}{500} = \frac{16}{25}$$

The Model

Take a random sample of size n. Record $X_1, X_2, \ldots, X_n$ where $X_i = \begin{cases} 1 & \text{person i likes Candidate A} \\ 0 & \text{person i likes Candidate B} \end{cases}$ Then $X_1, X_2, \ldots, X_n$ is a random sample from the Bernoulli distribution with parameter p.

Note that, with these 1's and 0's, \$ $\begin{aligned} \hat{p} &= \frac{\text{\# in the sample who like A}}{\text{\# in the sample}} \\ &= \frac{\sum_{i=1}^{n} X_i}{n} = \overline{X} \end{aligned}$ $ByTheCentralLimitTheorem,$\hat{p}=\overline{X }\$ has, for large samples, an approximately normal distribution.

$$E[\hat{p}] = E\left[X_1\right] = p$$
$$\operatorname{Var}[\hat{p}] = \frac{\operatorname{Var}\left[X_1\right]}{n} = \frac{p(1-p)}{n}$$

So, $\hat{p} \overset{\text{approx}}{\sim} N\left(p, \frac{p(1-p)}{n}\right)$

$$\hat{p} \overset{\text{approx}}{\sim} N\left(p, \frac{p(1-p)}{n}\right)$$

In particular, \$$\frac{\hat{p}-p}{\sqrt{\frac{p(1-p)}{n}}} behavesroughlylikea$N(0,1)$as$n\$ gets large.

$n > 30$ is a rule of thumb to apply to all distributions, but we can (and should!) do better with specific distributions.

- $\hat{p}$ lives between 0 and 1.
- The normal distribution lives between $-\infty$ and ∞.
- However, 99.7% of the area under a $N(0,1)$ curve lies between -3 and 3 ,

$$\begin{aligned} \hat{p} &\overset{\text{approx}}{\sim} N\left(p, \frac{p(1-p)}{n}\right) \\ &\Rightarrow \sigma_{\hat{p}} = \sqrt{\frac{p(1-p)}{n}} \end{aligned}$$

Go forward using normality if the interval \$$\left(\hat{p} - 3\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}, \hat{p} + 3\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right) iscompletelycontainedwithin$[0,1]\$.

Step One

Choose a statistic. $\widehat{p} =$ sample proportion for Candidate A

Step Two

Form of the test. Reject H_0, in favor of H_1, if $\hat{p} > c$.

Step Three

Use α to find c Assume normality of $\hat{p}$? It is a sample mean and $n > 30$.

- The interval \$$\left(\hat{p} - 3\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}, \hat{p} + 3\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right) is$(0.5756,0.7044)\$

$$\begin{aligned} \alpha &= \max_{p \in H_0} P(\text{ Type I Error }) \\ &= \max_{p \leq 0.65} P(\text{ Reject } H_0; p) \\ &= \max_{p \leq 0.65} P(\hat{p} > c; p) \end{aligned}$$

$$\begin{aligned} \alpha &= \max_{p \leq 0.65} P\left(\frac{\hat{p}-p}{\sqrt{\frac{p(1-p)}{n}}} > \frac{c-p}{\sqrt{\frac{p(1-p)}{n}}}; p\right) \\ &\approx \max_{p \leq 0.65} P\left(Z > \frac{c-p}{\sqrt{\frac{p(1-p)}{n}}}\right) \end{aligned}$$

$$\begin{aligned} 0.10 &= \max_{p \leq 0.65} P\left(Z > \frac{c-p}{\sqrt{\frac{p(1-p)}{n}}}\right) \\ &= P\left(Z > \frac{c-0.65}{\sqrt{\frac{0.65(1-0.65)}{n}}}\right) \\ &\Rightarrow \frac{c-0.65}{\sqrt{\frac{0.65(1-0.65)}{n}}} = z_{0.10} \end{aligned}$$

Reject H_0 if

$$\hat{p} > 0.65 + z_{0.10}\sqrt{\frac{0.65(1-0.65)}{n}}$$

Formula

$$\hat{p} > p + z_{0.10}\sqrt{\frac{p(1-p)}{n}}$$

1.13.11 T-Tests

What is a t-test, and when do we use it? A t-test is used to compare the means of one or two samples, when the underlying population parameters of those samples (mean and standard deviation) are unknown. Like a z-test, the t-test assumes that the sample follows a normal distribution. In particular, this test is useful for when we have a small sample size, as we can not use the Central Limit Theorem to use a z-test.

There are two kinds of t-tests:

1. One Sample t-tests
2. Two Sample t-tests

Let $X_1, X_2, \ldots, X_n$ be a random sample from the normal distribution with mean μ and unknown variance σ^2.

Consider testing the simple versus simple hypotheses \$$H_0 : \mu = \mu_0 \quad H_1 : \mu < \mu_0 where$\mu_0\$ is fixed and known.

Reject H_0, in favor of H_1, if

$$\overline{X} < \mu_0 + z_{1-\alpha}\frac{\sigma}{\sqrt{n}}$$

unknown!This is a useless test!

It was based on the fact that

$$\overline{X} \sim N\left(\mu, \sigma^2/n\right)$$

$$\frac{\overline{X}-\mu}{\sigma/\sqrt{n}} \sim N(0,1)$$

What is we use the sample standard deviation $S = \sqrt{S^2}$ in place of σ ?

$$\begin{aligned} \frac{\overline{X}-\mu}{S/\sqrt{n}} &= \frac{\overline{X}-\mu}{\sigma/\sqrt{n}} \cdot \frac{\sigma}{S} = \frac{\frac{\overline{X}-\mu}{\sigma/\sqrt{n}}}{\frac{S}{\sigma}} \\ &= \frac{\overline{X}-\mu}{\sigma/\sqrt{n}} / \sqrt{\frac{S^2}{\sigma^2}} \end{aligned}$$

$$\frac{\overline{X}-\mu}{S/\sqrt{n}} = \frac{\overline{X}-\mu}{\sigma/\sqrt{n}} / \sqrt{\frac{S^2}{\sigma^2}}$$
$$= \left(\frac{X-\mu}{\sigma/\sqrt{n}}\right) / \sqrt{\frac{\left(\frac{(n-1)S^2}{\sigma^2}\right.}{n-1}\chi^2(n-1)}$$
$$N(0,1)$$

$$\frac{\overline{X}-\mu}{S/\sqrt{n}} = \frac{\overline{X}-\mu}{\sigma/\sqrt{n}} / \sqrt{\frac{S^2}{\sigma^2}}$$
$$= \left(\frac{X-\mu}{\sigma/\sqrt{n}}\right) / \sqrt{\frac{\left(\frac{(n-1)S^2}{\sigma^2}\right.}{n-1}\chi^2(n-1)}$$
$$N(0,1)$$

Thus,

$$\frac{\bar{X}-\mu}{S/\sqrt{n}} \sim t(n-1)$$

Step four

Conclusion! Reject H_0, in favor of H_1, if

$$\overline{X} < \mu_0 + t_{1-\alpha,n-1}\frac{S}{\sqrt{n}}$$

Example

In 2019, the average health care annual premium for a family of 4 in the United States, was reported to be $\$6,015$.

In a more recent survey, 15 randomly sampled families of 4 reported an average annual health care premium of $\$6,033$ and a sample variance of $\$825$.

Can we say that the true average is currently greater than $\$6,015$ for all families of 4 ?

Use $\alpha = 0.10$

Assume that annual health care premiums are normally distributed. Let μ be the true average for all families of 4.

Step Zero

Set up the hypotheses.

$$H_0 : \mu = 6015 \quad H_1 : \mu > 6015$$

Step One

Choose a test statistic

$$\bar{X}$$

Step Two

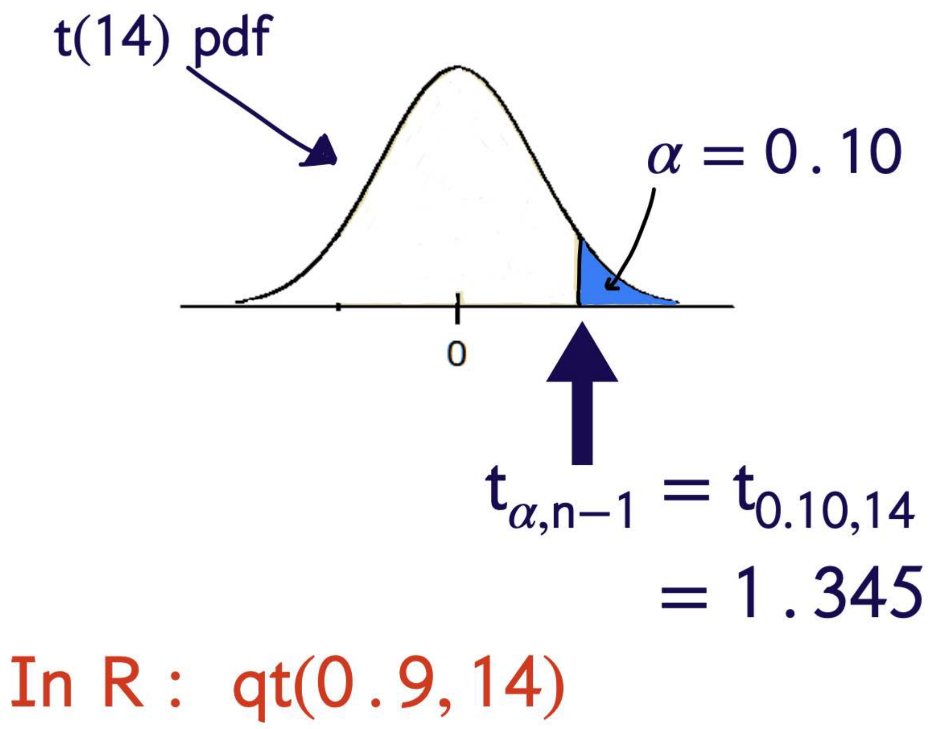
Give the form of the test. Reject 0 , in favor of h1, if > where c is to be determined.

Step Three

Find c

$$\begin{aligned}\alpha &= \max_{\mu=\mu_0} P(\text{ Type I Error }) \\ &= \max_{\mu=6015} P(\text{ Reject } H_0; \mu) \\ &= P(\text{ Reject } H_0; \mu=6015) \\ &= P(\overline{X} > c; \mu=6015)\end{aligned}$$

$$\begin{aligned}\alpha &= P(\overline{X} > c; \mu=6015) \\ = P\left(\frac{\overline{X}-\mu_0}{S/\sqrt{n}} > \frac{c-6015}{\sqrt{825}/\sqrt{15}}; \mu=6015\right) \\ &= P\left(T > \frac{c-6015}{\sqrt{825}/\sqrt{15}}\right)\end{aligned}$$



$$\Rightarrow \frac{c - 6015}{\sqrt{825}/\sqrt{15}} = 1.345$$
$$\Rightarrow c = 6024.98$$

Step Four

Conclusion. Rejection Rule: Reject H_0, in favor of H_1 if

$$\bar{X} > 6024.98$$

We had $\bar{x} = 6033$ so we reject H_0.

There is sufficient evidence (at level 0.10) in the data to suggest that the true mean annual healthcare premium cost for a family of 4 is greater than $\$6,015$.

P value

$$\text{P-Value } = P(\overline{X} > 6033; \mu = 6015)$$
$$= P\left(\frac{\overline{X} - \mu}{S/\sqrt{n}} > \frac{6033 - 6015}{\sqrt{825}/\sqrt{15}}; \mu = 6015\right)$$
$$= P(T > 2.43) \approx 0.015$$
$$\text{where } T \sim t(14)$$
$$(\text{In R} : 1 - pt(2.43, 14)$$

1.13.12 Two Sample Tests for Means

Fifth grade students from two neighboring counties took a placement exam.

Group 1, from County 1, consisted of 57 students. The sample mean score for these students was 77.2 and the true variance is known to be 15.3. Group 2, from County 2, consisted of 63 students and had a sample mean score of 75.3 and the true variance is known to be 19.7.

From previous years of data, it is believed that the scores for both counties are normally distributed.

Derive a test to determine whether or not the two population means are the same.

$$H_0 : \mu_1 = \mu_2$$
$$H_1 : \mu_1 \neq \mu_2$$

Suppose that $X_{1,1}, X_{1,2}, \ldots, X_{1,n_1}$ is a random sample of size n_1 from the normal distribution with mean μ_1 and variance σ_1^2. Suppose that $X_{2,1}, X_{2,2}, \ldots, X_{2,n_2}$ is a random sample of size n_2 from the normal distribution with mean μ_2 and variance σ_2^2.

- Suppose that σ_1^2 and σ_2^2 are known and that the samples are independent.

$$H_0 : \mu_1 = \mu_2 \quad H_1 : \mu_1 \neq \mu_2$$
$$H_0 : \mu_1 - \mu_2 = 0$$
$$H_1 : \mu_1 - \mu_2 \neq 0$$

Think of this as \$ $\theta = 0$ versus $\theta \neq 0$ for $\theta = \mu_1 - \mu_2$ \$

Step one

Choose an estimator for $\theta = \mu_1 - \mu_2$

$$\hat{\theta} = \bar{X}_1 - \bar{X}_2$$

Step Two

Give the "form" of the test. Reject H_0, in favor of H_1 if either $\hat{\theta} > c$ or $\hat{\theta} < -c$ for some c to be determined.

Step Three

Find c using α Will be working with the random variable

$$\bar{X}_1 - \bar{x}_2$$

We need to know its distribution…

$$\bar{X}_1 - \bar{x}_2 \text{ is normally distributed.}$$

Step Three

Find c using α.

$\bar{X}_1 - \bar{X}_2$ is normally distributed

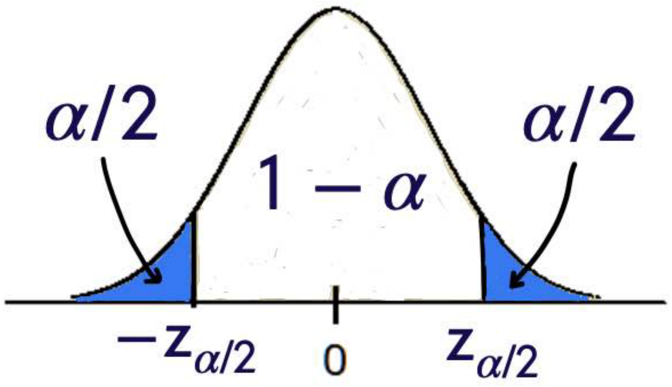
$$\overline{X}_1 - \overline{X}_2 \sim N\left(\mu_1 - \mu_2, \frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_1}\right)$$

$$Z = \frac{\overline{X}_1 - \overline{X}_2 - (\mu_1 - \mu_2)}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} \sim N(0,1)$$

$$\begin{aligned}
\alpha &= P(\text{ Type I Error })\\
&= P(\text{ Reject } H_0; \theta = 0)\\
&= P\left(\overline{X}_1 - \overline{X}_2 > c \text{ or } \overline{X}_1 - \overline{X}_2 < -c; \theta = 0\right)\\
&= 1 - P\left(-c \leq \overline{X}_1 - \overline{X}_2 \leq c; \theta = 0\right)\\
&= 1 - P\left(-c \leq \overline{X}_1 - \overline{X}_2 \leq c; \theta = 0\right)
\end{aligned}$$

$$\alpha = 1 - P\left(\frac{-c}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} \leq Z \leq \frac{c}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}\right)$$

$$1 - \alpha = P\left(\frac{-c}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} \leq Z \leq \frac{c}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}\right)$$



$$\Rightarrow \quad \frac{c}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} = z_{\alpha/2}$$

Step Four

Conclusion

Reject H_0, in favor of H_1, if

$$\overline{X}_1 - \overline{X}_2 > z_{\alpha/2}\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}$$

or

$$\overline{X}_1 - \overline{X}_2 < -z_{\alpha/2}\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}$$

Example

$$\begin{array}{ll} n_1 = 57 & n_2 = 63 \\ \overline{x}_1 = 77.2 & \overline{x}_2 = 75.3 \\ \sigma_1^2 = 15.3 & \sigma_2^2 = 19.7 \end{array}$$

Suppose that $\alpha = 0.05$. $ $\begin{array}{l} z_{\alpha/2} = z_{0.025} = 1.96 \\ z_{\alpha/2}\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}} = 1.49 \end{array}$ $

$$z_{\alpha/2}\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}} = 1.49$$
$$\overline{x}_1 - \overline{x}_2 = 77.2 - 75.3 = 1.9$$

So,

$$\bar{x}_1 - \bar{x}_2 > z_{\alpha/2}\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}$$

and we reject H_0. The data suggests that the true mean scores for the counties are different!

1.13.13 Two Sample t-Tests for a Difference of Means

Fifth grade students from two neighboring counties took a placement exam.

- Group 1, from County A, consisted of 8 students. The sample mean score for these students was 77.2 and the sample variance is 15.3.
- Group 2, from County B, consisted of 10 students and had a sample mean score of 75.3 and the sample variance is 19.7.

Pooled Variance

$$S_p^2 = \frac{(n_1 - 1)\,S_1^2 + (n_2 - 1)\,S_2^2}{n_1 + n_2 - 2}$$

Step Four

Reject H_0, in favor of H_1, if

$$\bar{X}_1 - \bar{X}_2 > t_{\alpha/2, n_1+n_2-2}\sqrt{\left(\frac{1}{n_1} + \frac{1}{n_2}\right) S_P^2}$$

or

$$\bar{X}_1 - \bar{X}_2 < -t_{\alpha/2, n_1+n_2-2}\sqrt{\left(\frac{1}{n_1} + \frac{1}{n_2}\right) s_P^2}$$

$$\begin{aligned} n_1 &= 8 & n_1 &= 10 \\ \overline{x}_1 &= 77.2 & \overline{x}_1 &= 75.3 \\ s_1^2 &= 15.3 & s_2^2 &= 19.7 \\ \alpha &= 0.01 & t_{0.005,16} &= 2.92 \\ s_p^2 & & &= \tfrac{(n_1-1)S_1^2+(n_2-1)S_2^2}{n_1+n_2-2} \\ & & &= 17.775 \end{aligned}$$

$$\begin{aligned} &\overline{x}_1 - \overline{x}_2 = 77.2 - 75.3 = 1.9 \\ &t_{\alpha/2, n_1+n_2-2}\sqrt{\left(\frac{1}{n_1} + \frac{1}{n_2}\right) S_P^2} \\ &\quad = 2.92\sqrt{\left(\frac{1}{8} + \frac{1}{10}\right)(17.775)} \\ &= 5.840 \end{aligned}$$

Since $\bar{x}_1 - \bar{x}_2 = 1.9$ is not above 5.840, or below -5.840 we fail to reject H_0, in favor of H_1 at 0.01 level of significance.

The data do not indicate that there is a significant difference between the true mean scores for counties A and B.

1.13.14 Welch's Test and Paired Data

Two Populations: Test

$$\begin{aligned} H_0 &: \mu_1 = \mu_2 \\ H_1 &: \mu_1 \neq \mu_2 \end{aligned}$$

- Suppose that $X_{1,1}, X_{1,2}, \ldots, X_{1,n_1}$ is a random sample of size n_1 from the normal distribution with mean μ_1 and variance σ_1^2.
- Suppose that $X_{2,1}, X_{2,2}, \ldots, X_{2,n}$ is a random sample of size n_2 from the normal distribution with mean μ_2 and variance σ_2^2.
- Suppose that σ_1^2 and σ_2^2 are unknown and that the samples are independent. Don't assume that σ_1^2 and σ_2^2 are equal!

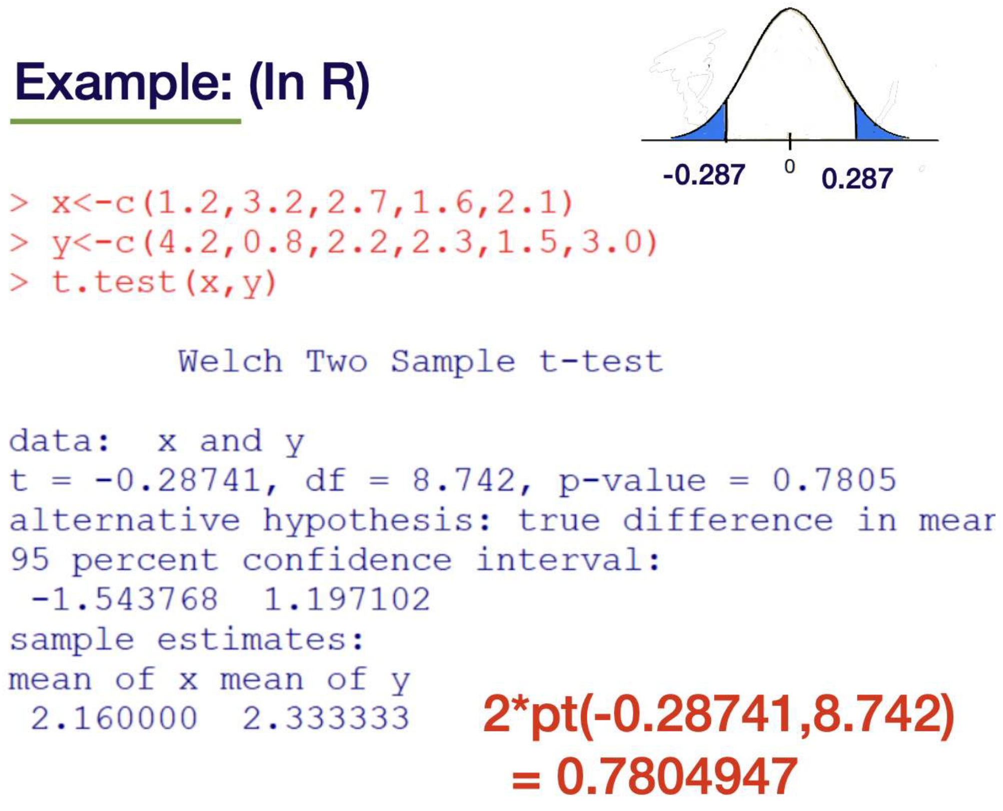
Welch says that:

$$\frac{\bar{X}_1 - \bar{X}_2 - (\mu_1 - \mu_2)}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

has an approximate t-distribution with r degrees of freedom where

$$r = \frac{S_1^2/n_1 + S_2^2/n_2}{\frac{\left(S_1^2/n_1\right)^2}{n_1-1} + \frac{\left(S_2^2/n_2\right)^2}{n_2-1}}$$

rounded down.



Example: (In R)

```
> x<-c(1.2,3.2,2.7,1.6,2.1)
> y<-c(4.2,0.8,2.2,2.3,1.5,3.0)
> t.test(x,y)

        Welch Two Sample t-test

data:  x and y
t = -0.28741, df = 8.742, p-value = 0.7805
alternative hypothesis: true difference in mear
95 percent confidence interval:
 -1.543768  1.197102
sample estimates:
mean of x mean of y
 2.160000  2.333333
```

Example

A random sample of 6 students' grades were recorded for Midterm 1 and Midterm 2. Assuming exam scores are normally distributed, test whether the true (total population of students) average grade on Midterm 2 is greater than Midterm 1. = 0.05

Student	Midterm 1 Grade	Midterm 2 Grade
1	72	81
2	93	89
3	85	87
4	77	84
5	91	100
6	84	82

Student	Midterm 1 Grade	Midterm 2 Grade	Differences: minus 2 Midterm 1
1	72	81	9
2	93	89	-4
3	85	87	2
4	77	84	7
5	91	100.	9
6	84	82	-2

The Hypotheses: Let μ be the true average difference for all students.

$$H_0 : \mu = 0$$
$$H_1 : \mu > 0$$

This is simply a one sample t-test on the differences.

Data:

$$9, -4, 2, 7, 9, -2$$

$$\sum x_i = 23 \quad \sum x_i^2 = 267 \quad n = 6$$

This is simply a one sample t-test on the differences.

This is simply a one sample t-test on the differences.

$$\bar{x} = 3.5$$
$$s^2 = \frac{\sum x_i^2 - \left(\sum x_i\right)^2 / n}{n-1} = 32.3$$

$$t_{\alpha,n-1} = t_{0.05,5} = 2.01$$

Reject H_0, in favor of H_1, if

$$\overline{X} > \mu_0 + t_{\alpha,n-1}\frac{S}{\sqrt{n}}$$

3.5 > 4.6

Conclusion: We fail to reject h0 , in favor of h1 , at 0.05 level of significance.

These data do not indicate that Midterm 2 scores are higher than Midterm 1 scores

1.13.15 Comparing Two Population Proportions

A random sample of 500 people in a certain county which is about to have a national election were asked whether they preferred "Candidate A" or "Candidate B". From this sample, 320 people responded that they preferred Candidate A.

A random sample of 400 people in a second county which is about to have a national election were asked whether they preferred "Candidate A" or "Candidate B".

From this second county sample, 268 people responded that they preferred Candidate A.

$$\hat{p}_1 = \frac{320}{500} = 0.64$$
$$\hat{p}_2 = \frac{268}{400} = 0.67$$

Test

$$H_0 : p_1 = p_2 \quad H_1 : p_1 \neq p_2$$

Change to:

$$H_0 : p_1 - p_2 = 0$$
$$H_1 : p_1 - p_2 \neq 0$$

Estimate $p_1 - p_2$ with $\hat{p}_1 - \hat{p}_2$ For large enough samples,

$$\hat{p}_1^{\text{approx}} N\left(p_1, \frac{p_1(1-p_1)}{n_1}\right)$$

and

$$\hat{p}_2^{\text{approx}} N\left(p_2, \frac{p_2(1-p_2)}{n_1}\right)$$

$$\begin{aligned} \hat{p}_1 - \hat{p}_2 &\sim N(?,?) \\ E[\hat{p}_1 - \hat{p}_2] = E[\hat{p}_1] - E[\hat{p}_2] &= p_1 - p_2 \\ \operatorname{Var}[\hat{p}_1 - \hat{p}_2] &\overset{\text{indep}}{=} \operatorname{Var}[\hat{p}_1] + \operatorname{Var}[\hat{p}_2] \\ &= \frac{p_1(1-p_1)}{n_1} + \frac{p_2(1-p_2)}{n_2} \end{aligned}$$

Use estimators for p1 and p2 assuming they are the same.

- Call the common value p.
- Estimate by putting both groups together.

$$\hat{p}_1 = \frac{320}{500} = 0.64 \quad \hat{p}_2 = \frac{268}{400} = 0.67$$

we have

$$\begin{aligned} \hat{p} = \frac{320+268}{500+400} &= \frac{588}{900} = \frac{49}{75} \\ &\approx 0.6533 \end{aligned}$$

$$\begin{aligned} Z &:= \frac{\hat{p}_1 - \hat{p}_2 - (p_1 - p_2)}{\sqrt{\frac{\hat{p}(1-\hat{p})}{n_1} + \frac{\hat{p}(1-\hat{p})}{n_2}} \sim N(0,1)} \\ &= \frac{\hat{p}_1 - \hat{p}_2 - (p_1 - p_2)}{\sqrt{\hat{p}(1-\hat{p})\left(\frac{1}{n_1} + \frac{1}{n_2}\right)}} \end{aligned}$$

Two-tailed test with z-critical values…

$$\begin{aligned} \hat{p} &= \frac{320+268}{500+400} = \frac{588}{900} = \frac{49}{75} \\ Z &= \frac{0.64 - 0.67 - 0}{\sqrt{0.6533(1-0.6533)\left(\frac{1}{500} + \frac{1}{400}\right)}} \end{aligned}$$

= 0.9397

$$z_{0.025} = 1.96$$

qnorm(1-0.05/2)

$Z = -0.9397$ does not fall in the rejection region!

1.13.16 Hypothesis Tests for the Exponential

Suppose that $X_1, X_2, \ldots, X_n$ is a random sample from the exponential distribution with rate $\lambda > 0$. Derive a hypothesis test of size α for

$$H_0 : \lambda = \lambda_0 \text{ vs. } H_1 : \lambda > \lambda_0$$

What statistic should we use?

Test 1: Using the Sample Mean

Step One

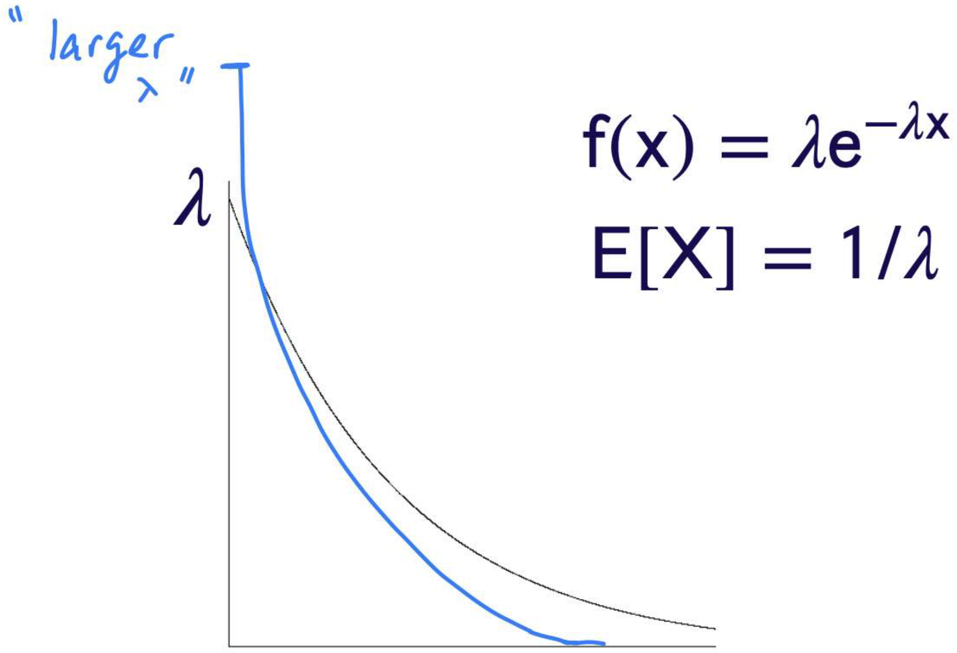
Choose a statistic.

$$\bar{x}$$

Step Two

Give the form of the test Reject 0 , in favor of h1 , if _bar <

for some c to be determined.

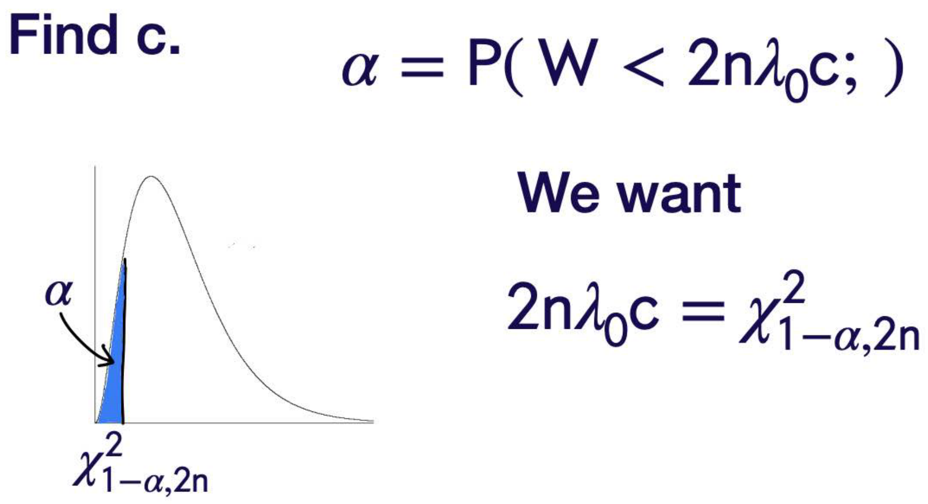


Step Three

$$\begin{aligned}\alpha &= P(\text{ Type I Error }) \\ &= P(\text{ Reject } H_0; \lambda_0) \\ &= P\left(\overline{X} < c; \lambda_0\right)\end{aligned}$$

$$\begin{aligned}&= P\left(2n\lambda_0\overline{x} < 2n\lambda_0 c; \lambda_0\right) \\ &= P\left(W < 2n\lambda_0 c; \lambda_0\right)\end{aligned}$$

where $W \sim \chi^2(2n)$



Find c.

$\alpha = \text{P}(\text{ W} < 2\text{n}\lambda_0\text{c};)$

We want

$2\text{n}\lambda_0\text{c} = \chi^2_{1-\alpha,2\text{n}}$

Step Four

Reject H_0, in favor of H_1, if

$$\bar{x} < \frac{\chi^2_{1-\alpha,2n}}{2n\lambda_0}$$

$\chi^2_{\alpha,n}$ In R, get $\chi^2_{0.10,6}$
by typing qchisq(0.90,6)

1.13.17 Best Test

1.13.18 UMP Tests

Suppose that $X_1, X_2, \ldots, X_n$ is a random sample from the exponential distribution with rate $\lambda > 0$.

Derive a uniformly most powerful hypothesis test of size α for

$$\begin{gathered}H_0 : \lambda = \lambda_0 \quad \text{vs.} \quad H_1 : \lambda > \lambda_0 \\ (\text{ Was } H_1 : \lambda = \lambda_1 \text{ for } \lambda_1 > \lambda_0)\end{gathered}$$

Step One

Consider the simple versus simple hypotheses

$$H_0 : \lambda = \lambda_0 \quad \text{vs. } H_1 : \lambda = \lambda_1$$

for some fixed $\lambda_1 > \lambda_0$.

###Steps Two, Three, and Four

Find the best test of size α for

$$H_0 : \lambda = \lambda_0 \text{ vs. } H_1 : \lambda = \lambda_1$$

for some fixed $\lambda_1 > \lambda_0$. This test is to reject H_0, in favor of H_1 if

$$\bar{x} < \frac{\chi^2_{1-\alpha,2n}}{2n\lambda_0}$$

Note that this test does not depend on the particular value of λ_1. -It does, however, depend on the fact that $\lambda_1 > \lambda_0$

The "UMP" test for

$$H_0 : \lambda = \lambda_0 \text{ vs. } H_1 : \lambda > \lambda_0$$

is to reject H_0, in favor of H_1 if

$$\bar{x} < \frac{\chi^2_{1-\alpha,2n}}{2n\lambda_0}$$

The "UMP" test for

$$H_0 : \lambda = \lambda_0 \text{ vs. } H_1 : \lambda < \lambda_0$$

is to reject H_0, in favor of H_1 if

$$\bar{x} > \frac{\chi^2_{,2n}}{2n\lambda_0}$$

1.13.19 Test for the Variance of the Normal Distribution

Suppose that $X_1, X_2, \ldots, X_n$ is a random sample from the normal distribution with mean μ and variance σ^2. Derive a test of size/level α for

$$H_0 : \sigma^2 \geq \sigma_0^2 \quad \text{vs. } H_1 : \sigma^2 < \sigma_0^2$$

step 1

Choose a statistic/estimator for σ^2

$$s^2 = \frac{\sum_{i=1}^{n} \left(X_i - \bar{X}\right)^2}{n-1}$$

step 2

Give the form of the test. Reject H_0, in favor of H_1, if

$$S^2 < C$$

for some c to be determined.

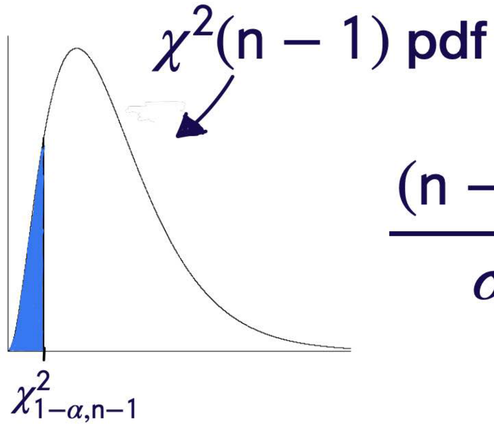
step 3

find c using alpha

$$\begin{aligned}\alpha &= \max P(\text{ Type I Error })\\ &= \max_{\sigma^2 \geq \sigma_0^2} P\left(\text{ Reject } H_0; \sigma^2\right)\\ &= \max_{\sigma^2 \geq \sigma_0^2} P\left(S^2 < c; \sigma^2\right)\end{aligned}$$

$$\begin{aligned}&= P\left(\left(\frac{(n-1)S^2}{\sigma^2}\right)\frac{(n-1)c}{\sigma^2}; \sigma^2\right)\\ &= P\left(W < \frac{(n-1)c}{\sigma^2}\right)\\ &\text{where } W \sim \chi^2(n-1)\end{aligned}$$

$$\alpha = P\left(W < \frac{(n-1)c}{\sigma_0^2}\right)$$



$$\Downarrow$$

$$\frac{(n-1)c}{\sigma_0^2} = \chi^2_{1-\alpha,n-1}$$

Step 4

Reject H_0, in favor of H_1, if

$$S^2 < \frac{\sigma_0^2 \chi^2_{1-\alpha,n-1}}{n-1}$$

Example

A lawn care company has developed and wants to patent a new herbicide applicator spray nozzle. Example: For safety reasons, they need to ensure that the application is consistent and not highly variable. The company selected a random sample of 10 nozzles and measured the application rate of the herbicide in gallons per acre

The measurements were recorded as

$0.213, 0.185, 0.207, 0.163, 0.179$
$0.161, 0.208, 0.210, 0.188, 0.195$

Assuming that the application rates are normally distributed, test the following hypotheses at level 0.04.

$$H_0 : \sigma^2 = 0.01 \quad H_1 : \sigma^2 > 0.01$$

Get sample variance in R.

$$\begin{aligned} x < -c(&0.213, 0.185, 0.207, 0.163, 0.179 \\ &0.161, 0.208, 0.210, 0.188, 0.195) \end{aligned}$$

or

$$x < -\operatorname{scan} 0$$

Hit and then input numbers, one by one, hitting in between and <Enter $>$ at the end.

Compute variance by typing

$$\operatorname{var}(x)$$

or $((\operatorname{sum}(x\hat{}2) - (\operatorname{sum}(x)\hat{}2)/10)/9$ Result: 0.000364

Reject H_0, in favor of H_1, if $S^2 > c$.

$$\begin{aligned} \alpha &= P\left(S^2 > c; \sigma^2 = 0.01\right) \\ &= P\left(\frac{(n-1)S^2}{\sigma^2} > \frac{9c}{0.01}; \sigma^2 = 0.01\right) \\ &= P\left(W > \frac{9c}{0.01}\right) \end{aligned}$$

Reject H_0, in favor of H_1, if $S^2 > c$

$$\begin{aligned} 0.04 &= P\left(W > \frac{9c}{0.01}\right) \\ \frac{9c}{0.01} &= \chi^2_{0.04,9} = 17.61 \\ &\text{qchisq(1-0.04,9)} \end{aligned}$$

Reject H_0, in favor of H_1, if $S^2 > c$

$$\begin{aligned} c &= (17.61)(0.01)/9 \approx 0.0196 \\ s^2 &= 0.000364 \end{aligned}$$

Fail to reject H_0, in favor of H_1, at level 0.04. There is not sufficient evidence in the data to suggest that $\sigma^2 > 0.01$.

1.14 Introduction

Calculus is a branch of mathematics that gives tools to study rate of change of functions trough two main areas: derivatives and integrals.

In the context of machine learning and data science, you can for instance use derivatives to optimize the parameters of a model with gradient descent. You might use integrals to calculate area under the curve.

1.14.1 Functions

A function is a rule that takes one or more inputs and produces a single output. For example, the function $f(x) = x+1$ takes a single input x, adds one to it, and produces a single output. In algebra, functions are written using symbols and formulas. For example, the function $f(x) = x + 1$ can be written as $f : x \to x + 1$. The input to a function is called the **argument** or **input variable**. The output is called the **value** or **output variable**.

Functions are often written using the following notation:

$$y = f(x)$$

The notation above is read as "y equals f of x" or "y is a function of x". The notation above is useful because it allows us to define a function without specifying its name. For example, we can define a function f as follows:

$$f(x) = x^2$$

We can then use the function f to compute the square of any number. For example, $f(2) = 2^2 = 4$ and $f(3) = 3^2 = 9$.

$$\begin{aligned} \text{f}(x) &= \sqrt{x+6} \\ \text{f}(6) &= \sqrt{10+6} \\ \text{f}(6) &= 4.0 \end{aligned}$$

$$\begin{aligned} f(x) &= \frac{x-3}{x+2} \\ f(3) &= \frac{3-3}{3+2} = \frac{0}{5} = 0 \end{aligned}$$

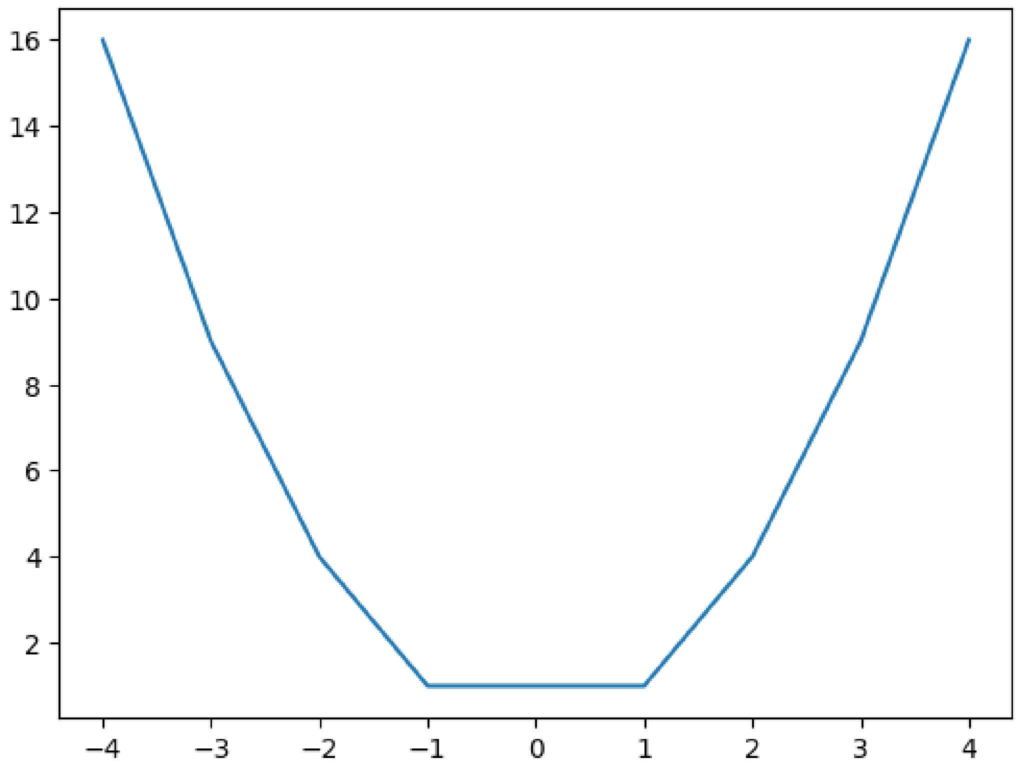
Domain and Range of a Function

The **domain** of a function is the set of all possible inputs to the function. The **range** of a function is the set of all possible outputs of the function. For example, the function $f(x) = x^2$ has a domain of all real numbers and a range of all non-negative real numbers. The domain of a function is often written as $D(f)$ and the range is often written as $R(f)$.

$$\begin{aligned} y &= f(x) \\ y &= x^2 \end{aligned}$$

```python
import seaborn as sb

func = lambda x: x ** 2

x = [-1,-2,-3, -4, 1, 2, 3, 4]
y = [func(i) for i in x]

sb.lineplot(x=x, y=y)
```

```
<Axes: >
```



Piecewise Functions

A piecewise function is a function that is defined by multiple sub-functions, each sub-function applying to a different interval of the main function's domain. For example, the function $f(x) = |x|$ is defined by two sub-functions:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

1.14.2 Expoents

An exponent is a number that indicates how many times a base number is multiplied by itself. For example, 2^3 is the same as $2 \times 2 \times 2$ and 2^4 is the same as $2 \times 2 \times 2 \times 2$. The number 2 is called the **base** and the number 3 is called the **exponent**. Exponents are often written using the following notation:

$$2^3 = 2 \times 2 \times 2 = 8$$

The notation above is read as "two to the power of three" or "two cubed".

Negative Exponents

A negative exponent indicates that the base number should be divided by itself a certain number of times. For example, 2^{-3} is the same as $\frac{1}{2^3}$ and 2^{-4} is the same as $\frac{1}{2^4}$. The number 2 is called the **base** and the number -3 is called the **exponent**. Negative exponents are often written using the following notation:

$$2^{-3} = \frac{1}{2^3} = \frac{1}{8}$$

The notation above is read as "two to the power of negative three" or "two to the power of minus three".

Fractional Exponents

A fractional exponent indicates that the base number should be multiplied by itself a certain number of times. For example, $2^{\frac{1}{2}}$ is the same as $\sqrt{2}$ and $2^{\frac{1}{3}}$ is the same as $\sqrt[3]{2}$. The number 2 is called the **base** and the number $\frac{1}{2}$ is called the **exponent**. Fractional exponents are often written using the following notation:

$$2^{\frac{1}{2}} = \sqrt{2} = 1.414213562373095$$

The notation above is read as "two to the power of one half" or "two to the power of one over two".

1.14.3 Logarithms

A logarithm is the inverse of an exponent. For example, the logarithm of 2^3 is 3. The logarithm of a number x to the base b is written as $\log_b(x)$. For example, $\log_2(8) = 3$ because $2^3 = 8$.

Common Logarithms

The common logarithm of a number x is the logarithm of x to the base 10. The common logarithm of x is written as $\log(x)$. For example, $\log(100) = 2$ because $10^2 = 100$.

Natural Logarithms

The natural logarithm of a number x is the logarithm of x to the base e. The natural logarithm of x is written as $\ln(x)$. For example, $\ln(100) = 4.60517$ because $e^{4.60517} = 100$.

1.14.4 Polynomials

A polynomial is an expression consisting of variables and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents.

For example, $x^2 + 2x + 1$ is a polynomial because it consists of the variables x and the coefficients 1 and 2.

The degree of a polynomial is the highest degree of its terms. For example, the polynomial $x^2 + 2x + 1$ has a degree of 2 because its highest degree term is x^2.

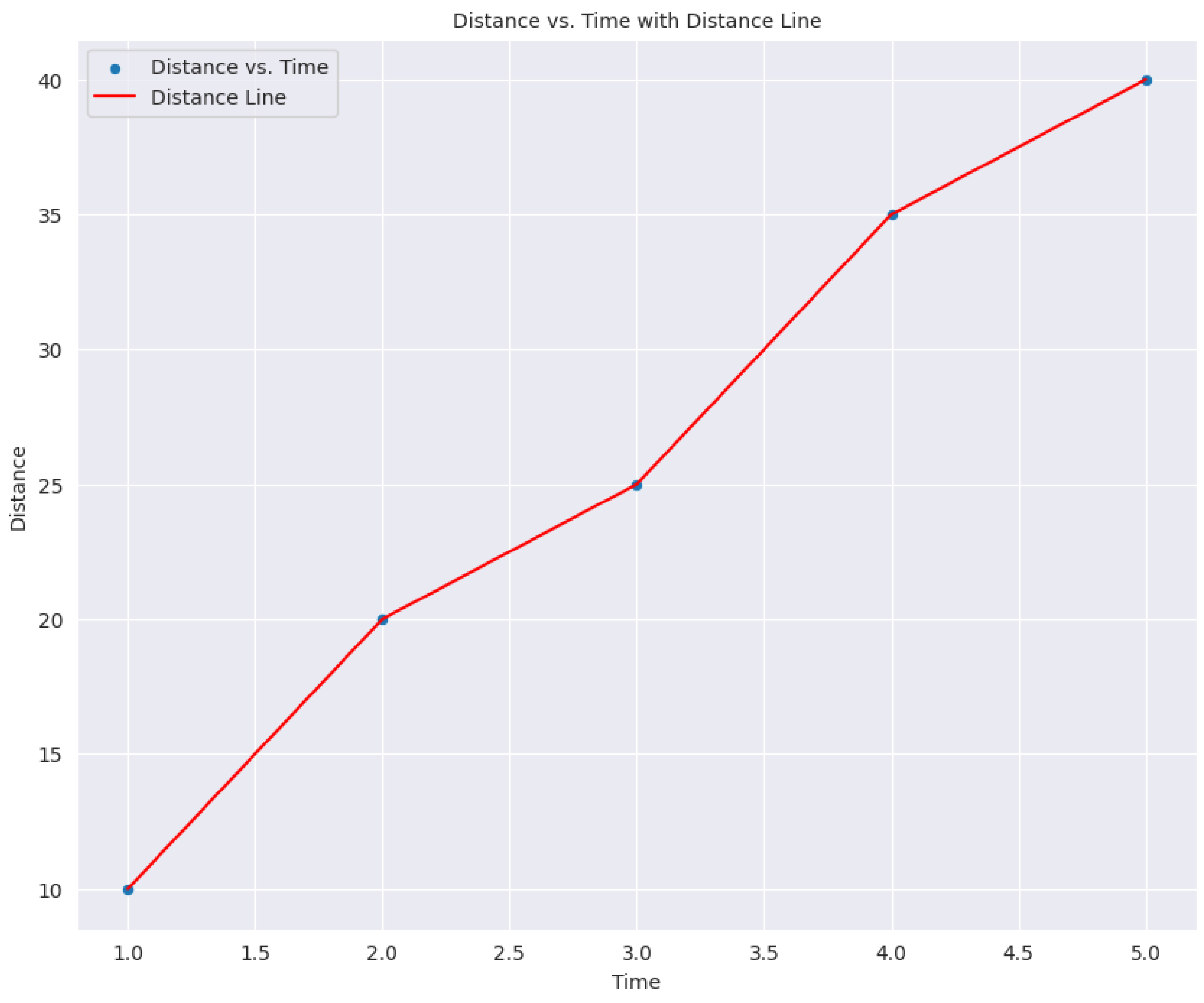
1.15 Derivatives and Partial Derivatives

Everything around us is changing, the universe is expanding, planets are moving, people are aging, even atoms don't stay in the same state, they are always moving or changing. Everything is changing with time. So how do we measure it?

How things change?

Suppose we are going on a car trip with our family. The speed of the car is constantly changing. Similarly the temperature at any given point on a day is changing. The overall temperature of Earth is changing. So we need a way to measure that change. Let's take the example of a family trip. Suppose the overall journey of our trip looks like this:

```python
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_style('darkgrid')

params = {'legend.fontsize': 'medium',
          'figure.figsize': (10, 8),
          'figure.dpi': 100,
         'axes.labelsize': 'medium',
         'axes.titlesize':'medium',
         'xtick.labelsize':'medium',
         'ytick.labelsize':'medium'}
plt.rcParams.update(params)

# Sample data (replace this with your own data)
time = [1, 2, 3, 4, 5]
distance = [10, 20, 25, 35, 40]

# Create a scatter plot
sns.scatterplot(x=time, y=distance, label='Distance vs. Time')

# Create a line plot on top of the scatter plot
sns.lineplot(x=time, y=distance, color='red', label='Distance Line')

# Add labels and title
plt.xlabel('Time')
plt.ylabel('Distance')
plt.title('Distance vs. Time with Distance Line')

# Show legend
plt.legend()

# Show the plot
plt.show()
```



1.15.1 Average vs Instantaneous rate of change

If we look at the graph, we can see that the car covered 40 miles in 5 hours. Now the average speed or the average rate of change will be the total distance divided by total time of the whole journey which in case is

$$\frac{\text{Total Distance}}{\text{Total time taken}} = \frac{40}{5} = 8 \text{ miles per hour}$$

But what if we want to find the rate of change at any given time, this is where the instantaneous rate of change comes into play. The instantaneous rate of change is given by the change at any given time or point. Take the example of a speedometer which gives you the change in speed every moment. We can also calculate the instantaneous change using a graph using the concept of a slope.

1.15.2 Slope of a line

Slope of a line is simply defined as the rate of change in the vertical direction due to rate of change in the horizontal direction or simply

$$\text{Slope } = \frac{\text{Rate of change in } y}{\text{Rate of change in } x} = \frac{\Delta y}{\Delta x} = \frac{y2 - y1}{x2 - x1}$$

But what about the instantaneous rate of change? Well, if you look at a curved line, the slope will be different for different points.

This instantaneous rate of change at any point is called the derivative at that point and is defined as:

$$\frac{\text{Rate of change in } y}{\text{Very small change in } x} = \frac{dy}{dx} = \frac{d}{dx}(f(x))$$

1.15.3 Derivative Explained

The derivative of a function is related to its rate of change. The rate of change tells you how much the output of the function changes when a change is done to the input. It is calculated as the ratio between a change in the output and the corresponding change in the input.

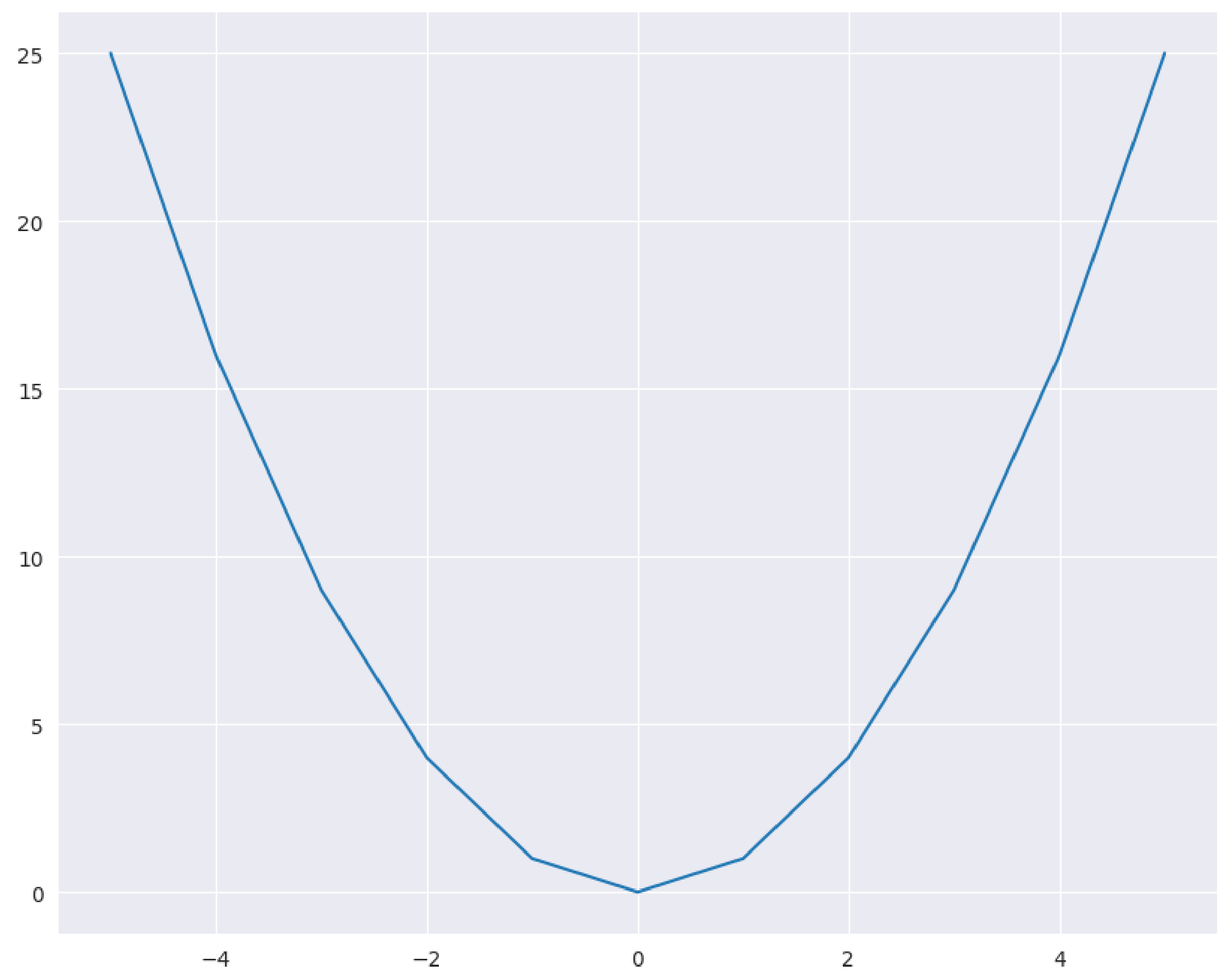
Graphically, it is the slope of the tangent at a given point of the function.

Let’s take an example. Suppose we have a function like this:

$$y = f(x) = x^2$$

The graph of this function looks like this

```
# seaborn grah of x squared function
x = [-5, -4, -3, -2, -1 , 0 , 1, 2, 3, 4, 5]
func = lambda i: i**2

y = [func(i) for i in x]

sns.lineplot(x=x, y=y)
plt.show()
```



Now let's say we want to find the slope or instantaneous change in y due to x. Let's say x change from x to x+h then:

$$y = f(x) = x^2$$

Lets say y changes due change in x so

$$x_2 = x + h$$

then

$$y_2 = f(x + h)$$

$$\begin{aligned} m &= \frac{y_2 - y_1}{x_2 - x_1} \\ &= \frac{f(x+h) - f(x)}{(x+h) - x} \\ &= \frac{(x+h)^2 - x^2}{(x+h) - x} \\ &= \frac{x^2 + h^2 + 2xh - x^2}{x - x - h} \\ &= \frac{h(2x+h)}{h} \\ &= 2x \end{aligned}$$

since h -> 0 so h will be zero.

1.15.4 Partial Derivatives

Now considering the derivative of a function with a single input, a partial derivative of a function is just the derivative of a function with multiple inputs with respect to a single variable i.e. the change in that function caused by the change in a single input. Let's suppose a function

$$f(x, y) = x^2 y + \sin y$$

Now we cannot find the derivative of this function directly since it depends on two inputs. So what we do is we find the derivative of this function assuming that one of the inputs is constant. Or simply that what change in the function is caused by the slight change in that single input. Let's find the partial derivative of this function with respect to both inputs one by 1

To calculate the partial derivatives of the given function $(f(x, y) = x^2 y + \sin(y))$ with respect to x and y, we will find the derivative of each term separately and then combine them using the rules of partial differentiation.

$$\frac{\partial(f(x,y))}{\partial x} = \lim_{h \to 0} \frac{f(x+h,y) - f(x,y)}{h}$$

Partial derivative with respect to x

denoted as $\frac{\partial f}{\partial x}$:

We treat y as a constant when taking the derivative with respect to x. Therefore, we differentiate $x^2 y$ with respect to x while keeping y constant:

$$\frac{\partial}{\partial x}(x^2 y) = 2xy$$

The derivative of $\sin(y)$ with respect to x is 0 because $\sin(y)$ does not depend on x. So, $\frac{\partial}{\partial x}(\sin(y)) = 0$.

Now, we can combine these partial derivatives:

$$\frac{\partial f}{\partial x} = 2xy + 0 = 2xy$$

So, the partial derivative of $f(x, y)$ with respect to xis2xy$.

$$\frac{\partial f}{\partial x} = 2xy$$

Partial derivative with respect to y

denoted as $\frac{\partial f}{\partial y}$

Now, we treat x as a constant when taking the derivative with respect to y. Therefore, we differentiate $(x^2 y$ with respect to y while keeping x constant:

$$\frac{\partial}{\partial y}(x^2 y) = x^2$$

The derivative of $\sin(y)$ with respect to y is $\cos(y)$, so $\frac{\partial}{\partial y}(\sin(y)) = \cos(y)$.

Now, we can combine these partial derivatives:

$$\frac{\partial f}{\partial y} = x^2 + \cos(y)$$

So, the partial derivative of $f(x, y)$ with respect to y is $x^2 + \cos(y)$.

In summary, the partial derivatives of the given function $f(x, y) = x^2y + \sin(y)$ are:

$$\frac{\partial f}{\partial x} = 2xy$$

and

$$\frac{\partial f}{\partial y} = x^2 + \cos(y)$$

Now if we want to find the partial derivative of the function at the point (-1, 2). We can just chug in values in both partial equations and find the change as:

$$\frac{\partial f}{\partial x}(-1, 2) = 2xy = 2 \cdot (-1) \cdot (2) = -4$$

Similarly

$$\frac{\partial f}{\partial y}(-1, 2) = x^2 + \cos y = (-1)^2 + \cos(2) = 0.5838$$

So we say that the change in the function with respect to input x is -4 times and with respect to y is +0.5838.

This means that the function is more sensitive to x than to y.

E.g https://www.youtube.com/watch?v=dfvnCHqzK54

https://www.youtube.com/watch?v=wqPt3qjB6uA&ab_channel=Dr.DataScience

https://www.youtube.com/watch?v=sIX_9n-1UbM

1.15.5 Derivative rules

Constant Rule

If you have a number (like 5 or 10) all by itself, its derivative is always 0. This means it doesn't change when you take the derivative.

A constant represents a horizontal line on the graph, which has no slope (i.e., it's perfectly flat). Therefore, the rate of change (derivative) is zero.

Proof: Let $f(x) = c$, where c is a constant. Then, by definition, the derivative of $f(x)$ is

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} = \lim_{h \to 0} \frac{c - c}{h} = \lim_{h \to 0} \frac{0}{h} = 0$$

Power Rule

If you have a number with an exponent (like x^2 or x^3), you can bring the exponent down and subtract 1 from it. For example, if you have x^2, the derivative is 2x because 2 times x^1 is 2x.

The derivative of x^n with respect to x is nx^{n-1}, where n is a constant. This rule is derived using the limit definition of the derivative and the binomial theorem.

Proof: Start with the limit definition of the derivative

$$\frac{d}{dx}(x^n) = \lim_{h \to 0} \frac{(x+h)^n - x^n}{h}$$

Use the binomial theorem to expand $(x+h)^n$ \$$(x+h)^n = x^n + nx^{n-1}h +$ higher order terms in h\$ Substitute this into the limit definition

$$\frac{d}{dx}(x^n) = \lim_{h \to 0} \frac{(x^n + nx^{n-1}h + \text{higher order terms in } h) - x^n}{h}$$

Cancel the x^n terms and divide by h

$$\frac{d}{dx}(x^n) = \lim_{h \to 0} \frac{nx^{n-1}h + \text{higher order terms in } h}{h}$$

Simplify and take the limit as h approaches 0: \$$\frac{d}{dx}(x^n) = nx^{n-1}$\$

Sum Rule

If you're adding or subtracting two things, like f(x) + g(x) or f(x) - g(x), you can take the derivative of each thing separately and keep them separate. For example, if you have $3x^2 + 4x$, you can find the derivative of $3x^2$ (which is 6x) and the derivative of 4x (which is 4), and then you keep them together as 6x + 4.

Chain Rule

Sometimes, you have functions inside of functions. Imagine you have $f(g(x))$. To find the derivative of that, you first find the derivative of the outer function (f) and then the derivative of the inner function (g). You multiply them together. It's like doing things step by step.

The chain rule is a fundamental rule in calculus that allows us to find the derivative of a composite function. In other words, it tells us how to differentiate a function that is composed of two or more functions. The chain rule is often stated as follows:

If f(u) and g(x) are differentiable functions, then the derivative of their composition f(g(x)) is given by:

$$\frac{d}{dx}[f(g(x))] = f'(g(x)) \cdot g'(x)$$

Here's an explanation of the chain rule with an example and a proof:

Explanation:

The chain rule essentially states that to find the derivative of a composite function, you first take the derivative of the outer function with respect to its inner function and then multiply it by the derivative of the inner function with respect to the variable of interest (in this case, x).

Example

Let's use an example to illustrate the chain rule. Consider the function $y = f(u) = u^2$ and $u = g(x) = x^3$. We want to find the derivative of y with respect to x which is $\frac{dy}{dx}$.

1. Find $\frac{dy}{du}$: This is the derivative of the outer function $f(u)$ with respect to its inner function u, which is $2u$.
2. Find $\frac{du}{dx}$: This is the derivative of the inner function $g(x)$ with respect to x, which is $3x^2$.
3. Apply the chain rule:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} = (2u) \cdot (3x^2) = 2 \cdot 3x^2 u = 6x^2 u$$

Now, substitute back $u = x^3$:

$$\frac{dy}{dx} = 6x^2 \cdot (x^3) = 6x^5$$

So, the derivative of $y = x^6$ with respect to x is $6x^5$.

Proof:

The proof of the chain rule relies on the definition of the derivative and the limit concept. Let's prove it step by step:

1. Start with the definition of the derivative of a function:

$$\frac{d}{dx}[f(u)] = \lim_{h \to 0} \frac{f(u+h) - f(u)}{h}$$

1. Now, we want to find the derivative of the composition $f(g(x))$. Let $v = g(x)$. So, we can write:

$$\frac{d}{dx}[f(g(x))] = \frac{d}{dx}[f(v)]$$

1. Using the definition of the derivative for $f(v)$, we have:

$$\frac{d}{dx}[f(v)] = \lim_{h \to 0} \frac{f(v+h) - f(v)}{h}$$

2. Rewrite $v + h$ as $g(x + h)$ because $v = g(x)$:

$$\frac{d}{dx}[f(v)] = \lim_{h \to 0} \frac{f(g(x+h)) - f(g(x))}{h}$$

3. Now, we can see that this is precisely the definition of the derivative of $f(g(x))$. So, we've shown that:

$$\frac{d}{dx}[f(g(x))] = \lim_{h \to 0} \frac{f(g(x+h)) - f(g(x))}{h}$$

1. And this can be simplified to:

$$\frac{d}{dx}[f(g(x))] = f'(g(x)) \cdot g'(x)$$

So, the chain rule is proven. It tells us how to find the derivative of a composite function by considering the derivatives of its components.

1.15.6 Backpropagation Chain rule

The chain rule is a crucial concept in neural network backpropagation, which is the algorithm used to train neural networks. It allows us to efficiently calculate the gradients of the loss function with respect to the network's parameters (weights and biases) by decomposing the overall gradient into smaller gradients associated with each layer of the network. Let's explain how the chain rule is used in neural network backpropagation with an example.

Suppose we have a simple feedforward neural network with one hidden layer. Here's a simplified network architecture:

- Input layer with n neurons
- Hidden layer with m neurons
- Output layer with k neurons

The network has weights $W^{(1)}$ for the connections between the input and hidden layers and weights $W^{(2)}$ for the connections between the hidden and output layers.

The forward pass of the network involves the following steps:

Compute the weighted sum and apply an activation function to the hidden layer:

$$z^{(1)} = XW^{(1)} + b^{(1)}$$

$$a^{(1)} = \sigma(z^{(1)})$$

where X is the input, $W^{(1)}$ are the weights of the first layer, $b^{(1)}$ are the biases of the first layer, $\sigma(\cdot)$ is the activation function (e.g., sigmoid or ReLU), and $a^{(1)}$ is the output of the hidden layer.

Compute the weighted sum and apply an activation function to the output layer:

$$z^{(2)} = a^{(1)}W^{(2)} + b^{(2)}$$

$$a^{(2)} = \sigma(z^{(2)})$$

where $W^{(2)}$ are the weights of the second (output) layer, $b^{(2)}$ are the biases of the second layer, and $a^{(2)}$ is the final output of the network.

Now, let's assume we have a loss function L that measures the error between the predicted output $a^{(2)}$ and the true target values Y. The goal of backpropagation is to update the network's weights and biases to minimize this loss.

To do this, we need to compute the gradients of the loss with respect to the network's parameters. The chain rule comes into play during this step. We calculate the gradients layer by layer, propagating the gradient backward through the network:

1. Compute the gradient of the loss with respect to the output layer's activations: $\frac{\partial L}{\partial a^{(2)}}$
2. Use the chain rule to calculate the gradient of the loss with respect to the output layer's weighted sum ($z^{(2)}$): $\frac{\partial L}{\partial z^{(2)}} = \frac{\partial L}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}}$
3. Compute the gradient of the loss with respect to the second layer's weights and biases ($W^{(2)}$ and $b^{(2)}$): $\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial W^{(2)}} \frac{\partial L}{\partial b^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial b^{(2)}}$
4. Use the chain rule again to calculate the gradient of the loss with respect to the hidden layer's activations ($a^{(1)}$): $\frac{\partial L}{\partial a^{(1)}} = \frac{\partial L}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial a^{(1)}}$
5. Compute the gradient of the loss with respect to the hidden layer's weighted sum ($z^{(1)}$): $\frac{\partial L}{\partial z^{(1)}} = \frac{\partial L}{\partial a^{(1)}} \cdot \frac{\partial a^{(1)}}{\partial z^{(1)}}$
6. Finally, calculate the gradient of the loss with respect to the first layer's weights and biases ($W^{(1)}$ and $b^{(1)}$): $\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial z^{(1)}} \cdot \frac{\partial z^{(1)}}{\partial W^{(1)}} \frac{\partial L}{\partial b^{(1)}} = \frac{\partial L}{\partial z^{(1)}} \cdot \frac{\partial z^{(1)}}{\partial b^{(1)}}$

The chain rule allows us to compute these gradients efficiently by breaking down the overall gradient into smaller gradients associated with each layer. Once we have these gradients, we can use them to update the network's weights and biases using optimization algorithms like gradient descent. This iterative process of forward and backward passes, driven by the chain rule, is how neural networks are trained to learn from data.

```
# Importing Required Libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import pandas as pd
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt

%matplotlib inline
```

```
#Load the dataset
data = load_iris()
data
```

```
{'data': array([[5.1, 3.5, 1.4, 0.2],
        [4.9, 3. , 1.4, 0.2],
        [4.7, 3.2, 1.3, 0.2],
        [4.6, 3.1, 1.5, 0.2],
        [5. , 3.6, 1.4, 0.2],
```

(continues on next page)

(continued from previous page)

```
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1],
       [5.4, 3.7, 1.5, 0.2],
       [4.8, 3.4, 1.6, 0.2],
       [4.8, 3. , 1.4, 0.1],
       [4.3, 3. , 1.1, 0.1],
       [5.8, 4. , 1.2, 0.2],
       [5.7, 4.4, 1.5, 0.4],
       [5.4, 3.9, 1.3, 0.4],
       [5.1, 3.5, 1.4, 0.3],
       [5.7, 3.8, 1.7, 0.3],
       [5.1, 3.8, 1.5, 0.3],
       [5.4, 3.4, 1.7, 0.2],
       [5.1, 3.7, 1.5, 0.4],
       [4.6, 3.6, 1. , 0.2],
       [5.1, 3.3, 1.7, 0.5],
       [4.8, 3.4, 1.9, 0.2],
       [5. , 3. , 1.6, 0.2],
       [5. , 3.4, 1.6, 0.4],
       [5.2, 3.5, 1.5, 0.2],
       [5.2, 3.4, 1.4, 0.2],
       [4.7, 3.2, 1.6, 0.2],
       [4.8, 3.1, 1.6, 0.2],
       [5.4, 3.4, 1.5, 0.4],
       [5.2, 4.1, 1.5, 0.1],
       [5.5, 4.2, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.2],
       [5. , 3.2, 1.2, 0.2],
       [5.5, 3.5, 1.3, 0.2],
       [4.9, 3.6, 1.4, 0.1],
       [4.4, 3. , 1.3, 0.2],
       [5.1, 3.4, 1.5, 0.2],
       [5. , 3.5, 1.3, 0.3],
       [4.5, 2.3, 1.3, 0.3],
       [4.4, 3.2, 1.3, 0.2],
       [5. , 3.5, 1.6, 0.6],
       [5.1, 3.8, 1.9, 0.4],
       [4.8, 3. , 1.4, 0.3],
       [5.1, 3.8, 1.6, 0.2],
       [4.6, 3.2, 1.4, 0.2],
       [5.3, 3.7, 1.5, 0.2],
       [5. , 3.3, 1.4, 0.2],
       [7. , 3.2, 4.7, 1.4],
       [6.4, 3.2, 4.5, 1.5],
       [6.9, 3.1, 4.9, 1.5],
       [5.5, 2.3, 4. , 1.3],
       [6.5, 2.8, 4.6, 1.5],
       [5.7, 2.8, 4.5, 1.3],
       [6.3, 3.3, 4.7, 1.6],
```

(continues on next page)

(continued from previous page)

```
        [4.9, 2.4, 3.3, 1. ],
        [6.6, 2.9, 4.6, 1.3],
        [5.2, 2.7, 3.9, 1.4],
        [5. , 2. , 3.5, 1. ],
        [5.9, 3. , 4.2, 1.5],
        [6. , 2.2, 4. , 1. ],
        [6.1, 2.9, 4.7, 1.4],
        [5.6, 2.9, 3.6, 1.3],
        [6.7, 3.1, 4.4, 1.4],
        [5.6, 3. , 4.5, 1.5],
        [5.8, 2.7, 4.1, 1. ],
        [6.2, 2.2, 4.5, 1.5],
        [5.6, 2.5, 3.9, 1.1],
        [5.9, 3.2, 4.8, 1.8],
        [6.1, 2.8, 4. , 1.3],
        [6.3, 2.5, 4.9, 1.5],
        [6.1, 2.8, 4.7, 1.2],
        [6.4, 2.9, 4.3, 1.3],
        [6.6, 3. , 4.4, 1.4],
        [6.8, 2.8, 4.8, 1.4],
        [6.7, 3. , 5. , 1.7],
        [6. , 2.9, 4.5, 1.5],
        [5.7, 2.6, 3.5, 1. ],
        [5.5, 2.4, 3.8, 1.1],
        [5.5, 2.4, 3.7, 1. ],
        [5.8, 2.7, 3.9, 1.2],
        [6. , 2.7, 5.1, 1.6],
        [5.4, 3. , 4.5, 1.5],
        [6. , 3.4, 4.5, 1.6],
        [6.7, 3.1, 4.7, 1.5],
        [6.3, 2.3, 4.4, 1.3],
        [5.6, 3. , 4.1, 1.3],
        [5.5, 2.5, 4. , 1.3],
        [5.5, 2.6, 4.4, 1.2],
        [6.1, 3. , 4.6, 1.4],
        [5.8, 2.6, 4. , 1.2],
        [5. , 2.3, 3.3, 1. ],
        [5.6, 2.7, 4.2, 1.3],
        [5.7, 3. , 4.2, 1.2],
        [5.7, 2.9, 4.2, 1.3],
        [6.2, 2.9, 4.3, 1.3],
        [5.1, 2.5, 3. , 1.1],
        [5.7, 2.8, 4.1, 1.3],
        [6.3, 3.3, 6. , 2.5],
        [5.8, 2.7, 5.1, 1.9],
        [7.1, 3. , 5.9, 2.1],
        [6.3, 2.9, 5.6, 1.8],
        [6.5, 3. , 5.8, 2.2],
        [7.6, 3. , 6.6, 2.1],
        [4.9, 2.5, 4.5, 1.7],
        [7.3, 2.9, 6.3, 1.8],
        [6.7, 2.5, 5.8, 1.8],
```

(continues on next page)

(continued from previous page)

```
        [7.2, 3.6, 6.1, 2.5],
        [6.5, 3.2, 5.1, 2. ],
        [6.4, 2.7, 5.3, 1.9],
        [6.8, 3. , 5.5, 2.1],
        [5.7, 2.5, 5. , 2. ],
        [5.8, 2.8, 5.1, 2.4],
        [6.4, 3.2, 5.3, 2.3],
        [6.5, 3. , 5.5, 1.8],
        [7.7, 3.8, 6.7, 2.2],
        [7.7, 2.6, 6.9, 2.3],
        [6. , 2.2, 5. , 1.5],
        [6.9, 3.2, 5.7, 2.3],
        [5.6, 2.8, 4.9, 2. ],
        [7.7, 2.8, 6.7, 2. ],
        [6.3, 2.7, 4.9, 1.8],
        [6.7, 3.3, 5.7, 2.1],
        [7.2, 3.2, 6. , 1.8],
        [6.2, 2.8, 4.8, 1.8],
        [6.1, 3. , 4.9, 1.8],
        [6.4, 2.8, 5.6, 2.1],
        [7.2, 3. , 5.8, 1.6],
        [7.4, 2.8, 6.1, 1.9],
        [7.9, 3.8, 6.4, 2. ],
        [6.4, 2.8, 5.6, 2.2],
        [6.3, 2.8, 5.1, 1.5],
        [6.1, 2.6, 5.6, 1.4],
        [7.7, 3. , 6.1, 2.3],
        [6.3, 3.4, 5.6, 2.4],
        [6.4, 3.1, 5.5, 1.8],
        [6. , 3. , 4.8, 1.8],
        [6.9, 3.1, 5.4, 2.1],
        [6.7, 3.1, 5.6, 2.4],
        [6.9, 3.1, 5.1, 2.3],
        [5.8, 2.7, 5.1, 1.9],
        [6.8, 3.2, 5.9, 2.3],
        [6.7, 3.3, 5.7, 2.5],
        [6.7, 3. , 5.2, 2.3],
        [6.3, 2.5, 5. , 1.9],
        [6.5, 3. , 5.2, 2. ],
        [6.2, 3.4, 5.4, 2.3],
        [5.9, 3. , 5.1, 1.8]]),
 'target': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]),
 'frame': None,
 'target_names': array(['setosa', 'versicolor', 'virginica'], dtype='<U10'),
 'DESCR': '.. _iris_dataset:\n\nIris plants dataset\n--------------------\n\n**Data Set␣
↪Characteristics:**\n\n:Number of Instances: 150 (50 in each of three classes)\n:Number␣
```

(continues on next page)

(continued from previous page)

```
↪of Attributes: 4 numeric, predictive attributes and the class\n:Attribute Information:\
↪n    - sepal length in cm\n    - sepal width in cm\n    - petal length in cm\n    -␣
↪petal width in cm\n    - class:\n            - Iris-Setosa\n            - Iris-
↪Versicolour\n            - Iris-Virginica\n\n:Summary Statistics:\n\n=============␣
↪==== ==== ======= ===== ====================\n                Min  Max   Mean    SD  ␣
↪Class Correlation\n============= ==== ==== ======= ===== ====================\nsepal␣
↪length:   4.3  7.9   5.84   0.83    0.7826\nsepal width:    2.0  4.4   3.05   0.43   -
↪0.4194\npetal length:   1.0  6.9   3.76   1.76    0.9490  (high!)\npetal width:    0.1␣
↪ 2.5   1.20   0.76    0.9565  (high!)\n============= ==== ==== ======= =====␣
↪====================\n\n:Missing Attribute Values: None\n:Class Distribution: 33.3%␣
↪for each of 3 classes.\n:Creator: R.A. Fisher\n:Donor: Michael Marshall (MARSHALL
↪%PLU@io.arc.nasa.gov)\n:Date: July, 1988\n\nThe famous Iris database, first used by␣
↪Sir R.A. Fisher. The dataset is taken\nfrom Fisher\'s paper. Note that it\'s the same␣
↪as in R, but not as in the UCI\nMachine Learning Repository, which has two wrong data␣
↪points.\n\nThis is perhaps the best known database to be found in the\npattern␣
↪recognition literature.  Fisher\'s paper is a classic in the field and\nis referenced␣
↪frequently to this day.  (See Duda & Hart, for example.)  The\ndata set contains 3␣
↪classes of 50 instances each, where each class refers to a\ntype of iris plant.  One␣
↪class is linearly separable from the other 2; the\nlatter are NOT linearly separable␣
↪from each other.\n\n|details-start|\n**References**\n|details-split|\n\n- Fisher, R.A.
↪"The use of multiple measurements in taxonomic problems"\n  Annual Eugenics, 7, Part␣
↪II, 179-188 (1936); also in "Contributions to\n  Mathematical Statistics" (John Wiley,␣
↪NY, 1950).\n- Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene␣
↪Analysis.\n  (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.\n-␣
↪Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System\n  Structure and␣
↪Classification Rule for Recognition in Partially Exposed\n  Environments".  IEEE␣
↪Transactions on Pattern Analysis and Machine\n  Intelligence, Vol. PAMI-2, No. 1, 67-
↪71.\n- Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions\n ␣
↪on Information Theory, May 1972, 431-433.\n- See also: 1988 MLC Proceedings, 54-64. ␣
↪Cheeseman et al"s AUTOCLASS II\n  conceptual clustering system finds 3 classes in the␣
↪data.\n- Many, many more ...\n\n|details-end|\n',
 'feature_names': ['sepal length (cm)',
  'sepal width (cm)',
  'petal length (cm)',
  'petal width (cm)'],
 'filename': 'iris.csv',
 'data_module': 'sklearn.datasets.data'}
```

```python
# Making it a 2 class problem
X_data = data["data"][:100]
Y_data = data["target"][:100]
len(X_data)
```

```
100
```

```python
# Splitting the data into training and testing with 80 and 20%
X_train, X_test, Y_train, Y_test = train_test_split(X_data, Y_data, test_size = 0.2)
print(X_train.shape, X_test.shape)
```

```
(80, 4) (20, 4)
```

```
# Making their tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32).t()
Y_train_tensor = torch.tensor(Y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32).t()
Y_test_tensor = torch.tensor(Y_test, dtype=torch.float32)
X_train_tensor.shape[1]
```

```
80
```

```
def init_params(n_x, n_h, n_y):
  '''
    This function is used to initializw the weights for the NN.
    n_x : input units
    n_h : hidden units
    n_y : output units
    It returns a dictionary which contains all the parameters
  '''
  W1 = torch.rand(n_x, n_h)
  b1 = torch.rand(n_h, 1)
  W2 = torch.rand(n_y, n_h)
  b2 = torch.rand(n_y, 1)

  params = {
      'W1': W1,
      'b1': b1,
      'W2': W2,
      'b2': b2
  }
  return params

def compute_cost(y_pred, y_actual):
  '''
    Uses the binary cross entropy loss to compute the cost
  '''
  return -(y_pred.log()* y_actual + (1-y_actual)*(1-y_pred).log()).mean()
  return -1/len(y_pred) * (y_actual * torch.log(y_pred) + (1 - y_actual) * torch.log(1 -␣
↪y_pred)).sum()

def forward_propagation(params, x_input):
  '''
    Performs the forward propagation step. Uses the parameters to predict the output A2
  '''
  #Extractng the parameters
  W1 = params['W1']
  b1 = params['b1']
  W2 = params['W2']
  b2 = params['b2']

  # Computing the first layer
  Z1 = torch.mm(W1.t(), x_input) + b1
  A1 = torch.sigmoid(Z1)
```

(continues on next page)

(continued from previous page)

```
    # Computing the second layer
    Z2 = torch.mm(W2, A1) + b2
    A2 = torch.sigmoid(Z2)

    # Returning the data
    data = {
        'Z1' : Z1,
        'A1' : A1,
        'Z2' : Z2,
        'A2' : A2
    }

    return A2, data

def back_propagation(params, data, x_input, y_input, learning_rate):
    '''
      Performs the back propagation step. Computes the gradients and updates the parameters
    '''
    m = x_input.shape[1]

    # Extracting the parameters
    W1 = params['W1']
    W2 = params['W2']
    b1 = params['b1']
    b2 = params['b2']

    # Extrcting the required predictions of first and second layers
    A1 = data['A1']
    A2 = data['A2']

    # Calculating the Gradients
    dZ2 = A2 - y_input
    dW2 = 1/m*(torch.mm(dZ2, A1.t()))
    db2 = 1/m*(torch.sum(dZ2, keepdims=True, axis=1))
    dZ1 = torch.mm(W2.t(), dZ2)*(1-torch.pow(A1,2))
    dW1 = 1/m*(torch.mm(dZ1, x_input.t()))
    db1 = 1/m*(torch.sum(dZ1, keepdims=True, axis=1))

    # Updating the parameters
    W1 -= learning_rate*dW1
    W2 -= learning_rate*dW2
    b1 -= learning_rate*db1
    b2 -= learning_rate*db2

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

```
def model(x_input, y_input, learning_rate = 0.01, no_iterations = 20000):
  '''
    Putting everything together and making the model
  '''
  n_x = x_input.shape[0]
  n_h = 4
  n_y = 1
  parameters = init_params(n_x, n_h, n_y)
  costs, iterations = [], []
  for i in range(no_iterations):
    A2, data = forward_propagation(parameters, x_input)

    cost = compute_cost(A2, y_input)
    cost_torch = F.binary_cross_entropy(A2, y_input.view(1, 80))

    parameters = back_propagation(parameters, data, x_input, y_input, learning_rate)

    if i%(no_iterations/20) == 0:
      print(f'Cost at iteration {i} is {cost}')
      print(f'Cost_torch at iteration {i} is {cost_torch.item()}')

    costs.append(cost)
    iterations.append(i)
  return parameters, costs, iterations
```

```
parameters, costs, iterations = model(X_train_tensor, Y_train_tensor, 1e-4, 20000)
plt.plot(iterations, costs)
```

```
Cost at iteration 0 is 1.4469505548477173
Cost_torch at iteration 0 is 1.4469505548477173
```

```
Cost at iteration 1000 is 1.3540798425674438
Cost_torch at iteration 1000 is 1.3540798425674438
```

```
Cost at iteration 2000 is 1.267162799835205
Cost_torch at iteration 2000 is 1.267162799835205
```

```
Cost at iteration 3000 is 1.1868066787719727
Cost_torch at iteration 3000 is 1.1868066787719727
```

```
Cost at iteration 4000 is 1.1134998798370361
Cost_torch at iteration 4000 is 1.1134998798370361
```

```
Cost at iteration 5000 is 1.0475634336471558
Cost_torch at iteration 5000 is 1.0475634336471558
```

```
Cost at iteration 6000 is 0.9891158938407898
Cost_torch at iteration 6000 is 0.9891158938407898
```

```
Cost at iteration 7000 is 0.9380590319633484
Cost_torch at iteration 7000 is 0.9380590319633484
```

```
Cost at iteration 8000 is 0.8940895199775696
Cost_torch at iteration 8000 is 0.8940895199775696
```

```
Cost at iteration 9000 is 0.8567306399345398
Cost_torch at iteration 9000 is 0.8567306399345398
```

```
Cost at iteration 10000 is 0.8253803253173828
Cost_torch at iteration 10000 is 0.8253803253173828
```

```
Cost at iteration 11000 is 0.7993641495704651
Cost_torch at iteration 11000 is 0.7993641495704651
```

```
Cost at iteration 12000 is 0.7779852151870728
Cost_torch at iteration 12000 is 0.7779852151870728
```

```
Cost at iteration 13000 is 0.7605642080307007
Cost_torch at iteration 13000 is 0.7605642080307007
```

```
Cost at iteration 14000 is 0.7464694976806641
Cost_torch at iteration 14000 is 0.7464694976806641
```

```
Cost at iteration 15000 is 0.7351330518722534
Cost_torch at iteration 15000 is 0.7351330518722534
```

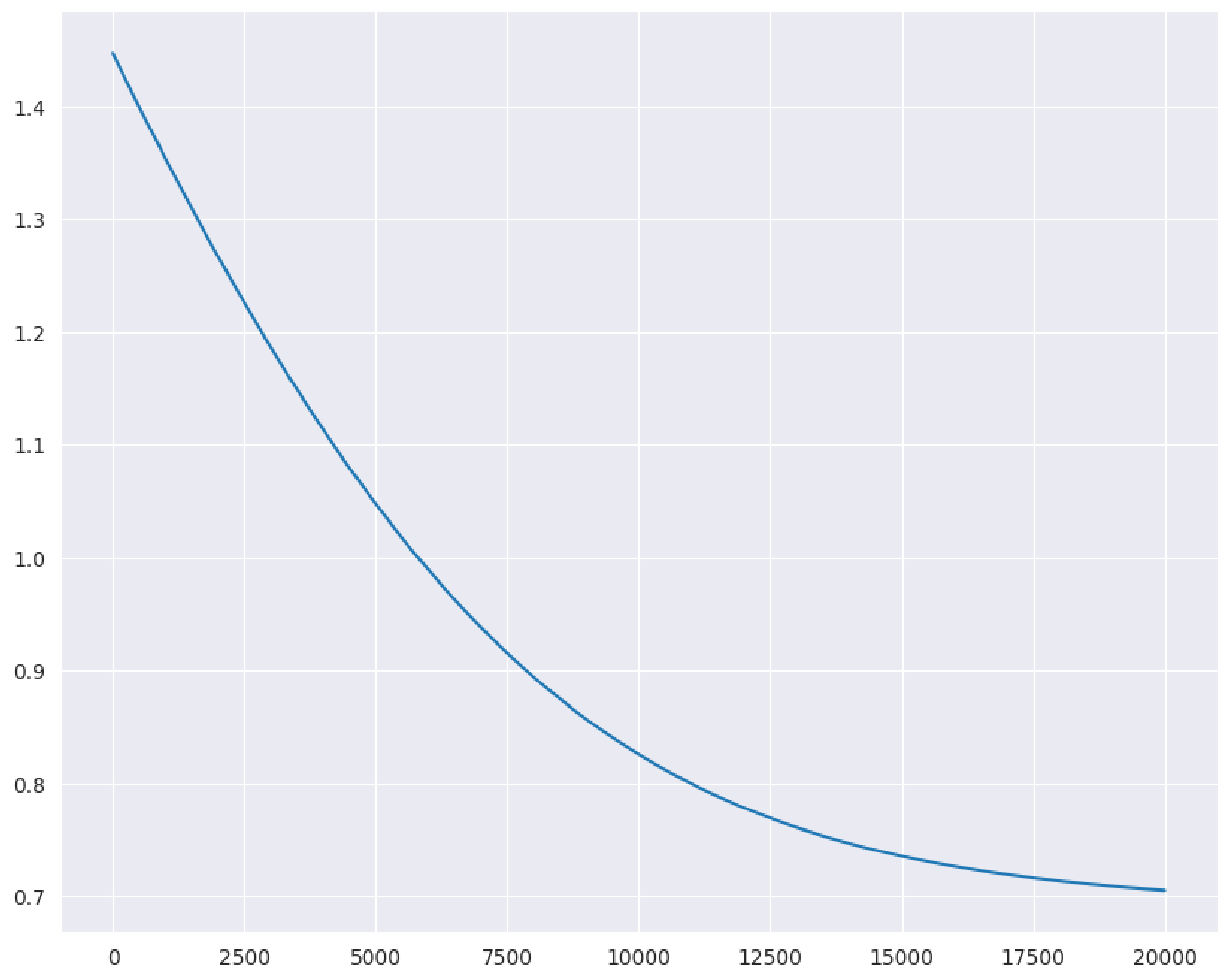
```
Cost at iteration 16000 is 0.7260592579841614
Cost_torch at iteration 16000 is 0.7260592579841614
```

```
Cost at iteration 17000 is 0.7188249230384827
Cost_torch at iteration 17000 is 0.7188249826431274
```

```
Cost at iteration 18000 is 0.7130751609802246
Cost_torch at iteration 18000 is 0.7130751609802246
```

```
Cost at iteration 19000 is 0.7085162401199341
Cost_torch at iteration 19000 is 0.7085162401199341
```

```
[<matplotlib.lines.Line2D at 0x7fc4584d7fd0>]
```



1.16 Algebra Introduction

This section introduces the basic concepts of algebra, including variables, constants, and functions

1.16.1 Functions

A function is a rule that takes one or more inputs and produces a single output. For example, the function $f(x) = x+1$ takes a single input x, adds one to it, and produces a single output. In algebra, functions are written using symbols and formulas. For example, the function $f(x) = x+1$ can be written as $f : x \to x+1$. The input to a function is called the **argument** or **input variable**. The output is called the **value** or **output variable**.

Functions are often written using the following notation:

$$y = f(x)$$

The notation above is read as "y equals f of x" or "y is a function of x". The notation above is useful because it allows us to define a function without specifying its name. For example, we can define a function f as follows:

$$f(x) = x^2$$

We can then use the function f to compute the square of any number. For example, $f(2) = 2^2 = 4$ and $f(3) = 3^2 = 9$.

$$\begin{aligned}\mathrm{f}(x) &= \sqrt{x+6}\\ \mathrm{f}(6) &= \sqrt{10+6}\\ \mathrm{f}(6) &= 4.0\end{aligned}$$

$$\begin{aligned}f(x) &= \frac{x-3}{x+2}\\ f(3) &= \frac{3-3}{3+2} = \frac{0}{5} = 0\end{aligned}$$

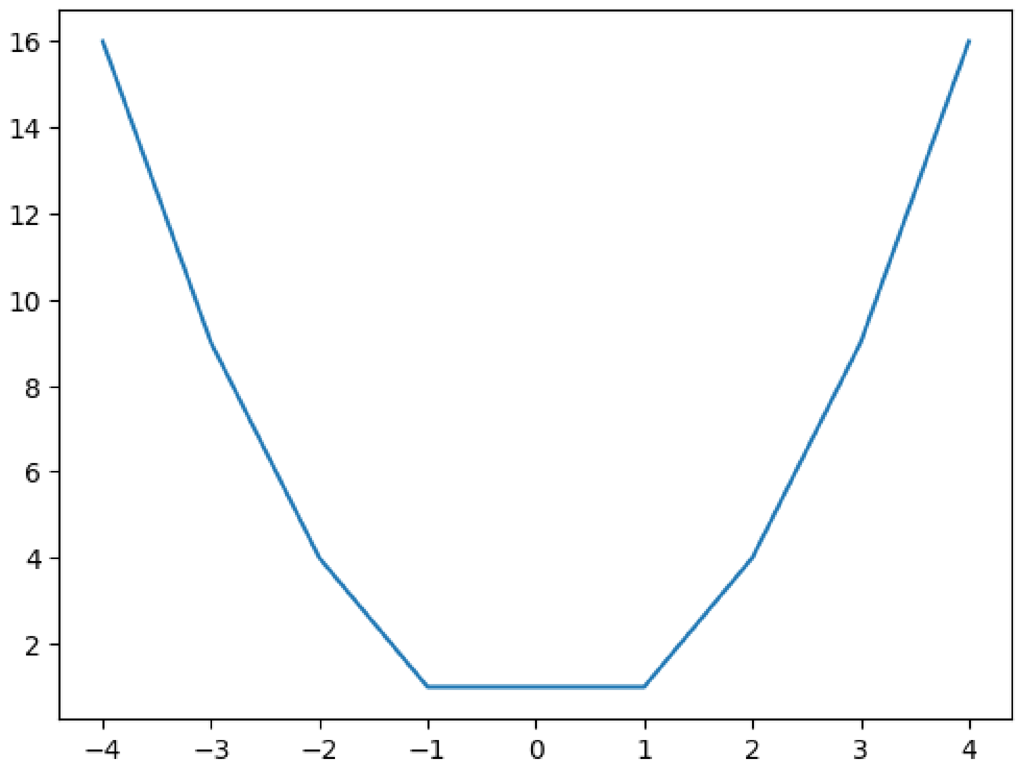
Domain and Range of a Function

The **domain** of a function is the set of all possible inputs to the function. The **range** of a function is the set of all possible outputs of the function. For example, the function $f(x) = x^2$ has a domain of all real numbers and a range of all non-negative real numbers. The domain of a function is often written as $D(f)$ and the range is often written as $R(f)$.

$$\begin{aligned}y &= f(x)\\ y &= x^2\end{aligned}$$

```
import seaborn as sb

func = lambda x: x ** 2

x = [-1,-2,-3, -4, 1, 2, 3, 4]
y = [func(i) for i in x]

sb.lineplot(x=x, y=y)
```

```
<Axes: >
```



Piecewise Functions

A piecewise function is a function that is defined by multiple sub-functions, each sub-function applying to a different interval of the main function's domain. For example, the function $f(x) = |x|$ is defined by two sub-functions:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

1.16.2 Expoents

An exponent is a number that indicates how many times a base number is multiplied by itself. For example, 2^3 is the same as $2 \times 2 \times 2$ and 2^4 is the same as $2 \times 2 \times 2 \times 2$. The number 2 is called the **base** and the number 3 is called the **exponent**. Exponents are often written using the following notation:

$$2^3 = 2 \times 2 \times 2 = 8$$

The notation above is read as "two to the power of three" or "two cubed".

Negative Exponents

A negative exponent indicates that the base number should be divided by itself a certain number of times. For example, 2^{-3} is the same as $\frac{1}{2^3}$ and 2^{-4} is the same as $\frac{1}{2^4}$. The number 2 is called the **base** and the number -3 is called the **exponent**. Negative exponents are often written using the following notation:

$$2^{-3} = \frac{1}{2^3} = \frac{1}{8}$$

The notation above is read as "two to the power of negative three" or "two to the power of minus three".

Fractional Exponents

A fractional exponent indicates that the base number should be multiplied by itself a certain number of times. For example, $2^{\frac{1}{2}}$ is the same as $\sqrt{2}$ and $2^{\frac{1}{3}}$ is the same as $\sqrt[3]{2}$. The number 2 is called the **base** and the number $\frac{1}{2}$ is called the **exponent**. Fractional exponents are often written using the following notation:

$$2^{\frac{1}{2}} = \sqrt{2} = 1.414213562373095$$

The notation above is read as "two to the power of one half" or "two to the power of one over two".

1.16.3 Logarithms

A logarithm is the inverse of an exponent. For example, the logarithm of 2^3 is 3. The logarithm of a number x to the base b is written as $\log_b(x)$. For example, $\log_2(8) = 3$ because $2^3 = 8$.

Common Logarithms

The common logarithm of a number x is the logarithm of x to the base 10. The common logarithm of x is written as $\log(x)$. For example, $\log(100) = 2$ because $10^2 = 100$.

Natural Logarithms

The natural logarithm of a number x is the logarithm of x to the base e. The natural logarithm of x is written as $\ln(x)$. For example, $\ln(100) = 4.60517$ because $e^{4.60517} = 100$.

1.16.4 Polynomials

A polynomial is an expression consisting of variables and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents.

For example, $x^2 + 2x + 1$ is a polynomial because it consists of the variables x and the coefficients 1 and 2.

The degree of a polynomial is the highest degree of its terms. For example, the polynomial $x^2 + 2x + 1$ has a degree of 2 because its highest degree term is x^2.

1.16.5 Proof by Induction

A proof by induction consists of two cases. The first, the base case, proves the statement for n = 0 without assuming any knowledge of other cases. The second case, the induction step, proves that if the statement holds for any given case n = k, then it must also hold for the next case n = k + 1. These two steps establish that the statement holds for every natural number n.

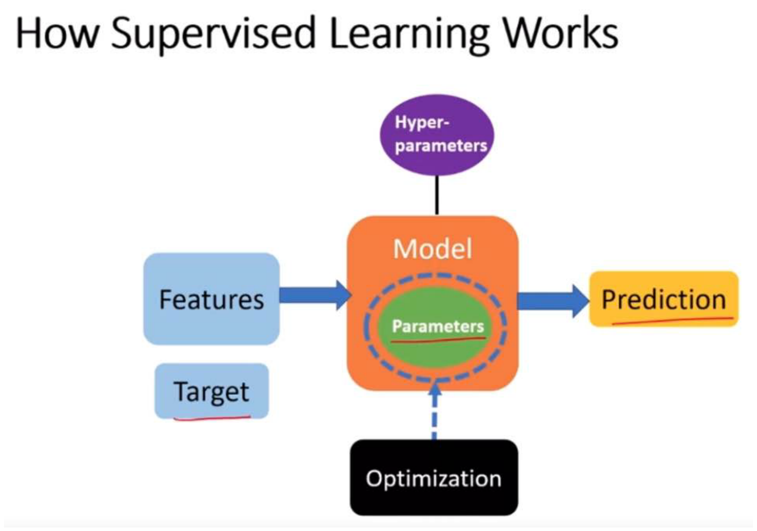
1.17 What is Machine Learning?

Statistical learning is about using many tools to understand data. These tools can be grouped into two types: supervised and unsupervised. Supervised learning means you create a model to predict or estimate an outcome based on inputs. This kind of problem is found in many areas like business, medicine, space science, and government policies. Unsupervised learning means you don't have a specific outcome you're looking for, but you still try to find patterns or relationships in the data.

Best book on Machine Learning

My notes are based on this book reference. https://www.statlearning.com/

1.17.1 Simple Linear Regression



Simple linear regression is a method used in statistics to model the relationship between two variables by fitting a linear equation to observed data. One variable is considered to be an explanatory variable (independent variable), and the other is considered to be a dependent variable. The linear regression aims to draw a straight line that best fits the data by minimizing the sum of the squares of the vertical distances of the points from the line.

Formula

$$Y \approx \beta_0 + \beta_1 X$$

The equation of a simple linear regression line is:

where:

- y is the dependent variable,
- x is the independent variable,
- β_0 is the y-intercept of the regression line,
- β_1 is the slope of the regression line, which indicates the change in y for a one-unit change in x,
- ϵ is the error term (the difference between the observed values and the values predicted by the model).

The goal is to estimate the coefficients β_0 and β_1 that minimize the sum of squared residuals (the differences between the observed values and the values predicted by the model).

For example, X may represent TV advertising and Y may represent sales. Then we can regress sales onto TV by ftting the model

$$\text{sales } \approx \beta_0 + \beta_1 \times \text{TV}$$

β_0 and β_1 are two unknown constants that represent he intercept and slope terms in the linear model. Together, β_0 and β_1 are known as the model coefficients or parameters. Once we have used our raining data to produce estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ for the model coefficients, we can predict future sales on the basis of a particular value of TV advertising sy computing

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$$

where $\hat{y}$ indicates a prediction of Y on the basis of $X = x$. Here we use a hat symbol, ' , to denote the estimated value for an unknown parameter or coefficient, or to denote the predicted value of the response.

Estimating the Coefcients

In practice, β_0 and β_1 are unknown. So before we can use to make predictions, we must use data to estimate the coefficients. Let \$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$\$

represent n observation pairs, each of which consists of a measurement of X and a measurement of Y. In the Advertising example, this data set consists of the TV advertising budget and product sales in $n = 200$ different markets. (Recall that the data are displayed in Figure 2.1.) Our goal is to obtain coefficient estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ such that the linear model (3.1) fits the available data well-that is, so that $y_i \approx \hat{\beta}_0 + \hat{\beta}_1 x_i$ for $i = 1, \ldots, n$. In other words, we want to find an intercept $\hat{\beta}_0$ and a slope $\hat{\beta}_1$ such that the resulting line is as close as possible to the $n = 200$ data points. There are a number of ways of measuring closeness. However, by far the most common approach involves minimizing the least squares criterion, and we take that approach in this chapter. Alternative approaches will be considered in Chapter 6.

Let $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$ be the prediction for Y based on the i th value of X. Then $e_i = y_i - \hat{y}_i$ represents the i th residual - this is the difference between the i th observed response value and the i th response value that is predicted by our linear model. We define the residual sum of squares (RSS) as \$$\text{RSS} = e_1^2 + e_2^2 + \cdots + e_n^2$\$

Example 1: Height and Weight

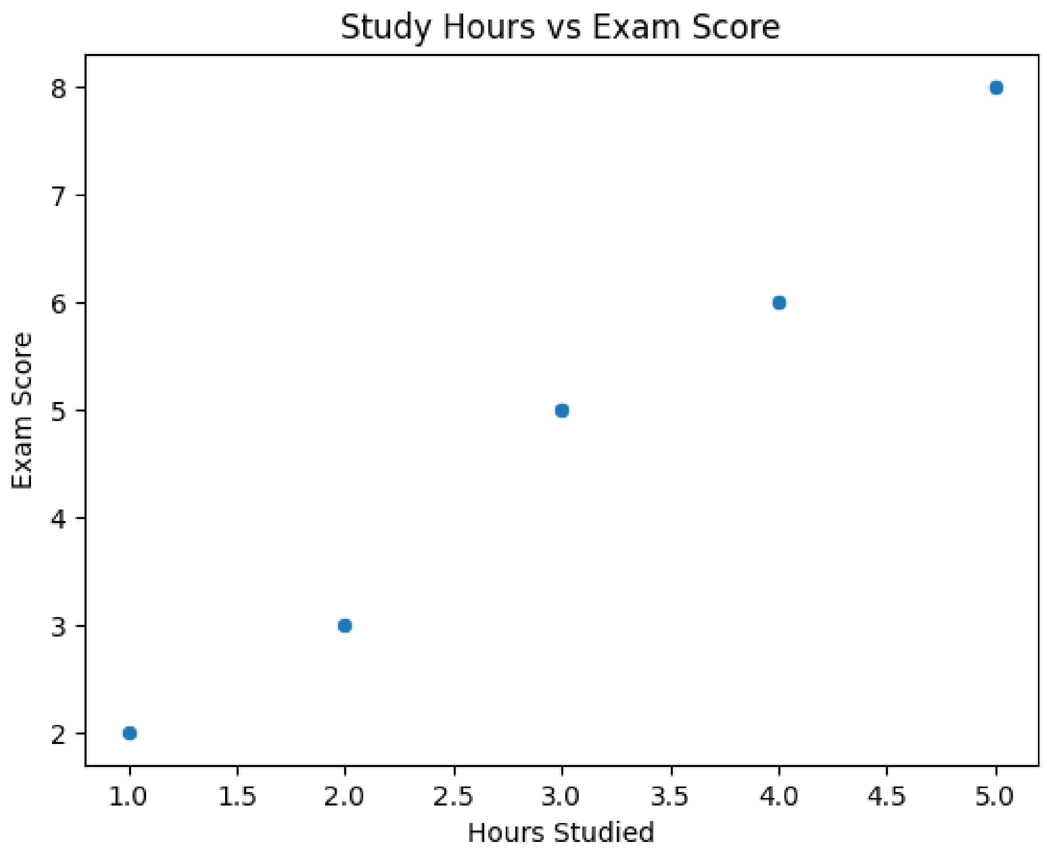
Imagine we have data on the heights and weights of a group of people. We could use simple linear regression to predict the weight of someone based on their height. In this example:

- The independent variable x would be height (e.g., in centimeters).
- The dependent variable y would be weight (e.g., in kilograms).
- By analyzing the data, we calculate the values of b_0 (the intercept) and b_1 (the slope).
- Suppose we find the regression equation to be $y = 50 + 0.5x$. This means that for every additional centimeter in height, we expect the weight to increase by 0.5 kilograms, starting from 50 kilograms.

Example 2: Hours spent studying vs. Exam score

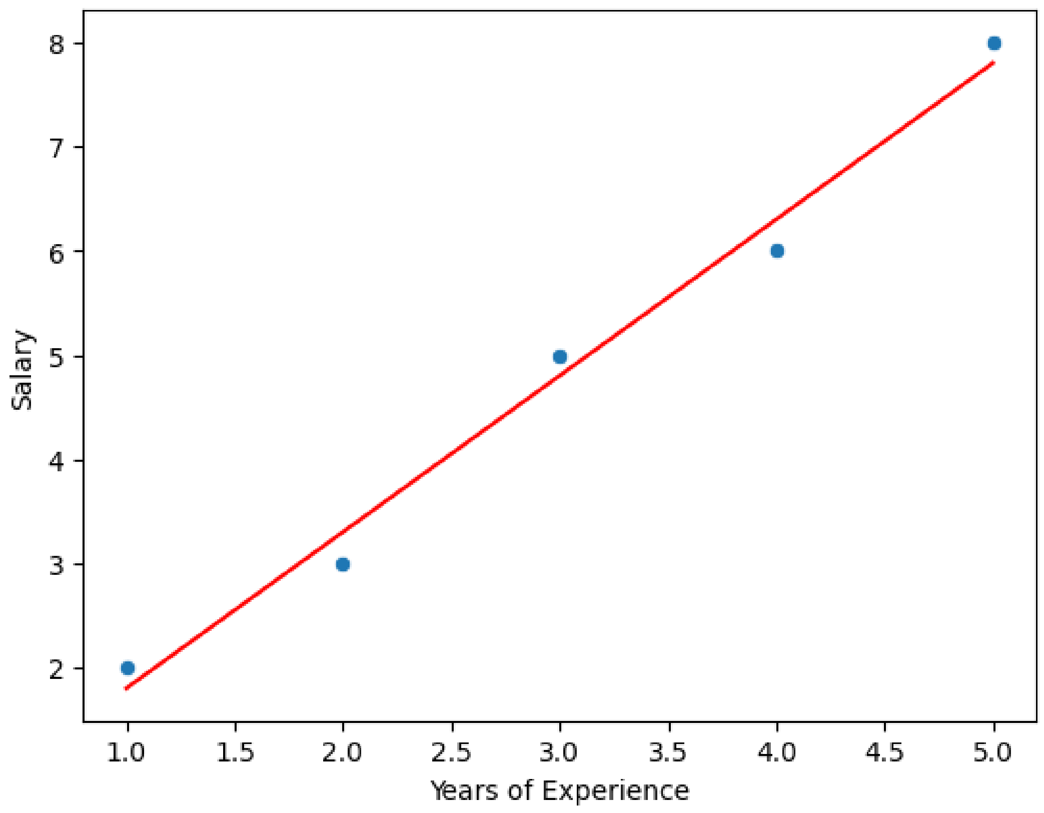
First, we can visualize the data to understand its distribution and the relationship between x and y

```python
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Sample data: Hours spent studying vs. Exam score
data = {
    'Hours_Studied': [1, 2, 3, 4, 5],
    'Exam_Score': [2, 3, 5, 6, 8]
}
df = pd.DataFrame(data)

# Plotting the data
sns.scatterplot(data=df, x='Hours_Studied', y='Exam_Score')
plt.xlabel('Hours Studied')
plt.ylabel('Exam Score')
plt.title('Study Hours vs Exam Score')
plt.show()
```



```
beta_1, beta_0 = np.polyfit(df.Hours_Studied, df.Exam_Score, 1)
# Print the computed parameters
print(f"Slope: {beta_1}, Intercept: {beta_0}")

regression_line = beta_0 + beta_1 * df.Hours_Studied

# Plot original data points
sns.scatterplot(x=df.Hours_Studied, y=df.Exam_Score)

# Plot regression line
plt.plot(df.Hours_Studied, regression_line, color='red')

plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()
```

```
Slope: 1.4999999999999998, Intercept: 0.30000000000000001
```



```
import torch
import torch.nn as nn
import torch.optim as optim

# Preparing the data
X = torch.tensor(df['Hours_Studied'].values, dtype=torch.float32).view(-1, 1)
Y = torch.tensor(df['Exam_Score'].values, dtype=torch.float32).view(-1, 1)

# Defining the linear regression model
model = nn.Linear(in_features=1, out_features=1)

# Loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training the model
epochs = 500
for epoch in range(epochs):
    # Zero the gradients
    optimizer.zero_grad()

    # Forward pass
    outputs = model(X)
    loss = criterion(outputs, Y)

    # Backward and optimize
```

(continues on next page)

(continued from previous page)

```
    loss.backward()
    optimizer.step()

    if (epoch+1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item()}')

# Coefficients
print('Estimated coefficients:')
print('Weight:', model.weight.item())
print('Bias:', model.bias.item())
```

```
Epoch [100/500], Loss: 0.06318604946136475
Epoch [200/500], Loss: 0.06161843612790108
Epoch [300/500], Loss: 0.06082212179899216
Epoch [400/500], Loss: 0.06041758134961128
Epoch [500/500], Loss: 0.06021212413907051
Estimated coefficients:
Weight: 1.4905762672424316
Bias: 0.33402296900749207
```

Optmization

In simple linear regression, the Least Squares Method is used to find the values of the coefficients β_0 and β_1 that minimize the sum of the squared differences between the observed values and the values predicted by the linear model. The sum of squared differences, also known as the sum of squared residuals (SSR), is given by:

$$SSR = \sum_{i=1}^{n}(y_i - \hat{y_i})^2$$

where y_i are the observed values, $\hat{y_i}$ are the predicted values, and n is the number of observations. The predicted values are calculated using the linear model equation:

$$\hat{y_i} = \beta_0 + \beta_1 x_i$$

To find the values of β_0 and β_1 that minimize the SSR, we take the partial derivatives of the SSR with respect to β_0 and β_1, set them equal to zero, and solve the resulting system of equations. This process is called the method of least squares.

Derivatives Calculation

1. **Partial Derivative with Respect to** β_0:

$$\frac{\partial SSR}{\partial \beta_0} = \frac{\partial}{\partial \beta_0}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)^2$$

Using the chain rule, this derivative simplifies to:

$$-2\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)$$

2. **Partial Derivative with Respect to** β_1:

$$\frac{\partial SSR}{\partial \beta_1} = \frac{\partial}{\partial \beta_1} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2$$

This derivative simplifies to:

$$-2 \sum_{i=1}^{n} x_i (y_i - \beta_0 - \beta_1 x_i)$$

Solving for β_0 and β_1

To find the values of β_0 and β_1 that minimize the SSR, we set each derivative equal to zero:

$-2 \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i) = 0$

$-2 \sum_{i=1}^{n} x_i (y_i - \beta_0 - \beta_1 x_i) = 0$

These can be simplified and rearranged to form a system of linear equations:

$\sum_{i=1}^{n} y_i = n\beta_0 + \beta_1 \sum_{i=1}^{n} x_i$

$\sum_{i=1}^{n} x_i y_i = \beta_0 \sum_{i=1}^{n} x_i + \beta_1 \sum_{i=1}^{n} x_i^2$

Solving this system of equations gives the values of β_0 and β_1:

$$\beta_1 = \frac{n \sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n \sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}$$

$$\beta_0 = \frac{\sum_{i=1}^{n} y_i - \beta_1 \sum_{i=1}^{n} x_i}{n}$$

Interpretation

- β_1 (the slope) represents the estimated change in the dependent variable (y) for a one-unit change in the independent variable (x).
- β_0 (the intercept) represents the estimated value of y when $x = 0$.

This method ensures that the line of best fit is determined by minimizing the difference between the observed values and the values predicted by the linear model, specifically by minimizing the sum of the squares of these differences.

Evaluating the Model

In linear regression, the evaluation of the model's performance often involves determining how well the model fits the observed data. Two key metrics used in this context are the Residual Sum of Squares (RSS) and the Total Sum of Squares (TSS). Understanding these metrics allows us to assess the proportion of the variance in the dependent variable that is predictable from the independent variable.

Residual Sum of Squares (RSS)

The Residual Sum of Squares (RSS), also known as the Sum of Squared Residuals (SSR), measures the amount of variance in the dependent variable that is not explained by the regression model. In simpler terms, it quantifies how much the data points deviate from the regression line. Mathematically, RSS is defined as:

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

where:

- y_i is the actual value of the dependent variable for the ith observation,
- $\hat{y}_i$ is the predicted value of the dependent variable for the ith observation based on the regression line,
- n is the total number of observations.

A smaller RSS indicates a model that closely fits the data.

Total Sum of Squares (TSS)

The Total Sum of Squares (TSS) measures the total variance in the dependent variable. It represents how much the data points deviate from the mean of the dependent variable. TSS is an indicator of the total variability within the data set. It is defined as:

$$TSS = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

where:

- y_i is the actual value of the dependent variable for the ith observation,
- $\bar{y}$ is the mean value of the dependent variable across all observations.

TSS is used as a baseline to compare the performance of the regression model.

The relationship between RSS and TSS is crucial for understanding how well the linear regression model fits the data. To quantify this, one common metric is the R^2 statistic, also known as the coefficient of determination. R^2 measures the proportion of the variance in the dependent variable that is predictable from the independent variable(s). It is calculated as:

$$R^2 = 1 - \frac{RSS}{TSS}$$

- An R^2 value of 1 indicates that the regression model perfectly fits the data (with RSS = 0).
- An R^2 value of 0 suggests that the model does not explain any of the variability in the dependent variable around its mean (with RSS = TSS).

While R^2 is a useful indicator of model fit, it's important to consider other metrics and tests as well, especially when dealing with multiple regression models or models that might be overfitting the data.

Coefficient Significance

The significance of a coefficient in a linear regression model indicates whether a variable has a statistically significant relationship with the dependent variable. It's assessed using the t-test, which evaluates the null hypothesis that the coefficient is equal to zero (no effect) against the alternative hypothesis that the coefficient is not equal to zero (a significant effect).

T-statistic is calculated as:

$$t = \frac{\hat{\beta}_j}{SE(\hat{\beta}_j)}$$

where $\hat{\beta}_j$ is the estimated coefficient for the predictor variable j, and $SE(\hat{\beta}_j)$ is the standard error of the estimated coefficient.

The t-statistic is compared against a critical value from the t-distribution with $n - p - 1$ degrees of freedom, where n is the number of observations and p is the number of predictors. If the absolute value of the t-statistic is greater than the critical value, the null hypothesis is rejected, indicating that the coefficient is statistically significant.

Example: Imagine a dataset where we're predicting house prices based on various features. One of the features is the number of bedrooms. Suppose the estimated coefficient for the number of bedrooms is 20,000 (indicating that each additional bedroom is associated with an average increase of $20,000 in house price), with a standard error of 5,000.

$$t = \frac{20,000}{5,000} = 4$$

Assuming a critical value of 2.576 for a 99% confidence level, the t-statistic is greater than the critical value, suggesting the number of bedrooms is a significant predictor of house price.

Test Error

Test error, also known as the mean squared error (MSE) on the test set, measures the model's performance on unseen data. It's calculated by comparing the model's predictions on the test set with the actual values.

$$MSE = \frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} (y_i - \hat{y}_i)^2$$

where n_{test} is the number of observations in the test set, y_i are the actual values, and $\hat{y}_i$ are the predicted values by the model.

The MSE provides a numeric measure of the average squared deviation between the observed actual outcomes and the outcomes predicted by the model. A lower MSE indicates a better fit of the model to the data.

Example: Continuing with the house price prediction model, suppose we have a test set of 100 houses. After predicting the prices with our model, we calculate the differences between the actual prices and the predicted prices, square those differences, and average them. If the resulting MSE is $10,000,000, this means, on average, the model's predictions deviate from the actual prices by about 3,162 (the square root of MSE) per house.

Both the significance of coefficients and test error provide valuable insights into a linear regression model's effectiveness and reliability. Coefficient significance helps identify which variables have a meaningful impact on the dependent variable, while test error quantifies the model's predictive accuracy on new, unseen data.

1.17.2 Multi-Linear Regression

Multi-linear regression is an extension of simple linear regression to predict an outcome based on multiple predictors or independent variables. The goal of multi-linear regression is to model the relationship between two or more predictors and a response variable by fitting a linear equation to observed data. The formula for a multi-linear regression model can be expressed as:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n + \epsilon$$

Where:

- Y is the dependent variable (the variable being predicted),
- $X_1, X_2, ..., X_n$ are the independent variables (predictors),
- β_0 is the intercept,
- $\beta_1, \beta_2, ..., \beta_n$ are the coefficients of the predictors which represent the weight of each predictor in the equation,
- ϵ is the error term, the part of Y the regression model is unable to explain.

Multi-Linear Regression in PyTorch

PyTorch is a popular open-source machine learning library for Python, primarily used for applications such as computer vision and natural language processing. It's also extensively used for deep learning applications, but you can certainly use it for simple multi-linear regression.

Here's a basic example of how to implement multi-linear regression in PyTorch:

1. **Data Preparation**: First, we need to prepare our dataset. This involves splitting our data into predictors (X) and a target variable (Y).
2. **Model Creation**: We'll create a linear model using PyTorch's `nn.Linear` module to represent our multi-linear regression model.
3. **Loss Function**: The loss function will measure how well the model predicts the target variable. For regression problems, Mean Squared Error (MSE) is commonly used.
4. **Optimizer**: This is what we'll use to update the model parameters (β values) to minimize the loss function. A common optimizer is Stochastic Gradient Descent (SGD).

Example Code

This code snippet demonstrates a simple example of performing multi-linear regression using PyTorch. Here, `X` is our matrix of predictors, and `Y` is the target variable. We define a `model` with `nn.Linear` specifying 3 input features and 1 output feature. The training loop updates our model's weights using the specified optimizer and loss function. After training, we print the loss at every 100 epochs to observe how it decreases, indicating that our model is learning. Finally, we print out the learned parameters, which are the weights and bias from our model.

Remember, this is a very basic example. In practice, you'd have a much larger dataset, you would divide your data into training and test datasets, and you might need to normalize your data for better performance.

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Example dataset with 3 predictors
X = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]], dtype=torch.float32)
Y = torch.tensor([[14], [32], [50], [68]], dtype=torch.float32)  # target variable

# Define the model
model = nn.Linear(in_features=3, out_features=1)

# Loss function
criterion = nn.MSELoss()

# Optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training loop
epochs = 100
for epoch in range(epochs):
    # Forward pass: Compute predicted y by passing x to the model
    Y_pred = model(X)

    # Compute loss
    loss = criterion(Y_pred, Y)
```

(continues on next page)

(continued from previous page)

```
    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 50 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}')

# Display learned parameters
for name, param in model.named_parameters():
    if param.requires_grad:
        print(name, param.data)
```

```
Epoch 50, Loss: inf
Epoch 100, Loss: inf
weight tensor([[-4.2858e+35, -4.8812e+35, -5.4765e+35]])
bias tensor([-5.9534e+34])
```

Bias-Variance Trade-Off

The Bias-Variance Trade-Off is a fundamental concept in supervised machine learning that describes the trade-off between the ability of a model to approximate the true underlying function (bias) and its sensitivity to fluctuations in the training data (variance). Understanding this trade-off is crucial for building models that generalize well from training data to unseen data.

Bias

Bias refers to the error due to overly simplistic assumptions in the learning algorithm. High bias can cause the model to miss relevant relations between features and target outputs (underfitting), meaning the model is not complex enough to capture the underlying patterns in the data.

Variance

Variance refers to the error due to too much complexity in the learning algorithm. High variance can cause the model to model the random noise in the training data (overfitting), meaning the model is too complex and captures noise as if it were a pattern in the data.

Trade-Off

The trade-off is that algorithms with high bias typically have lower variance, and vice versa. A model with high bias might overly simplify the model, making it perform poorly on both training and unseen data. A model with high variance might perform exceptionally well on training data but poorly on unseen data because it's too tailored to the training data, including its noise.

Ideally, you want to find a sweet spot that minimizes both bias and variance, providing good generalization to new data.

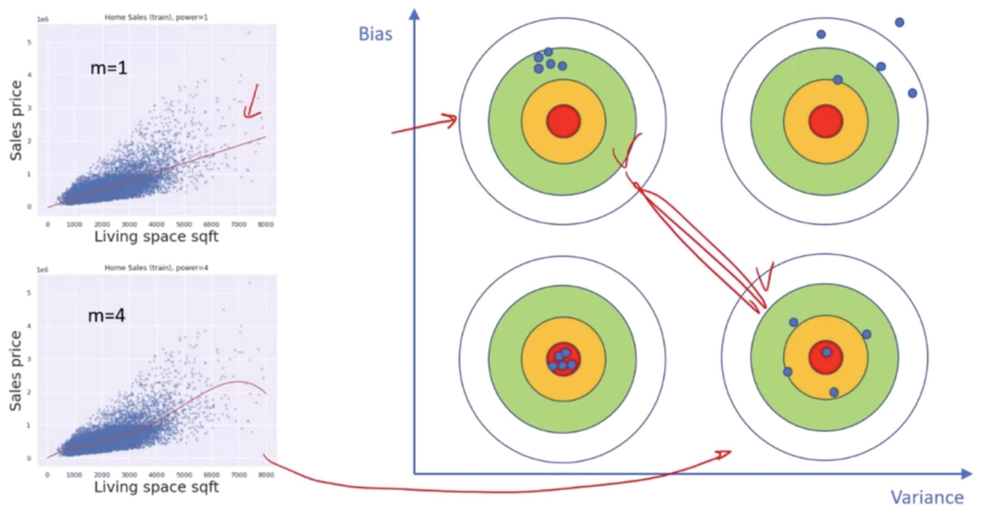


Image Examples

Let's illustrate this with some image examples.

1. **High Bias (Underfitting)**: Imagine a scenario where you're trying to fit a line (linear regression) through data points that clearly form a curved pattern (quadratic). The linear model is too simple to capture the curve, resulting in high bias.
2. **High Variance (Overfitting)**: Now, imagine fitting a high-degree polynomial through those same points. The model fits the training data points perfectly, including the noise, resulting in a wavy line that doesn't capture the true underlying pattern.
3. **Bias-Variance Tradeoff**: The optimal model would be one that correctly assumes a quadratic relationship, fitting a curve that captures the underlying pattern without being affected by the noise.

Let's generate these examples visually.

1. **High Bias Example**: A straight line attempting to fit through quadratic data points.
2. **High Variance Example**: A high-degree polynomial line that passes through every point, including noise.
3. **Ideal Model**: A quadratic curve that fits well with the underlying pattern of the data, avoiding overfitting and underfitting.

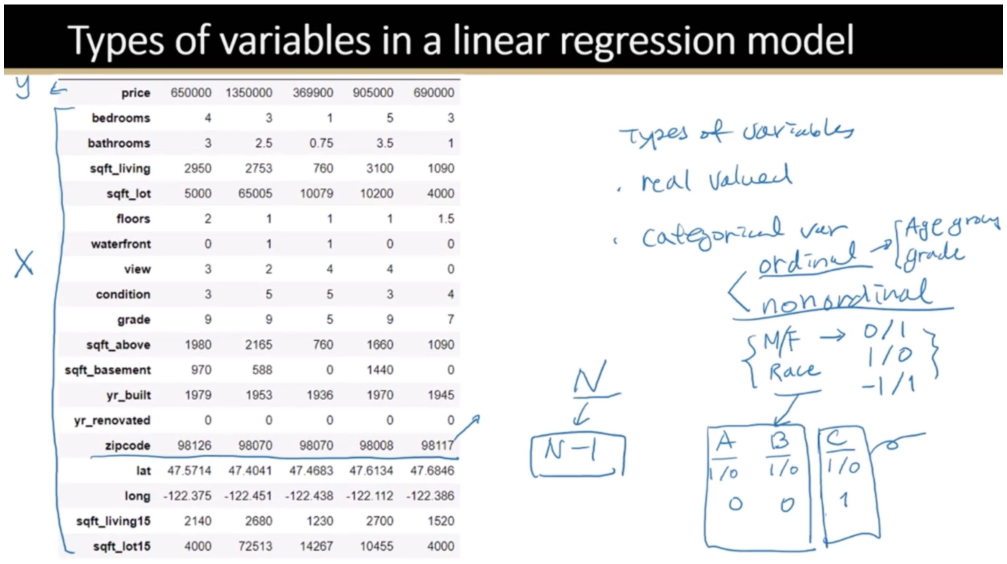
I'll create an image to represent the concept of the Bias-Variance Tradeoff visually.

The image above visually represents the Bias-Variance Tradeoff in machine learning. It consists of three panels, each illustrating a key concept:

1. **High Bias**: The first panel shows a simple straight line that does not capture the essence of the underlying curve formed by the data points. This scenario is typical of models with high bias, where the simplicity of the model prevents it from capturing the true relationship between variables, leading to underfitting.
2. **High Variance**: The second panel depicts a highly complex, wavy line that passes through every single data point, including noise. This is characteristic of models with high variance, where the complexity of the model causes it to capture noise as if it were a significant pattern, leading to overfitting.
3. **Ideal Model**: The third panel illustrates an optimal curve that smoothly captures the general pattern of the data points without fitting to the noise. This represents the desirable balance in the Bias-Variance Tradeoff, where the model is complex enough to capture the underlying pattern but not so complex that it fits the noise in the data.

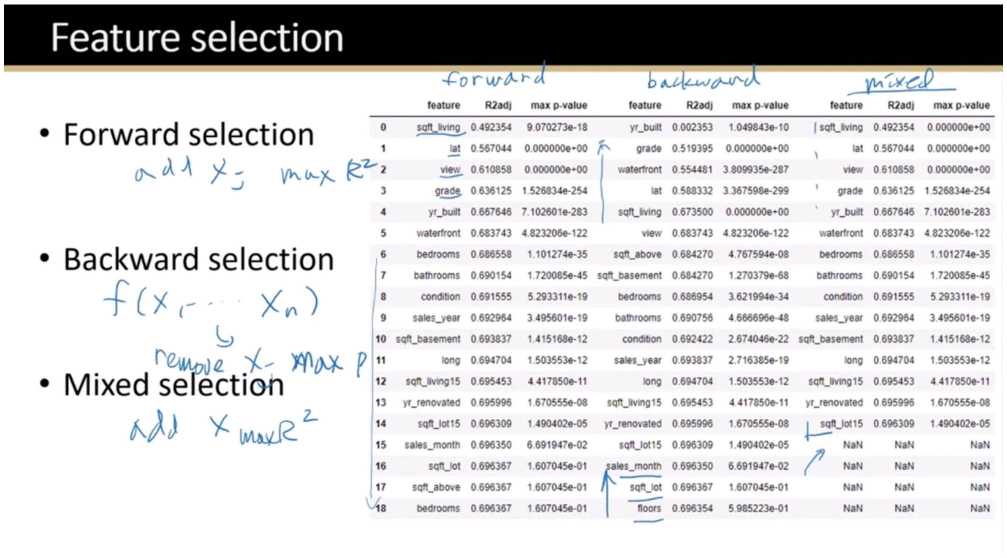
This visualization aids in understanding the trade-off between bias and variance, highlighting the importance of finding a model that achieves a good balance for optimal performance on unseen data.

Types of variables in multi linear regression



Types of variables in a linear regression model

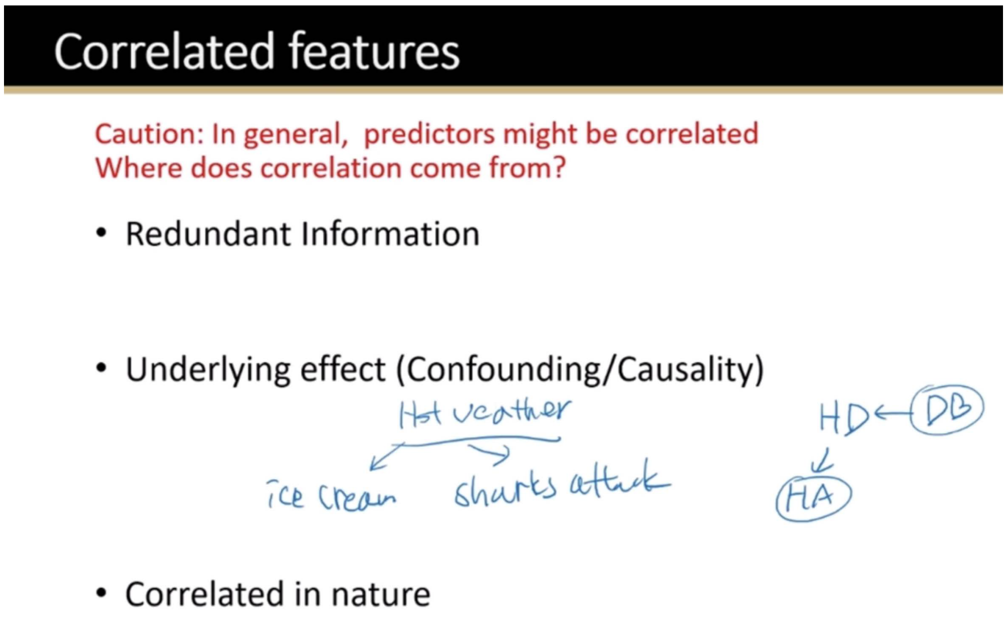
price	650000	1350000	369900	905000	690000
bedrooms	4	3	1	5	3
bathrooms	3	2.5	0.75	3.5	1
sqft_living	2950	2753	760	3100	1090
sqft_lot	5000	65005	10079	10200	4000
floors	2	1	1	1	1.5
waterfront	0	1	1	0	0
view	3	2	4	4	0
condition	3	5	5	3	4
grade	9	9	5	9	7
sqft_above	1980	2165	760	1660	1090
sqft_basement	970	588	0	1440	0
yr_built	1979	1953	1936	1970	1945
yr_renovated	0	0	0	0	0
zipcode	98126	98070	98070	98008	98117
lat	47.5714	47.4041	47.4683	47.6134	47.6846
long	-122.375	-122.451	-122.438	-122.112	-122.386
sqft_living15	2140	2680	1230	2700	1520
sqft_lot15	4000	72513	14267	10455	4000



Feature selection

- Forward selection
- Backward selection
- Mixed selection

	feature	R2adj	max p-value	feature	R2adj	max p-value	feature	R2adj	max p-value
0	sqft_living	0.492354	9.070273e-18	yr_built	0.002353	1.049843e-10	sqft_living	0.492354	0.000000e+00
1	lat	0.567044	0.000000e+00	grade	0.519395	0.000000e+00	lat	0.567044	0.000000e+00
2	view	0.610858	0.000000e+00	waterfront	0.554481	3.809935e-287	view	0.610858	0.000000e+00
3	grade	0.636125	1.526834e-254	lat	0.588332	3.367598e-299	grade	0.636125	1.526834e-254
4	yr_built	0.667646	7.102601e-283	sqft_living	0.673500	0.000000e+00	yr_built	0.667646	7.102601e-283
5	waterfront	0.683743	4.823206e-122	view	0.683743	4.823206e-122	waterfront	0.683743	4.823206e-122
6	bedrooms	0.686558	1.101274e-35	sqft_above	0.684270	4.767594e-08	bedrooms	0.686558	1.101274e-35
7	bathrooms	0.690154	1.720085e-45	sqft_basement	0.684270	1.270379e-68	bathrooms	0.690154	1.720085e-45
8	condition	0.691555	5.293311e-19	bedrooms	0.686954	3.621994e-34	condition	0.691555	5.293311e-19
9	sales_year	0.692964	3.495601e-19	bathrooms	0.690756	4.666696e-48	sales_year	0.692964	3.495601e-19
10	sqft_basement	0.693837	1.415168e-12	condition	0.692422	2.674046e-22	sqft_basement	0.693837	1.415168e-12
11	long	0.694704	1.503553e-12	sales_year	0.693837	2.716385e-19	long	0.694704	1.503553e-12
12	sqft_living15	0.695453	4.417850e-11	long	0.694704	1.503553e-12	sqft_living15	0.695453	4.417850e-11
13	yr_renovated	0.695996	1.670555e-08	sqft_living15	0.695453	4.417850e-11	yr_renovated	0.695996	1.670555e-08
14	sqft_lot15	0.696309	1.490402e-05	yr_renovated	0.695996	1.670555e-08	sqft_lot15	0.696309	1.490402e-05
15	sales_month	0.696350	6.691947e-02	sqft_lot15	0.696309	1.490402e-05	NaN	NaN	NaN
16	sqft_lot	0.696367	1.607045e-01	sales_month	0.696350	6.691947e-02	NaN	NaN	NaN
17	sqft_above	0.696367	1.607045e-01	sqft_lot	0.696367	1.607045e-01	NaN	NaN	NaN
18	bedrooms	0.696367	1.607045e-01	floors	0.696354	5.985223e-01	NaN	NaN	NaN



Correlated features

Caution: In general, predictors might be correlated
Where does correlation come from?

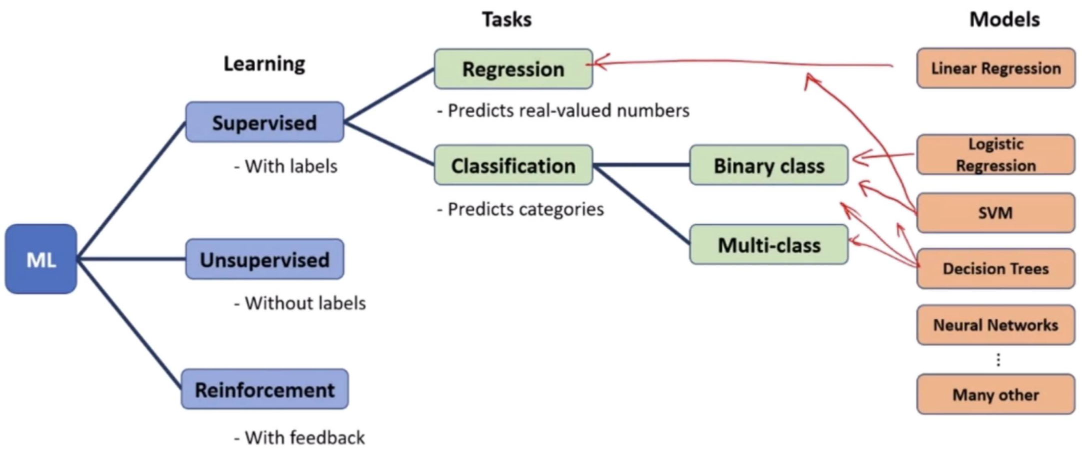
- Redundant Information
- Underlying effect (Confounding/Causality)

Hot weather
ice cream
sharks attack

HD
DB
HA

- Correlated in nature

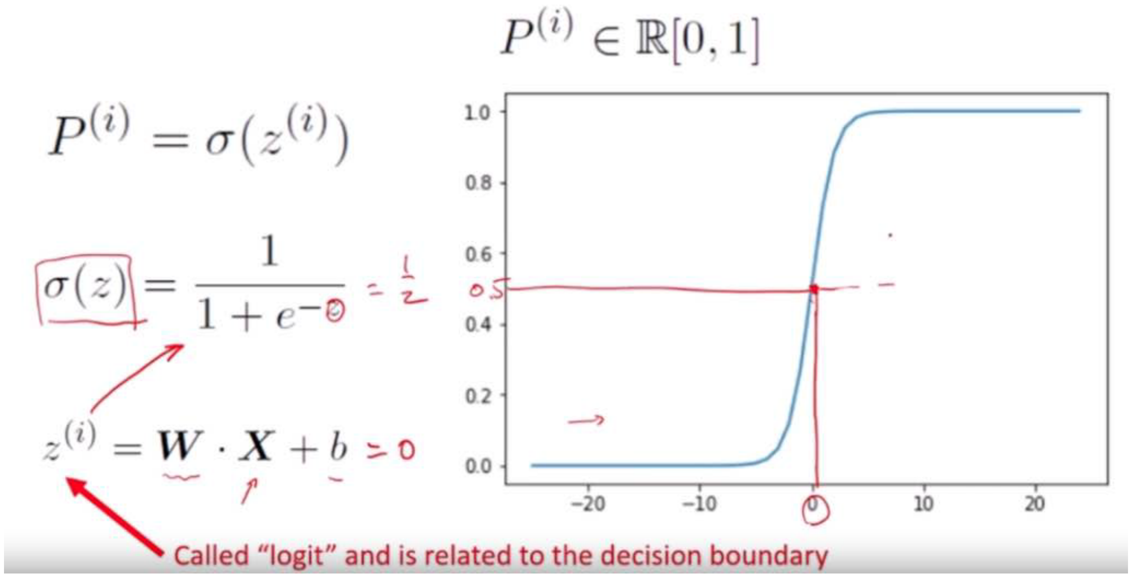
1.18 Logistic Regression

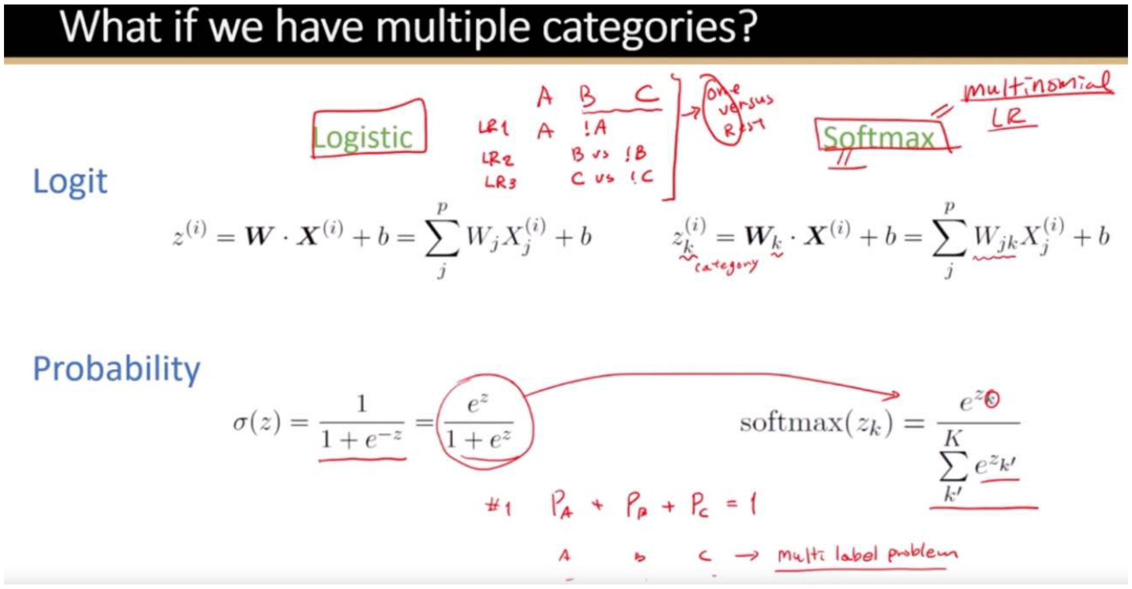


Logistic regression is a statistical method for analyzing a dataset in which there are one or more independent variables that determine an outcome. The outcome is measured with a dichotomous variable (in which there are only two possible outcomes). It's used extensively for binary classification problems, such as spam detection (spam or not spam), loan default (default or not), disease diagnosis (positive or negative), etc. Logistic regression predicts the probability that a given input belongs to a certain category.

1.18.1 Sigmoid / Logistic Function :

The core of logistic regression is the sigmoid function, which maps any real-valued number into a value between 0 and 1, making it suitable for probability estimation. The sigmoid function is defined as $\sigma(z) = \frac{1}{1+e^{-z}}$, where z is the input to the function, often $z = w^T x + b$, with w being the weights, x the input features, and b the bias.





What if we have multiple categories?

	A	B	C
LR1	A	!A	
LR2		B vs !B	
LR3		C vs !C	

One versus Rest

Logistic

Softmax

multinomial LR

Logit

$$z^{(i)} = \boldsymbol{W} \cdot \boldsymbol{X}^{(i)} + b = \sum_{j}^{p} W_j X_j^{(i)} + b \qquad \underset{\text{category}}{z_k^{(i)}} = \boldsymbol{W}_k \cdot \boldsymbol{X}^{(i)} + b = \sum_{j}^{p} W_{jk} X_j^{(i)} + b$$

Probability

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z} \qquad \text{softmax}(z_k) = \frac{e^{z_k}}{\sum_{k'}^{K} e^{z_{k'}}}$$

#1 $P_A + P_B + P_C = 1$

A b c → multi label problem

1.18.2 Cost / loss Function:

MLE in Binary Classification

Maximum Likelihood Estimation (MLE) is a central concept in statistical modeling, including binary classification tasks. Binary classification involves predicting whether an instance belongs to one of two classes (e.g., spam or not spam, diseased or healthy) based on certain input features.

In binary classification, you often model the probability of the positive class ($y = 1$) as a function of input features (X) using a logistic function, leading to logistic regression. The probability that a given instance belongs to the positive class can be expressed as:

$$P(Y = 1|X; \theta) = \frac{1}{1 + e^{-(\beta_0+\beta_1 X_1+\ldots+\beta_n X_n)}}$$

Here, θ represents the model parameters ($\beta_0, \beta_1, ..., \beta_n$), and $X_1, ..., X_n$ are the input features.

The likelihood function $L(\theta)$ in the context of binary classification is the product of the probabilities of each observed label, given the input features and the model parameters. For a dataset with m instances, where y_i is the label of the i-th instance, and p_i is the predicted probability of the i-th instance being in the positive class, the likelihood is:

$$L(\theta) = \prod_{i=1}^{m} p_i^{y_i} (1 - p_i)^{1-y_i}$$

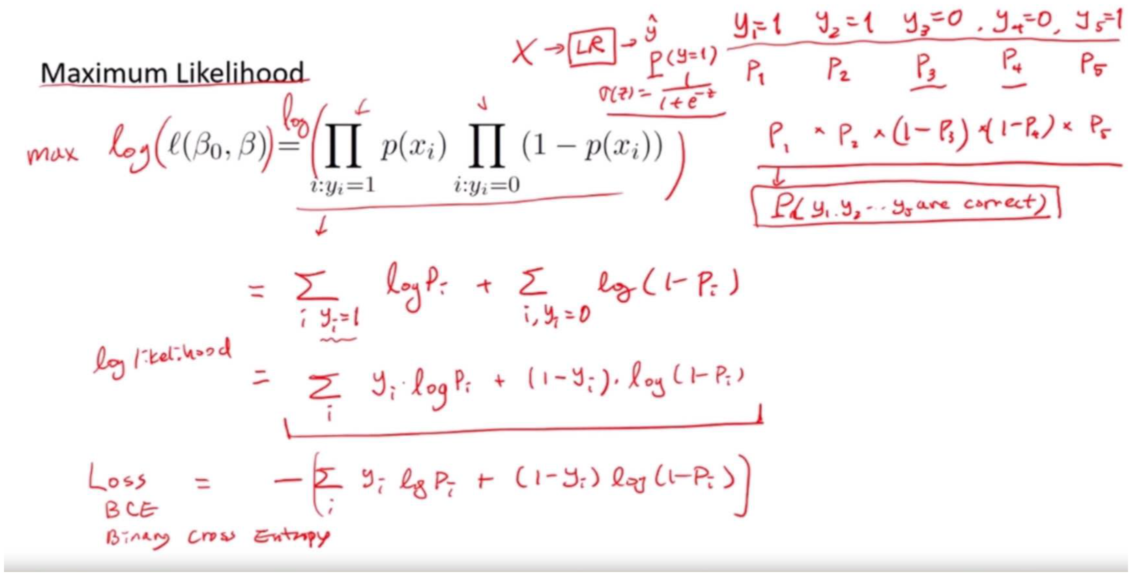
This product is maximized when the model parameters (θ) are such that the predicted probabilities (p_i) are close to 1 for actual positive instances and close to 0 for actual negative instances.

Log-Likelihood:

To simplify calculations and handle numerical stability, we use the log-likelihood, which converts the product into a sum:

$$\ell(\theta) = \sum_{i=1}^{m} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

The goal is to find the parameters (θ) that maximize this log-likelihood.



Maximum Likelihood

$$X \rightarrow \boxed{LR} \rightarrow \overset{\hat{y}}{P(y=1)} \qquad \sigma(z) = \frac{1}{1+e^{-z}}$$

$$\begin{array}{ccccc} y_1=1 & y_2=1 & y_3=0, & y_4=0, & y_5=1 \\ \hline P_1 & P_2 & \underline{P_3} & \underline{P_4} & P_5 \end{array}$$

$$P_1 \times P_2 \times (1-P_3) \times (1-P_4) \times P_5$$

$$\boxed{P(y_1, y_2 \ldots y_5 \text{ are correct})}$$

$$\max \log\left(\ell(\beta_0, \beta)\right) = \log\left(\prod_{i:y_i=1} p(x_i) \prod_{i:y_i=0} (1 - p(x_i))\right)$$

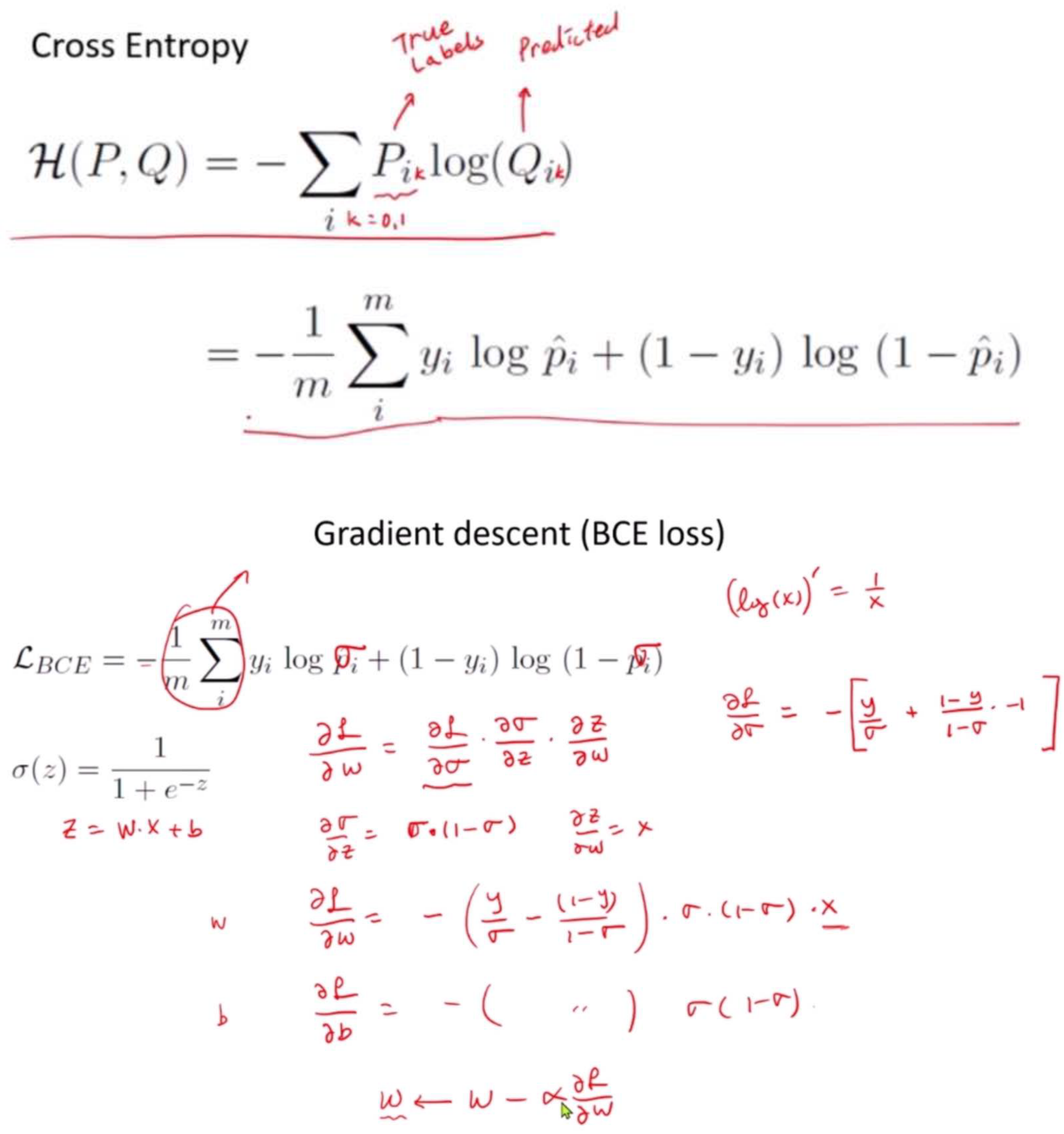
$$= \sum_{i,\, y_i=1} \log P_i + \sum_{i,\, y_i=0} \log(1-P_i)$$

log likelihood

$$= \underbrace{\sum_i y_i \cdot \log P_i + (1-y_i) \cdot \log(1-P_i)}$$

$$\text{Loss}_{\text{BCE}} = -\left[\sum_i y_i \log P_i + (1-y_i) \log(1-P_i)\right]$$

Binary Cross Entropy



Cross Entropy

$$\mathcal{H}(P,Q) = -\sum_{i\ k=0,1} \overset{\text{True Labels}}{P_{ik}} \log(\overset{\text{Predicted}}{Q_{ik}})$$

$$= -\frac{1}{m}\sum_{i}^{m} y_i \log \hat{p}_i + (1-y_i)\log(1-\hat{p}_i)$$

Gradient descent (BCE loss)

$$\mathcal{L}_{BCE} = -\frac{1}{m}\sum_{i}^{m} y_i \log \sigma_i + (1-y_i)\log(1-\sigma_i)$$

$$(\log(x))' = \frac{1}{x}$$

$$\frac{\partial L}{\partial \sigma} = -\left[\frac{y}{\sigma} + \frac{1-y}{1-\sigma}\cdot -1\right]$$

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

$$z = W\cdot X + b$$

$$\frac{\partial L}{\partial w} = \underline{\frac{\partial L}{\partial \sigma}}\cdot\frac{\partial \sigma}{\partial z}\cdot\frac{\partial z}{\partial w}$$

$$\frac{\partial \sigma}{\partial z} = \sigma\cdot(1-\sigma) \qquad \frac{\partial z}{\partial w} = x$$

$$w \qquad \frac{\partial L}{\partial w} = -\left(\frac{y}{\sigma} - \frac{(1-y)}{1-\sigma}\right)\cdot\sigma\cdot(1-\sigma)\cdot\underline{x}$$

$$b \qquad \frac{\partial L}{\partial b} = -\left(\quad '' \quad\right)\ \sigma(1-\sigma)$$

$$\underset{\sim}{w} \leftarrow w - \alpha\frac{\partial L}{\partial w}$$

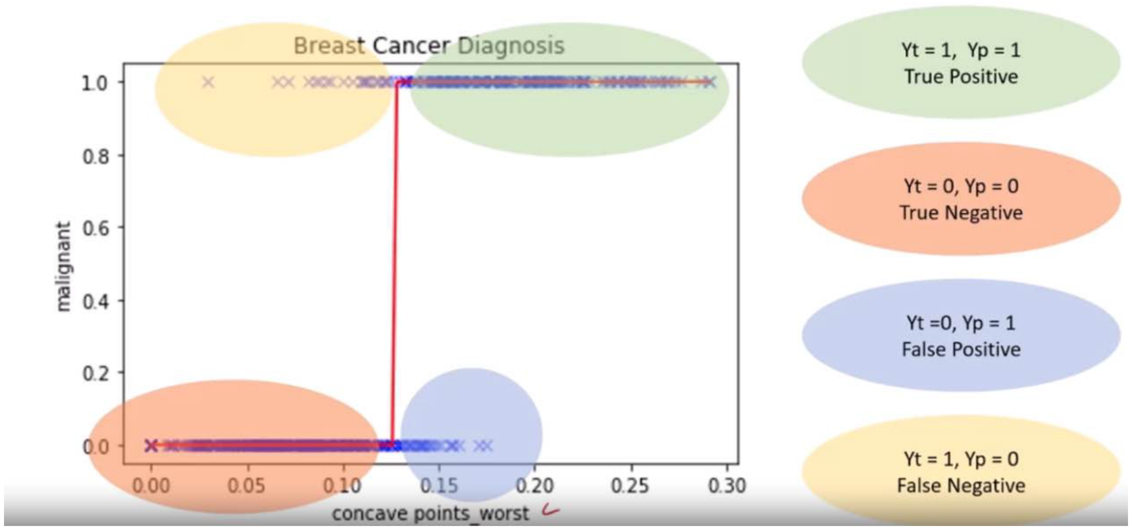
Threshold Decision:

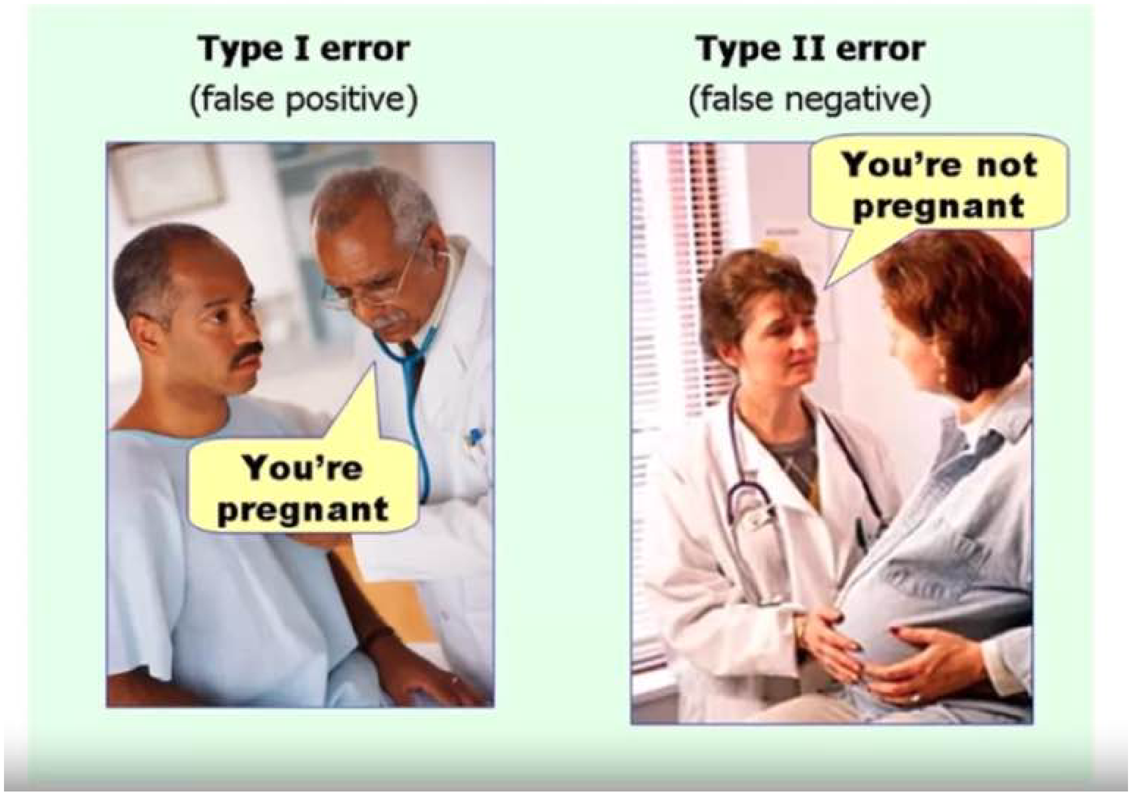
The probability outcome from the sigmoid function is converted into a binary outcome via a threshold decision rule, usually 0.5 (if the sigmoid output is greater than or equal to 0.5, the outcome is classified as 1, otherwise as 0).

Performance Metrics:

Here are some performance metrics that can be used to evaluate the performance of a binary classifier:

- Accuracy
- Precision
- Recall
- F1 score
- ROC curve
- Confusion matrix
- AUC (Area Under the Curve)





Logistic Regression in PyTorch:

Here's a simple example of how to implement logistic regression in PyTorch. PyTorch is a deep learning framework that provides a lot of flexibility and capabilities, including automatic differentiation which is handy for logistic regression.

Step 1: Import Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
```

Step 2: Create Dataset

For simplicity, let's assume a binary classification task with some synthetic data.

```
# Features [sample size, number of features]
X = torch.tensor([[1, 2], [4, 5], [7, 8], [9, 10]], dtype=torch.float32)
# Labels [sample size, 1]
y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)
```

Step 3: Define the Model

```python
class LogisticRegressionModel(nn.Module):
    def __init__(self, input_size, num_classes):
        super(LogisticRegressionModel, self).__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        out = torch.sigmoid(self.linear(x))
        return out
```

Step 4: Instantiate Model, Loss, and Optimizer

```python
input_size = 2
num_classes = 1
model = LogisticRegressionModel(input_size, num_classes)

criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

Step 5: Train the Model

```python
num_epochs = 100
for epoch in range(num_epochs):
    # Forward pass
    outputs = model(X)
    loss = criterion(outputs, y)

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

This code snippet demonstrates the essential parts of implementing logistic regression in PyTorch, including model definition, data preparation, loss computation, and the training loop. After training, the model's weights are adjusted to minimize the loss, making the model capable of predicting the probability that a new, unseen input belongs to a certain category.

1.19 Non Parametric Models

1.19.1 KNN (K-Nearest Neighbors)

The k-Nearest Neighbors (k-NN) algorithm is a type of supervised machine learning algorithm used for both classification and regression tasks. However, it's more commonly used for classification purposes. The "k" in k-NN represents a number specified by the user, and it refers to the number of nearest neighbors in the data that the algorithm will consider to make a prediction for a new data point.

How It Works

1. **Choose the number of k and a distance metric**: First, you decide on the number of neighbors, "k", and the method for measuring distance between data points (common metrics include Euclidean, Manhattan, and Hamming distance).
2. **Find the k-nearest neighbors**: For a given data point that you want to classify or predict its value, the algorithm identifies the k nearest data points in the training dataset based on the distance metric.
3. **Make predictions**:
   - **For classification**, the algorithm assigns the class to the new data point based on the majority vote of its k nearest neighbors.
   - **For regression**, it predicts the value for the new point based on the average (or another aggregate measure) of the values of its k nearest neighbors.

Example

Imagine you have a dataset of fruits, where each fruit is described by two features: weight and color (let's simplify color to a numerical value for the purpose of this example, where 1 = green, 2 = yellow, 3 = red), and you're trying to classify them as either "Apple" or "Banana".

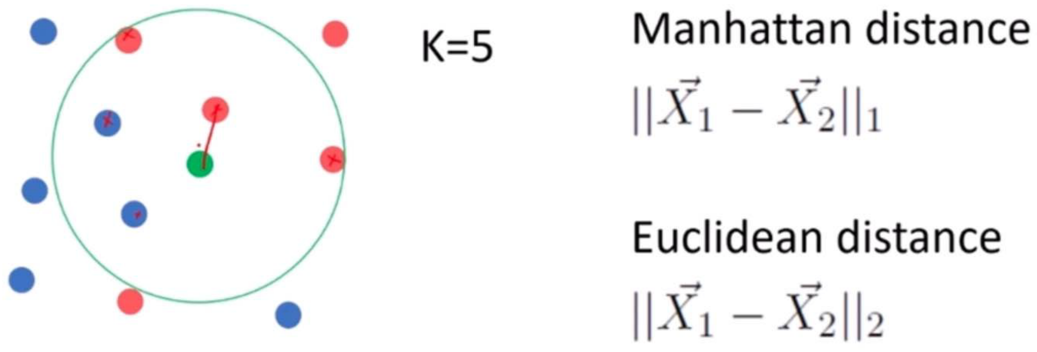
Now, you have a new fruit that you want to classify, and this fruit weighs 150 grams and has a color value of 1 (green).

If you choose k=3 (looking at the three nearest neighbors), and the three closest fruits in your dataset to this new fruit are:

- Fruit 1: 145 grams, color 1, labeled "Apple"
- Fruit 2: 160 grams, color 2, labeled "Apple"
- Fruit 3: 155 grams, color 1, labeled "Banana"

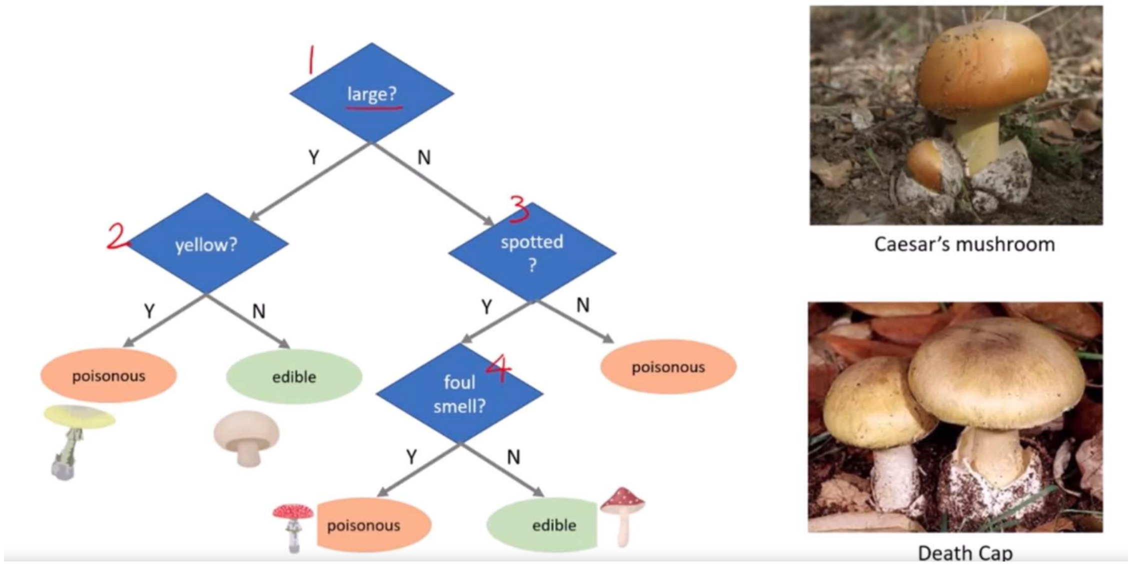
Based on the majority vote among the nearest neighbors, the algorithm would classify the new fruit as an "Apple" because two out of three nearest neighbors are labeled as "Apple".

This example simplifies the concept to make it easier to understand. In practice, the k-NN algorithm can handle datasets with many more features and more complex decision boundaries. The key takeaway is that the k-NN algorithm relies on the similarity of the nearest observations in the feature space to make predictions.



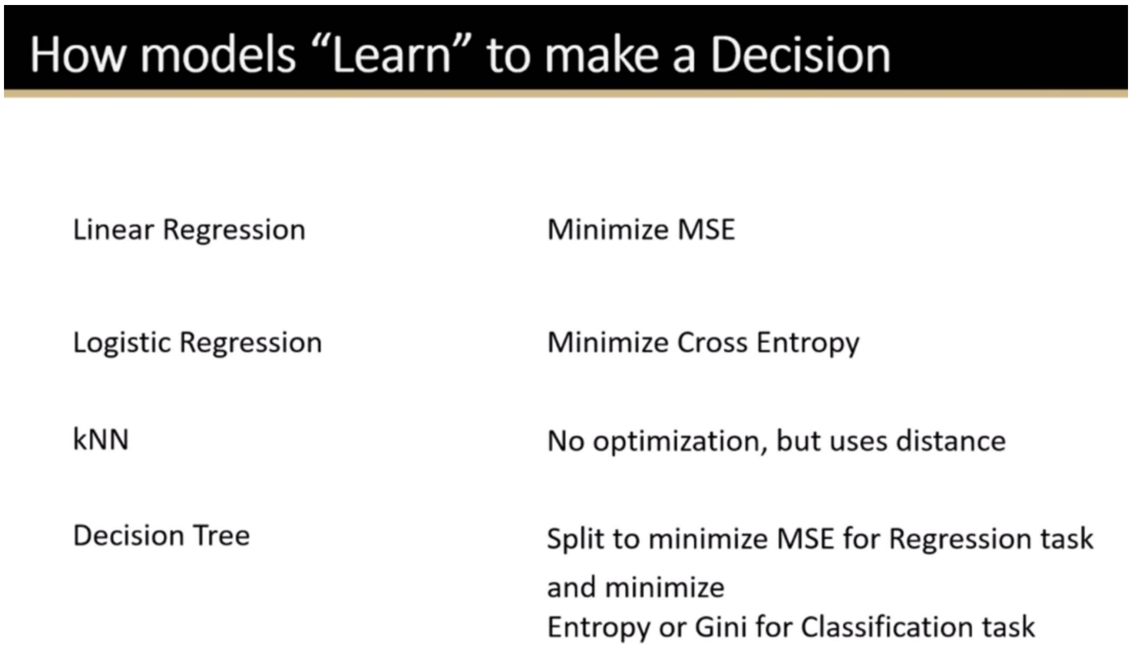
1.20 Decision Tree

Decision trees are a type of supervised learning algorithm used for both classification and regression tasks, though they are more commonly used for classification. They are called "decision trees" because the model uses a tree-like structure of decisions and their possible consequences, including chance event outcomes, resource costs, and utility.



1.20.1 How Decision Trees Work

The algorithm divides the data into two or more homogeneous sets based on the most significant attributes making the groups as distinct as possible. It uses a method called "recursive partitioning" or "splitting" to do this, which starts at the top of the tree (the "root") and splits the data into subsets by making decisions based on feature values. This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the algorithm cannot make any further splits or when it reaches a predefined condition set by the user, such as a maximum tree depth or a minimum number of samples per leaf.

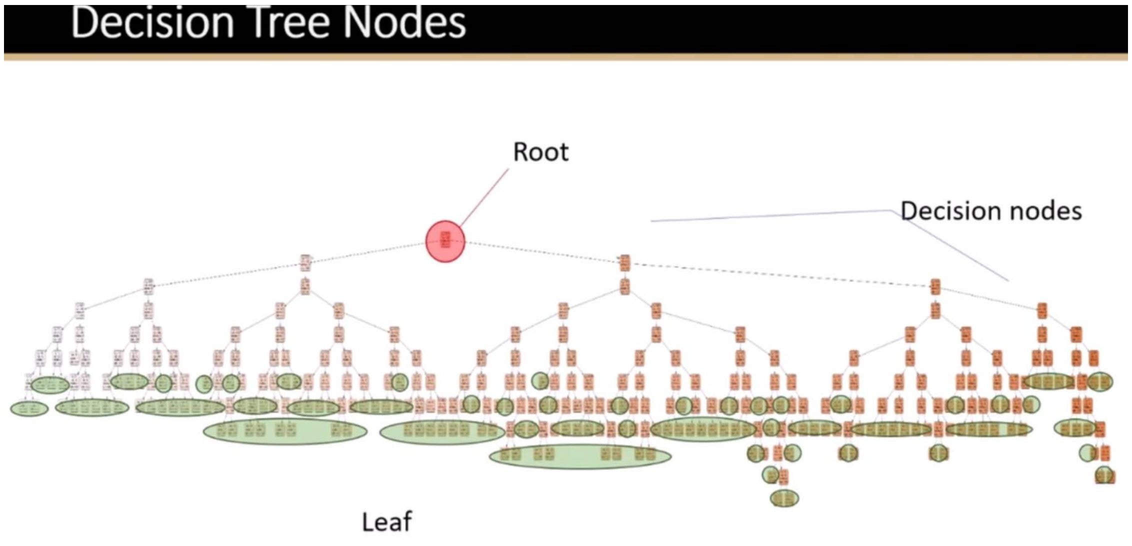


How models "Learn" to make a Decision

Linear Regression	Minimize MSE
Logistic Regression	Minimize Cross Entropy
kNN	No optimization, but uses distance
Decision Tree	Split to minimize MSE for Regression task and minimize Entropy or Gini for Classification task

Components of Decision Trees

- **Root Node**: Represents the entire dataset, which gets divided into two or more homogeneous sets.
- **Splitting**: Process of dividing a node into two or more sub-nodes based on certain conditions.
- **Decision Node**: After splitting, the sub-nodes become decision nodes, where further splits can occur.
- **Leaf/Terminal Node**: Nodes that do not split further, representing the outcome or decision.
- **Pruning**: Reducing the size of decision trees by removing parts of the tree that do not provide additional power to classify instances. This is done to make the tree simpler and to avoid overfitting.



Criteria for Splitting

Decision trees use various metrics to decide how to split the data at each step:

- For classification tasks, commonly used metrics are Gini impurity, Entropy, and Classification Error.
- For regression tasks, variance reduction is often used.

Example

Imagine you want to decide on the activity for your weekend. The decision could depend on multiple factors such as the weather and whether you have company. A decision tree for this scenario might look something like this:

- The root node starts with the question: "Is it raining?"
  - If "Yes", the tree might direct you to a decision "Stay in and read".
  - If "No", it then asks, "Do you have company?"
    - If "Yes", the decision might be "Go hiking".
    - If "No", the decision could be "Visit a cafe".

This example simplifies the decision tree concept. In real-world data science tasks, decision trees consider many more variables and outcomes, and the decisions are based on quantitative data from the features of the dataset.

Advantages and Disadvantages

Advantages:

- Easy to understand and interpret.
- Requires little data preparation.
- Can handle both numerical and categorical data.
- Can handle multi-output problems.

Disadvantages:

- Prone to overfitting, especially with complex trees.
- Can be unstable because small variations in the data might result in a completely different tree being generated.
- Decision boundaries are linear, which may not accurately represent the data's actual structure.

To combat overfitting, techniques like pruning (reducing the size of the tree), setting a maximum depth for the tree, and ensemble methods like Random Forests are often used.

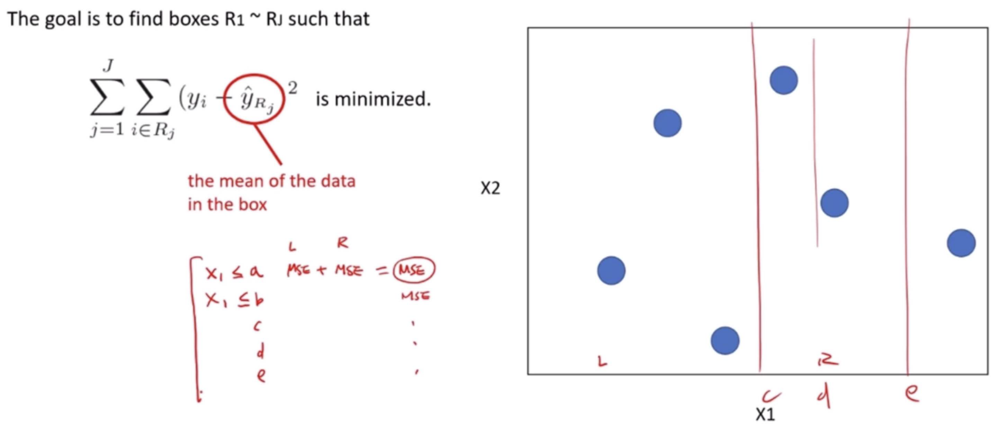
1.20.2 Decision Tree Regressor

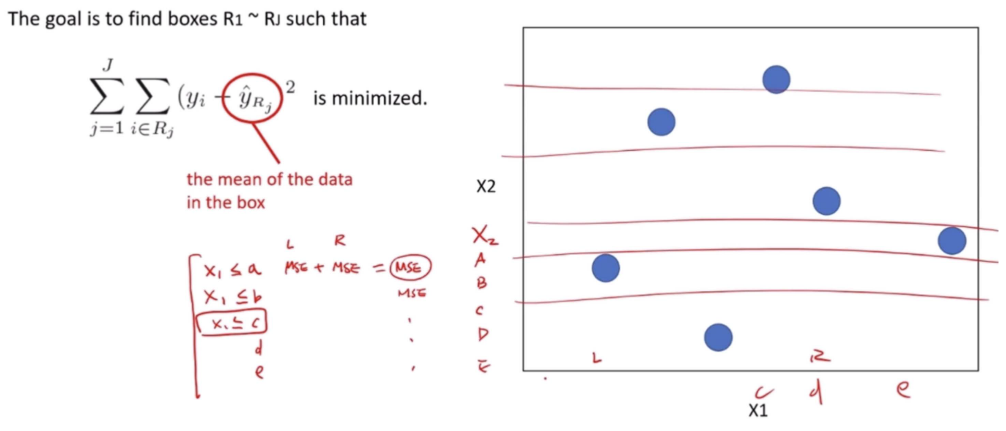
A Decision Tree Regressor is a type of machine learning model used for predicting continuous values, unlike its counterpart, the Decision Tree Classifier, which predicts categorical outcomes. It works by breaking down a dataset into smaller subsets while simultaneously developing an associated decision tree. The final result is a tree with decision nodes and leaf nodes.

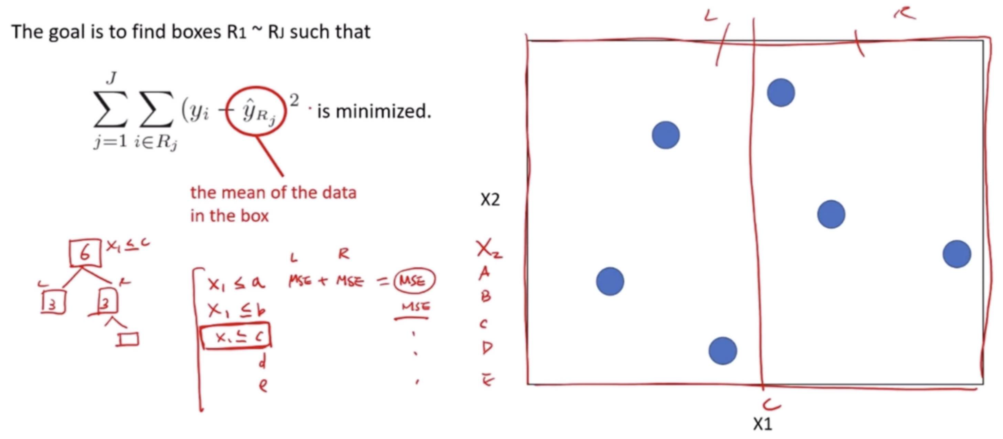
The Decision Tree Regressor uses the Mean Squared Error (MSE) as a measure to decide on the best split at each decision node. MSE is a popular metric used to evaluate the performance of a regression model, indicating the average squared difference between the observed actual outturns and the predictions made by the model. The goal of the regressor is to minimize the MSE at each step of building the tree.

How it Works Using MSE

1. **Starting at the Root**: The entire dataset is considered as the root.
2. **Best Split Decision**: To decide on a split, it calculates the MSE for every possible split in every feature and chooses the one that results in the lowest MSE. This split is the one that, if used to split the dataset into two groups, would result in the most similar responses within each group.
3. **Recursion on Subsets**: This process of finding the best split is then recursively applied to each resulting subset. The recursion is completed when the algorithm reaches a predetermined stopping criterion, such as a maximum depth of the tree or a minimum number of samples required to split a node further.
4. **Prediction**: For a prediction, the input features of a new data point are fed through the decision tree. The path followed by the data point through the tree leads to a leaf node. The average of the values in this leaf node is used as the prediction.


The goal is to find boxes R1 ~ RJ such that
$\sum_{j=1}^{J}\sum_{i\in R_j}(y_i - \hat{y}_{R_j})^2$ is minimized.
the mean of the data in the box
L R
$x_1 \le a$ MSE + MSE = MSE
$x_1 \le b$ MSE
c
d
e
X2
X1
L
R
c d e


The goal is to find boxes R1 ~ RJ such that
$\sum_{j=1}^{J}\sum_{i\in R_j}(y_i - \hat{y}_{R_j})^2$ is minimized.
the mean of the data in the box
L R
$x_1 \le a$ MSE + MSE = MSE
$x_1 \le b$ MSE
$x_1 \le c$
d
e
X2
X2
A
B
C
D
E
X1
L
R
c d e

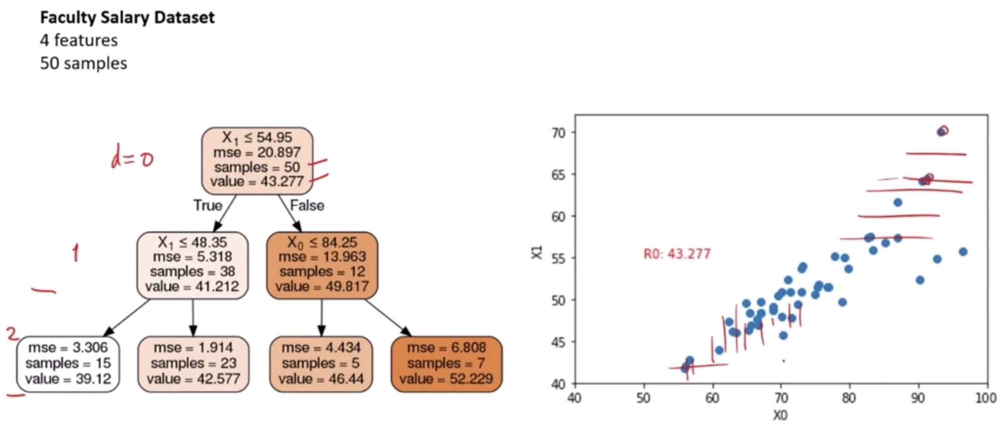

The goal is to find boxes R1 ~ RJ such that
$\sum_{j=1}^{J}\sum_{i\in R_j}(y_i - \hat{y}_{R_j})^2$ is minimized.
the mean of the data in the box
X2
X1
L
R
c
$x_1 \le c$
$x_1 \le a$
$x_1 \le b$
MSE + MSE = MSE

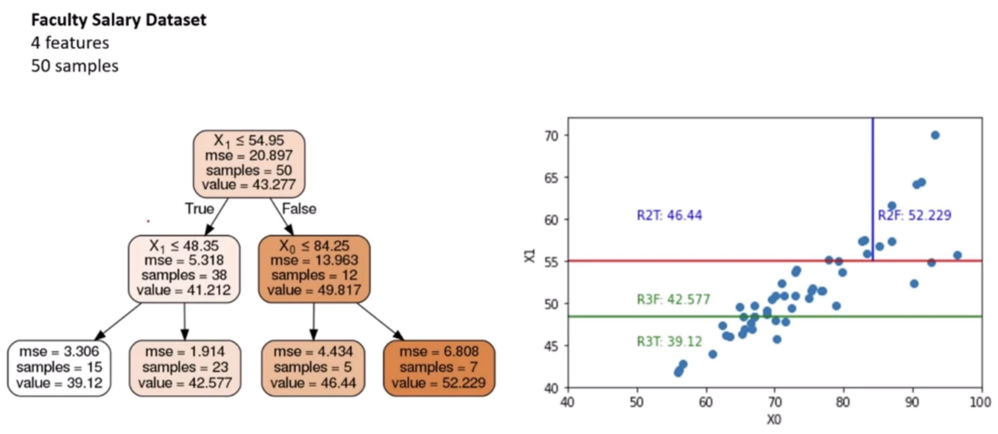
Example

Imagine we are using a dataset of houses, where our features include the number of bedrooms, the number of bathrooms, square footage, and the year built, and our target variable is the house price.

1. **Root**: Initially, the entire dataset is the root.
2. **Best Split Calculation**: The algorithm evaluates all features and their possible values to find the split that would result in subsets with the most similar house prices (lowest MSE). Suppose the best initial split divides the dataset into houses with less than 2 bathrooms and houses with 2 or more bathrooms.
3. **Recursive Splitting**: This process continues, with each subset being split on features and feature values that minimize the MSE within each resulting subset. For instance, within the subset of houses with less than 2 bathrooms, the next split might be on the number of bedrooms.
4. **Stopping Criterion Reached**: Eventually, when the stopping criteria are met (for example, a maximum depth of the tree), the splitting stops.
5. **Making Predictions**: To predict the price of a new house, we would input its features into the decision tree. The house would follow a path down the tree determined by its features until it reaches a leaf node. The prediction would be the average price of the houses in that leaf node.

This example simplifies the complexity involved in building a decision tree regressor but gives an outline of how MSE is used to create a model that can predict continuous outcomes like house prices.


Faculty Salary Dataset
4 features
50 samples
d=0
1
2
$X_1 \le 54.95$ mse = 20.897 samples = 50 value = 43.277
True
False
$X_1 \le 48.35$ mse = 5.318 samples = 38 value = 41.212
$X_0 \le 84.25$ mse = 13.963 samples = 12 value = 49.817
mse = 3.306 samples = 15 value = 39.12
mse = 1.914 samples = 23 value = 42.577
mse = 4.434 samples = 5 value = 46.44
mse = 6.808 samples = 7 value = 52.229
R0: 43.277
X0
X1



1.20.3 Decision Tree Classifier

Decision Tree Classifier is similar to Decision Tree Regressor, with the difference that it classifies the target variable.

A Decision Tree Classifier is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g., "Is the animal you're thinking of larger than a breadbox?"), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules.

How it works:

1. **Starting at the root:** Based on the data (features of animals in our analogy), the algorithm chooses the most significant feature to split the data into groups. This is usually done using measures like Gini impurity or information gain to determine which feature brings us closer to a decision.
2. **Splitting:** This process is repeated for each child node (e.g., if the first question splits the animals into "larger than a breadbox" and "not larger than a breadbox," each of those groups is then split again based on another feature), creating the tree.
3. **Stopping Criteria:** The tree stops growing when it meets a stopping criterion, like when it can no longer reduce uncertainty or when it reaches a specified depth.
4. **Decision Making:** Once built, you can use the tree to classify new cases (predict the class of a new animal) by following the decisions in the tree based on the features of the new case.

Gini Impurity: Simplified

$$H(X_m) = \sum_k p_{mk}(1 - p_{mk})$$

The Gini impurity formula you've provided is a mathematical way to quantify how "pure" a set (usually a set of data points in a decision tree node) is. It tells us what the chance is that a randomly chosen element from the set is incorrectly labeled if it was randomly labeled according to the distribution of classes in the set. Let's break it down:

- $H(X_m)$ is the Gini impurity of a set m.
- p_{mk} is the proportion (or probability) of class k in the set m

The formula sums up the product of the probability of each class with the probability of not being that class (which is $1 - p_{mk}$). This product gives a measure of the probability of misclassification for each class. By summing over all classes, the Gini impurity considers the probability of misclassification regardless of what class we're talking about.

Let's go through a step-by-step example with a tangible dataset.

Example Dataset:

Imagine we have a small dataset of 10 animals, with two features: "Can Fly" (Yes or No) and "Has Fins" (Yes or No). We want to classify them into two classes: "Bird" or "Fish".

Here are the animals:

- 4 are birds that can fly.
- 2 are birds that cannot fly (perhaps penguins).
- 3 are fish that have fins.
- 1 is a fish that does not have fins (maybe an eel).

Step 1: Calculate Class Proportions

We first need to calculate the proportions of each class in the set.

- Birds: $p_{bird} = \frac{4+2}{10} = \frac{6}{10} = 0.6$
- Fish: $p_{fish} = \frac{3+1}{10} = \frac{4}{10} = 0.4$

Step 2: Plug Proportions Into the Formula

Now we use the Gini formula.

- Gini for birds: $p_{bird} \times (1 - p_{bird}) = 0.6 \times (1 - 0.6) = 0.6 \times 0.4 = 0.24$
- Gini for fish: $p_{fish} \times (1 - p_{fish}) = 0.4 \times (1 - 0.4) = 0.4 \times 0.6 = 0.24$

Step 3: Sum the Gini for All Classes

The Gini impurity for the set is the sum of the Gini for all classes.

- Total Gini impurity: $H(X_m) = 0.24 + 0.24 = 0.48$

This Gini impurity value of 0.48 is relatively high, indicating that the set is quite mixed (if it were 0, the set would be perfectly pure).

Step 4: Interpretation

A Gini impurity of 0.48 means that if we pick a random animal from this dataset and then randomly assign a label based on the distribution of the classes (60% chance of being labeled as a bird and 40% as a fish), there's a 48% chance of mislabeling the animal.

The goal in a decision tree is to create splits that result in subsets with lower Gini impurity scores compared to the parent node. A perfect split would be one that results in nodes with a Gini impurity of 0, meaning all elements in each node are from a single class.

- **Gini Impurity = 1 - sum (probability of each class)^2**

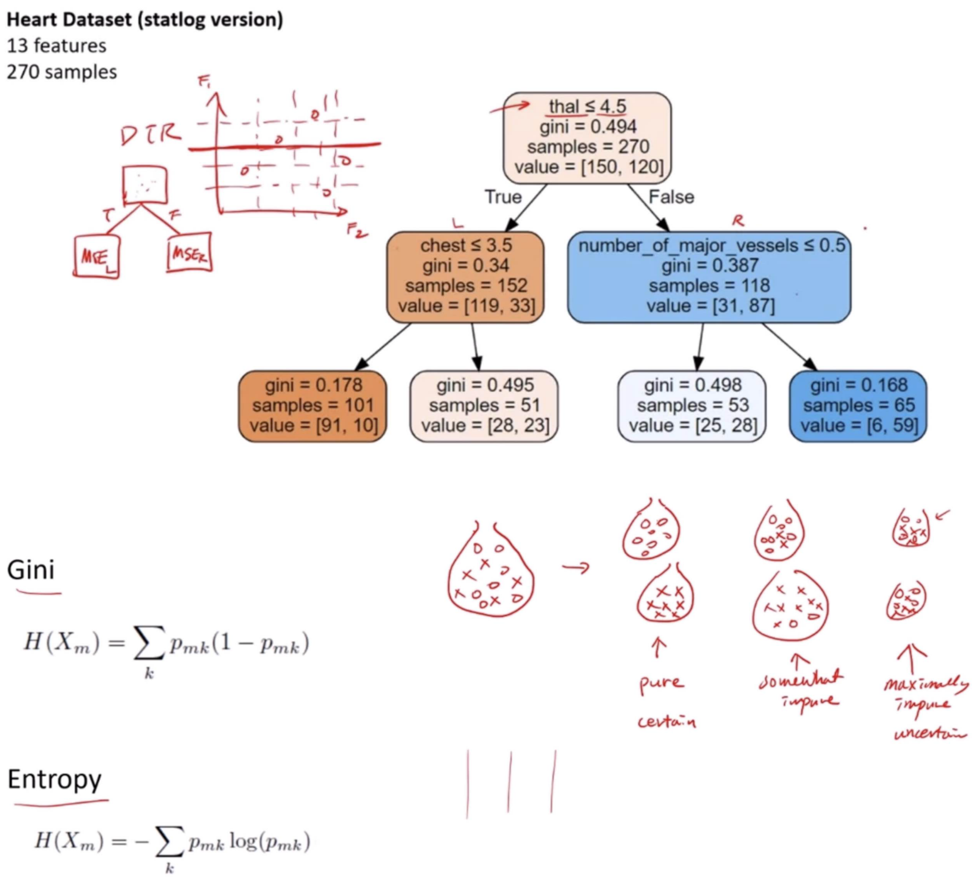
The best (lowest) Gini impurity is 0, where all elements belong to a single class (pure). The worst (highest) Gini impurity is 0.5 in a binary classification, where the dataset is evenly split between two classes (completely mixed).

Example:

Let's say we have a dataset of 10 animals, 6 are fish and 4 are birds.

- The probability of picking a fish randomly is 6/10, and the probability of picking a bird is 4/10.
- Gini Impurity = 1 - ((6/10)^2 + (4/10)^2) = 1 - (0.36 + 0.16) = 1 - 0.52 = 0.48

This Gini score tells us how often we would be wrong if we randomly assigned a label to an animal in this group based on the distribution of classes. A lower score is better, indicating less impurity.



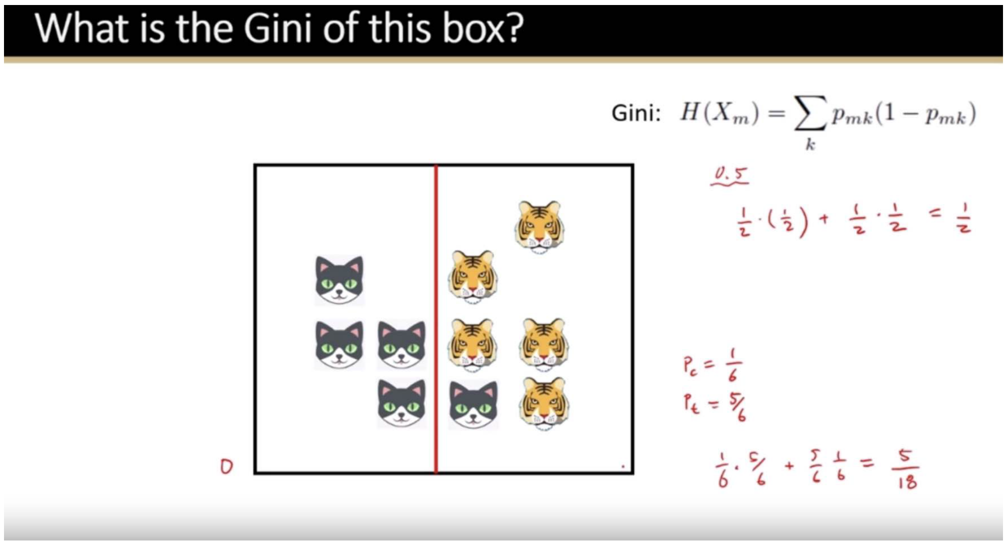
Heart Dataset (statlog version)
13 features
270 samples

Gini

$$H(X_m) = \sum_k p_{mk}(1 - p_{mk})$$

Entropy

$$H(X_m) = -\sum_k p_{mk}\log(p_{mk})$$


What is the Gini of this box?
Gini: $H(X_m) = \sum_k p_{mk}(1 - p_{mk})$
0.5
$\frac{1}{2} \cdot (\frac{1}{2}) + \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2}$
$P_c = \frac{1}{6}$
$P_t = \frac{5}{6}$
$\frac{1}{6} \cdot \frac{5}{6} + \frac{5}{6} \frac{1}{6} = \frac{5}{18}$
D

We'll apply the Gini impurity formula:

$$Gini : H(X_m) = \sum_k p_{mk}(1 - p_{mk})$$

For the given probabilities of the two classes:

$p_{\text{class1}} = 0.77777777777$ $p_{\text{class2}} = 0.22222222222$

The Gini impurity for each class is calculated as:

For class 1: $H(X_{m1}) = p_{\text{class1}}(1 - p_{\text{class1}}) = 0.77777777777 \times (1 - 0.77777777777)$

For class 2: $H(X_{m2}) = p_{\text{class2}}(1 - p_{\text{class2}}) = 0.22222222222 \times (1 - 0.22222222222)$

Adding the Gini impurity of both classes:

$$\begin{aligned}
H(X_m) &= H(X_{m1}) + H(X_{m2}) \\
H(X_m) &= 0.77777777777 \times 0.22222222223 + 0.22222222222 \times 0.77777777778 \\
H(X_m) &= 0.17283950617 + 0.17283950616 \\
H(X_m) &= 0.34567901233
\end{aligned}$$

The Gini impurity for the binary classification with the given class probabilities is approximately 0.3457.

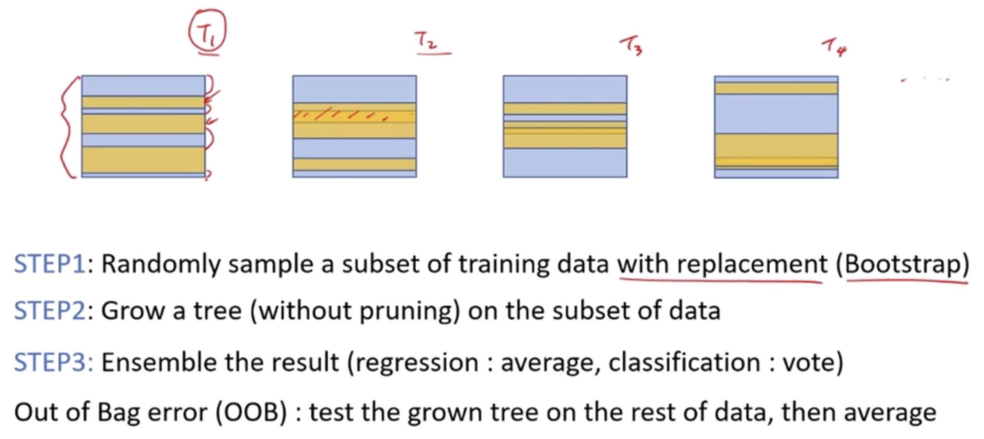
1.20.4 Ensemble Methods

Ensemble methods are techniques that combine multiple models to improve the robustness and accuracy of predictions. These methods work on the principle that a group of "weak learners" can come together to form a "strong learner."

Bagging (Bootstrap Aggregating)

Bagging reduces variance and helps to avoid overfitting. It involves creating multiple versions of a predictor and using these to get an aggregated predictor. The steps are:

- Randomly create "bootstrap" samples of the original dataset (with replacement).
- Train a model on each of these samples.
- Combine the models using the average (regression) or majority vote (classification).



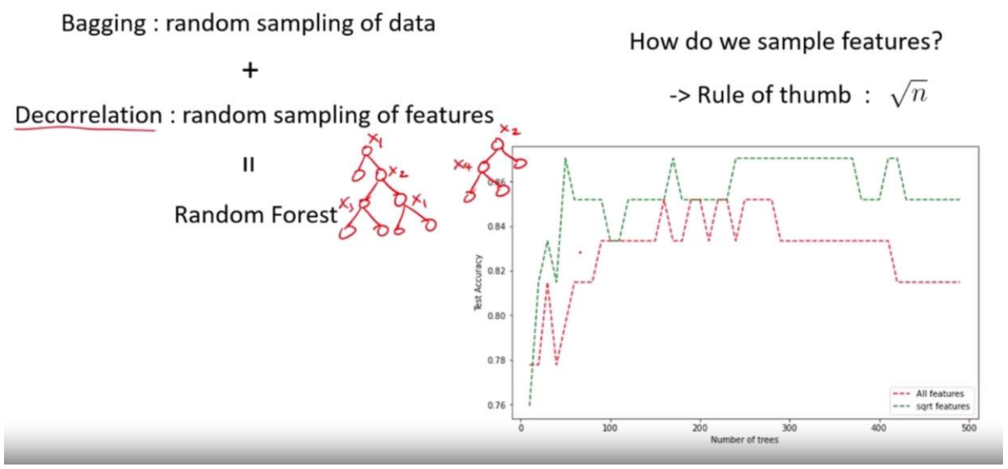
STEP1: Randomly sample a subset of training data with replacement (Bootstrap)

STEP2: Grow a tree (without pruning) on the subset of data

STEP3: Ensemble the result (regression : average, classification : vote)

Out of Bag error (OOB) : test the grown tree on the rest of data, then average

Example: Random Forest is a popular bagging ensemble of decision trees. Each tree is built on a bootstrap sample of the data, and the final prediction is averaged across all trees.



Bagging : random sampling of data

+

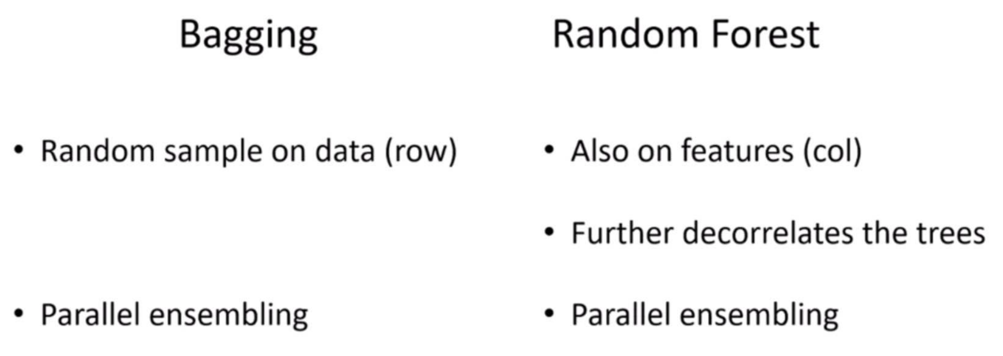
Decorrelation : random sampling of features

=

Random Forest

How do we sample features?

-> Rule of thumb : $\sqrt{n}$



Bagging

- Random sample on data (row)
- Parallel ensembling

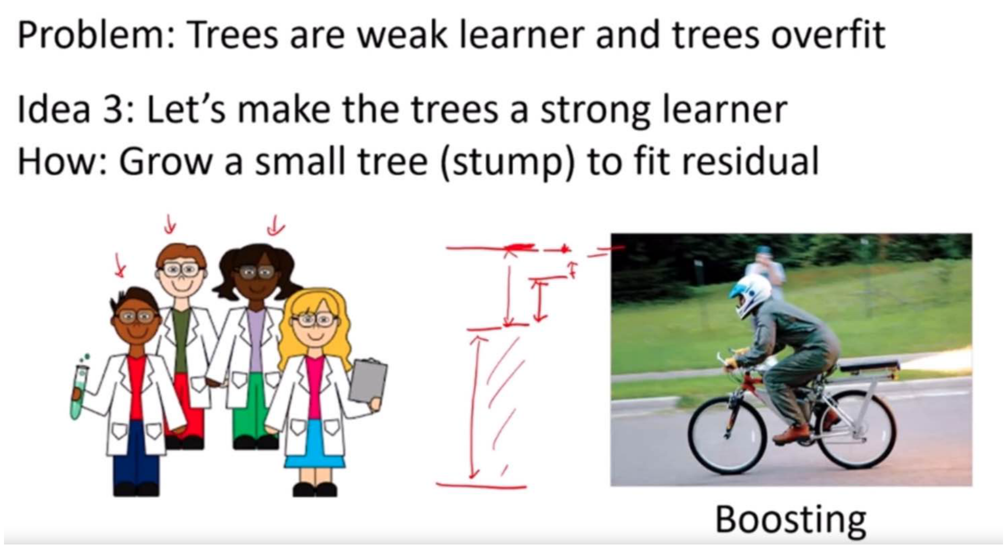
Random Forest

- Also on features (col)
- Further decorrelates the trees
- Parallel ensembling

Boosting

Boosting focuses on converting weak learners into strong learners sequentially. Each model attempts to correct the errors made by the previous ones. The steps include:

- Train a model on the entire dataset.
- Build another model to correct the errors of the first model.
- Continue adding models until a limit is reached or no further improvements can be made.
- The final model makes predictions based on the weighted sum of the predictions of all models.



Problem: Trees are weak learner and trees overfit

Idea 3: Let's make the trees a strong learner
How: Grow a small tree (stump) to fit residual

Boosting

Examples: AdaBoost (Adaptive Boosting) and Gradient Boosting are well-known boosting methods. AdaBoost adjusts the weights of incorrectly classified instances so that subsequent classifiers focus more on difficult cases.

Advantages of Ensemble Methods

- **Improved Accuracy:** Combining models often yields better results than any single model.
- **Reduced Overfitting:** Especially with bagging and stacking, since they average out biases.
- **Increased Robustness:** The ensemble's prediction is less sensitive to noise and outliers.

Disadvantages of Ensemble Methods

- **Increased Complexity:** More parameters to tune and a more complicated model to explain.
- **Higher Computational Cost:** Training multiple models is computationally more expensive.
- **Risk of Overfitting with Boosting:** Especially if the dataset is noisy.

Ensemble methods are widely used in various applications, from competitive machine learning competitions to practical business problems, because of their ability to improve prediction accuracy and model robustness.

1.20.5 Adaboost

The AdaBoost (Adaptive Boosting) algorithm is a powerful ensemble technique used to improve the performance of binary classifiers. It combines multiple weak learners (a learner that performs slightly better than random guessing) to create a strong classifier. Here's a detailed breakdown of how AdaBoost works and the key formula involved:

How AdaBoost Works:

1. **Initialization**: Each observation in the dataset is initially given an equal weight. This signifies that at the start, every instance is equally important for the model.
2. **Iterative Training**:
   - **Weak Learner Training**: In each round, a weak learner is trained on the dataset. The goal of this learner is only to be slightly better than random guessing, regardless of its complexity.
   - **Error Calculation**: After training a weak learner, AdaBoost calculates the error rate of the learner. This error is weighted based on the weights of the observations. The error rate (ϵ) is essentially the sum of the weights of the incorrectly classified observations divided by the sum of all weights.
   - **Learner Weight Calculation**: AdaBoost assigns a weight (α) to the weak learner based on its accuracy. More accurate learners are given more weight. The weight is calculated using the formula: $\alpha = \frac{1}{2}\ln\left(\frac{1-\epsilon}{\epsilon}\right)$, where $\ln$ is the natural logarithm.
   - **Update Weights**: The weights of the observations are updated so that the weights of the incorrectly classified observations are increased, and the weights of the correctly classified observations are decreased. This makes the algorithm focus more on the harder-to-classify instances in the next round.
   - **Normalization**: The updated weights are normalized so that their sum is 1.
3. **Final Model**:
   - After a specified number of rounds, or if perfect prediction is achieved, AdaBoost combines the weak learners into a final model. The final model makes predictions based on a weighted majority vote (or sum) of the weak learners' predictions. Each weak learner's vote is weighted by its α value.

Key Formulae:

- **Error of a Weak Learner**: $\epsilon = \frac{\sum_{i=1}^{N} w_i \cdot \text{error}_i}{\sum_{i=1}^{N} w_i}$, where w_i is the weight of the i^{th} observation, and error_i is an indicator function that is 1 if the observation is misclassified and 0 otherwise.
- **Weight of a Weak Learner**: $\alpha = \frac{1}{2}\ln\left(\frac{1-\epsilon}{\epsilon}\right)$.
- **Weight Update Rule**: For each observation, the new weight (w_i') is updated using the rule: $w_i' = w_i \cdot e^{\alpha \cdot \text{error}_i}$, followed by a normalization step.

Conclusion:

AdaBoost is effective because it focuses on the observations that are hard to classify and gives more importance to the more accurate learners. This adaptiveness to the errors of the learners makes it a powerful algorithm for classification problems. The sequential process of adjusting weights puts more focus on instances that are hard to predict, leading to a highly accurate ensemble model.

1.20.6 Gradient Boosting

Gradient Boosting is a powerful machine learning algorithm that's used for both regression and classification problems. It builds a model in a stage-wise fashion like AdaBoost, but it generalizes the boosting process by allowing optimization of an arbitrary differentiable loss function.

How Gradient Boosting Works:

1. **Initialization**: It begins with a simple model (often a decision tree), which could be just a prediction of the mean of the target values. This initial model serves as the base upon which further models (often called trees) are added.
2. **Sequential Tree Building**:
   - For each iteration, the algorithm first calculates the residuals or errors of the current model. In the context of regression, these are simply the differences between the predicted and actual values. In classification, the process is slightly more complex, involving the gradient of the loss function.
   - A new model (tree) is then trained to predict these residuals or gradients. Essentially, instead of directly predicting the target value or class, each new model aims to correct the mistakes of the combined ensemble of all previous models.
   - Once the new model is trained on the residuals, it is added to the ensemble. The idea is to take a step in the direction that minimizes the loss, akin to the gradient descent optimization algorithm, hence the name "Gradient Boosting."
3. **Loss Function Optimization**:
   - Gradient Boosting involves the minimization of a loss function. The loss function quantifies how far off a prediction is from the actual result. The choice of loss function depends on the type of problem (regression, classification, etc.).
   - After each tree is added, the algorithm updates the predictions for each observation, effectively taking a step that reduces the loss (error).
4. **Shrinkage (Learning Rate)**:
   - To prevent overfitting, Gradient Boosting introduces a learning rate (also known as "shrinkage") that scales the contribution of each tree. A smaller learning rate requires more trees to model the data but generally results in a more robust model.

Example:

Imagine we're trying to predict the price of houses based on features like size, location, and number of bedrooms. Our initial model might predict that each house costs $300,000, which is the average price of all houses in the training set.

1. **First Iteration**: We calculate the residuals (actual price - predicted price) for each house. Then, we train a decision tree to predict these residuals based on our features.
2. **Update Predictions**: We add this new tree to our model, adjusting our predictions for each house. Suppose this tree predicts that houses with 3 bedrooms are undervalued by $50,000. Our updated predictions for 3-bedroom houses would increase by this amount.
3. **Repeat**: We calculate new residuals based on our updated predictions and train a new tree to predict these residuals. This process repeats, with each new tree correcting the errors of the ensemble so far.
4. **Final Model**: After a specified number of iterations, or once our loss has been minimized to an acceptable level, we combine all the trees to make final predictions.

Conclusion:

Gradient Boosting is a versatile and powerful technique capable of handling both regression and classification problems. Its sequential nature, focusing on correcting its own errors, makes it highly effective, though it can be prone to overfitting if not properly regularized or if too many trees are used. The learning rate and the number of trees are crucial hyperparameters that need careful tuning to achieve the best performance.

1.21 What is Deep Learning

1.21.1 Overview

Deep Learning

a type of machine learning based on artificial neural networks in which multiple layers of processing are used to extract progressively higher level features from data.

Machine Learning

development of computer systems that can learn to more accurately predict the outcomes without following explicit instructions, by using algorithms and statistical models to draw inferences from patterns in data.

Differences between Deep Learning and Machine Learning

Machine Learning

- uses algorithms to parse data, learn from that data, and make informed decisions based on what it has learned.
- needs a human to identify and hand-code the applied features based on the data type. | tries to learn features extraction and representation as well.
- tend to parse data in parts, then combined those into a result (e.g. first number plate localization and then recognition).
- requires relatively less data and training time

Deep learning

- structures algorithms in layers to create an "artificial neural network" that can learn and make intelligent decisions on its own.
- tries to learn features extraction and representation as well.
- Deep learning systems look at an entire problem and generate the final result in one go (e.g. outputs the coordinates and the class of object together).
- requires a lot more data and training time

1.21.2 Applications Of Machine Learning/Deep Learning

- Email spam detection
- Fingerprint / face detection & matching (e.g., phones)
- Web search (e.g., DuckDuckGo, Bing, Google)
- Sports predictions
- ATMs (e.g., reading checks)
- Credit card fraud
- Stock predictions

1.21.3 Broad categories of Deep learning

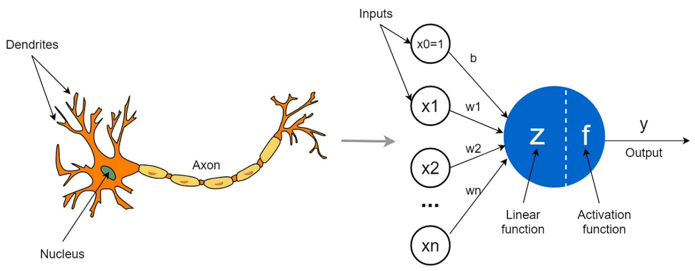
1.21.4 Perceptron

Definition

Simplest artificial neuron that takes binary inputs and based on their weighted sum reaching a threshold, generates a binary output.

Artificial neurons

- Takes in multiple inputs and learns what should be the appropriate output
- Essentially a mathematical function where the weights multiplied with the inputs are learnable
- Acts like a logic gate but the operation performed adjusts according to the data



- connect them in a network to create an artificial brain(let)

History of the Perceptron

- Invented in 1957 by Frank Rosenblatt to binary classify an input data.
- An attempt to replicate the process and ability of human nervous system.

A Biological Neuron

McCulloch & Pitts Neuron Model

Computational Model of a Biological Neuron

Terminology

- Net input $=$ weighted inputs, z
- Activations = activation function(net input); $a = \sigma(z)$
- Label output $=$ threshold(activations of last layer); $\hat{y} = f(a)$

Special cases:

- In perceptron: activation function = threshold function
- In linear regression: activation $=$ net input $=$ output

$$\sigma\left(\sum_{i=1}^{m} x_i w_i + b\right) = \sigma\left(\mathbf{x}^T \mathbf{w} + b\right) = \hat{y}$$

Often more convenient notation: define bias unit as w_0 and prepend a 1 to each input vector as an additional feature value

$$\sigma\left(\sum_{i=0}^{m} x_i w_i\right) = \sigma\left(\mathbf{x}^\top \mathbf{w}\right) = \hat{y}$$

Perceptron Learning Algorithm

Let $\mathcal{D} = \left(\left\langle \mathbf{x}^{[1]}, y^{[1]} \right\rangle, \left\langle \mathbf{x}^{[2]}, y^{[2]} \right\rangle, \ldots, \left\langle \mathbf{x}^{[n]}, y^{[n]} \right\rangle\right) \in \left(\mathbb{R}^m \times \{0,1\}\right)^n$

1. Initialize $\mathbf{w} := 0^m \quad$ (assume notation where weight incl. bias)
2. For every training epoch:
   - For every $\left\langle \mathbf{x}^{[i]}, y^{[i]} \right\rangle \in \mathcal{D}$:
     1. $\hat{y}^{[i]} := \sigma\left(\mathbf{x}^{[i]\top} \mathbf{w}\right)$
     2. err $:= \left(y^{[i]} - \hat{y}^{[i]}\right)$
     3. $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$

Vectorization in Python

Running Computations is a Big Part of Deep Learning!

```python
import torch

def forloop(x, w):
    z = 0.
    for i in range(len(x)):
        z += x[i] * w[i]
    return z
```

(continues on next page)

(continued from previous page)

```
def listcomprehension(x, w):
    return sum(x_i*w_i for x_i, w_i in zip(x, w))


def vectorized(x, w):
    return x.dot(w)


x, w = torch.rand(1000), torch.rand(1000)

%timeit -r 10 -n 10 forloop(x, w)

%timeit -r 10 -n 10 listcomprehension(x, w)

%timeit -r 10 -n 10 vectorized(x, w)
```

```
8.64 ms ± 60.2 µs per loop (mean ± std. dev. of 10 runs, 10 loops each)
```

```
7.42 ms ± 47.1 µs per loop (mean ± std. dev. of 10 runs, 10 loops each)
The slowest run took 11.05 times longer than the fastest. This could mean that an␣
↪intermediate result is being cached.
6.61 µs ± 9.7 µs per loop (mean ± std. dev. of 10 runs, 10 loops each)
```

Perceptron Pytorch Implementation

Label data

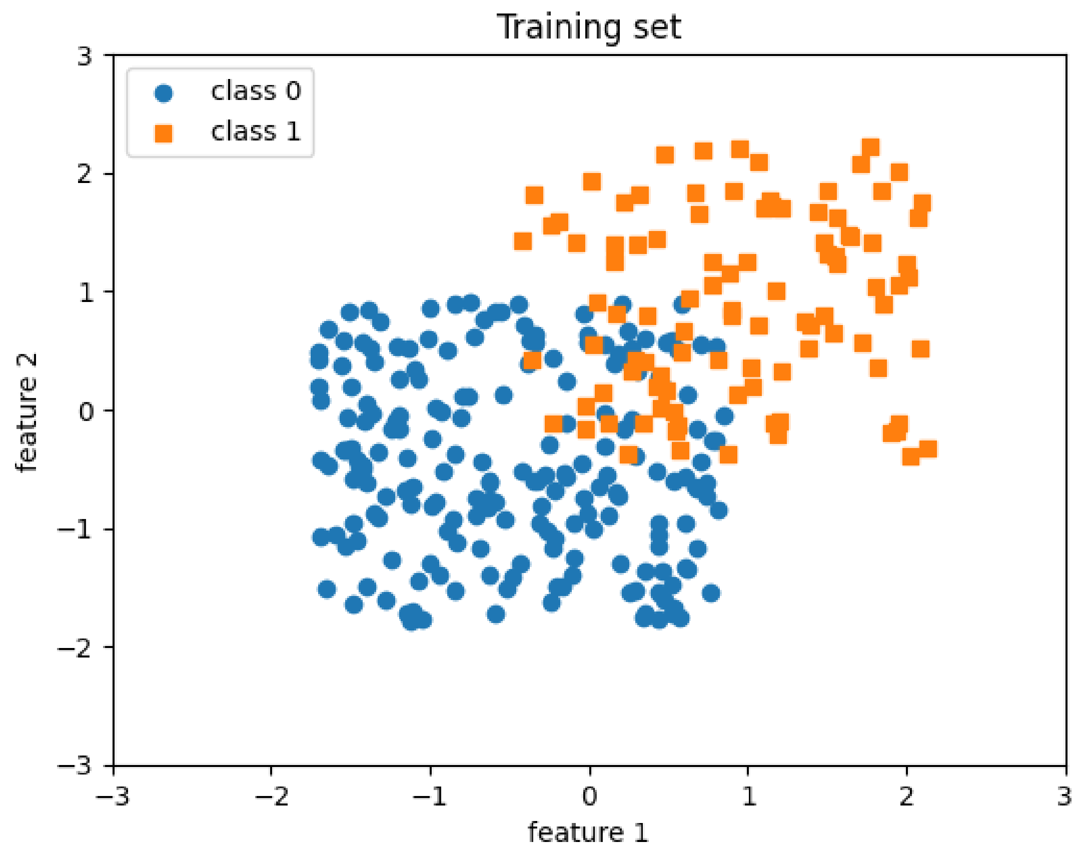
```
import torch
import matplotlib.pyplot as plt

c1_mean , c2_mean = -0.5 , 0.5

c1 = torch.distributions.uniform.Uniform(c1_mean-1,c1_mean+1).sample((200,2))
c2 = torch.distributions.uniform.Uniform(c2_mean-1,c2_mean+1).sample((200,2))
features = torch.cat([c1,c2], axis=0)

labels = torch.cat([torch.zeros((200,1)), torch.ones((200,1))], axis = 0)
data = torch.cat([features, labels],axis=1)

X, y = data[:, :2], data[:, 2]
y = y.to(torch.int)

print('X.shape:', X.shape)
print('y.shape:', y.shape)


X_train, X_test = X[:300], X[100:]
y_train, y_test = y[:300], y[100:]
```

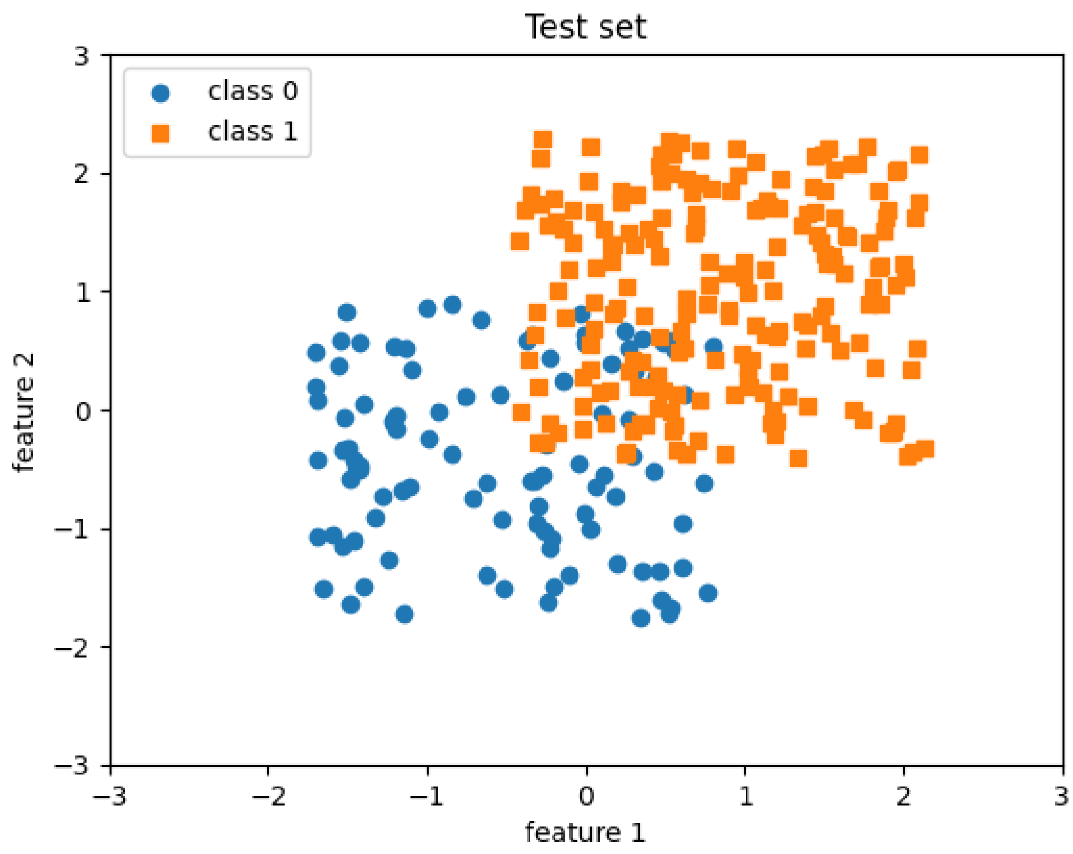
(continues on next page)

(continued from previous page)

```
# Normalize (mean zero, unit variance)
mu, sigma = X_train.mean(axis=0), X_train.std(axis=0)
X_train = (X_train - mu) / sigma
X_test = (X_test - mu) / sigma

plt.scatter(X_train[y_train==0, 0], X_train[y_train==0, 1], label='class 0', marker='o')
plt.scatter(X_train[y_train==1, 0], X_train[y_train==1, 1], label='class 1', marker='s')
plt.title('Training set')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.legend()
plt.show()


plt.scatter(X_test[y_test==0, 0], X_test[y_test==0, 1], label='class 0', marker='o')
plt.scatter(X_test[y_test==1, 0], X_test[y_test==1, 1], label='class 1', marker='s')
plt.title('Test set')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.legend()
plt.show()
```

```
X.shape: torch.Size([400, 2])
y.shape: torch.Size([400])
```


Training set
class 0
class 1
feature 1
feature 2
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3



Train and evaluate

```
import torch
device = torch.device("cuda:0" if torch.cuda.is_available() else "mps")

class Perceptron:
    def __init__(self, num_features):
        self.num_features = num_features
        self.weights = torch.zeros(num_features, 1,
                                   dtype=torch.float32, device=device)
        self.bias = torch.zeros(1, dtype=torch.float32, device=device)

        self.ones = torch.ones((1, 1), device=device)
        self.zeros = torch.zeros((1, 1), device=device)

    def forward(self, x):
        linear = torch.mm(x, self.weights) + self.bias
        predictions = torch.where(linear > 0., self.ones, self.zeros)
        return predictions

    def backward(self, x, y):
        predictions = self.forward(x)
        errors = y - predictions
```

(continues on next page)

(continued from previous page)

```
        return errors

    def train(self, x, y, epochs):
        for e in range(epochs):

            for i in range(y.shape[0]):
                errors = self.backward(x[i].reshape(1, self.num_features), y[i]).
↪reshape(-1)
                self.weights += (errors * x[i]).reshape(self.num_features, 1)
                self.bias += errors

    def evaluate(self, x, y):
        predictions = self.forward(x).reshape(-1)
        accuracy = torch.sum(predictions == y).float() / y.shape[0]
        return accuracy



ppn = Perceptron(num_features=2)

X_train_tensor = torch.tensor(X_train, dtype=torch.float32, device=device)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32, device=device)

ppn.train(X_train_tensor, y_train_tensor, epochs=5)

print('Model parameters:')
print('Weights: %s' % ppn.weights)
print('Bias: %s' % ppn.bias)

X_test_tensor = torch.tensor(X_test, dtype=torch.float32, device=device)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32, device=device)

test_acc = ppn.evaluate(X_test_tensor, y_test_tensor)
print('Test set accuracy: %.2f%%' % (test_acc*100))
```

1.22 Vectors, Matrices, and Tensors

Scalars, vectors, matrices, and tensors are the fundamental data structures of deep learning. In this section, we will briefly review these concepts.

Scalar

rank-0 tensor
$x \in \mathbb{R}$
x = 1.23

Vector

rank-1 tensor

$x \in \mathbb{R}^n x1$

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

Matrix

rank-2 tensor
$x \in \mathbb{R}^{nxm}$

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & \dots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \dots & x_{m,n} \end{bmatrix}$$

```
import torch

t = torch.tensor([ [1, 2, 3, 4,], [6, 7, 8, 9] ])

print(t)
print(t.shape)
print(t.ndim)
print(t.dtype)
```

```
tensor([[1, 2, 3, 4],
        [6, 7, 8, 9]])
torch.Size([2, 4])
2
torch.int64
```

1.22.1 Data onto the GPU

```
print(torch.cuda.is_available())
print(torch.backends.mps.is_available())

if torch.cuda.is_available():
  t = t.to(torch.device('cuda:0'))
  print(t)
```

```
False
False
```

1.22.2 Broadcasting

Making Vector and Matrix computations more convenient

Computing the Output From Multiple Training Examples at Once

- The perceptron algorithm is typically considered an "online" algorithm (i.e., it updates the weights after each training example)
- However, during prediction (e.g., test set evaluation), we could pass all data points at once (so that we can get rid of the "forloop")
- Two opportunities for parallelism:
    1. computing the dot product in parallel
    2. computing multiple dot products at once

```
import torch

X = torch.arange(6).view(2, 3)

print(X)

w = torch.tensor([1, 2, 3])

print(w)

print(X.matmul(w))

w = w.view(-1, 1)

print(X.matmul(w))
```

```
tensor([[0, 1, 2],
        [3, 4, 5]])
tensor([1, 2, 3])
tensor([ 8, 26])
tensor([[ 8],
        [26]])
```

This (general) feature is called "broadcasting"

```
print(torch.tensor([1, 2, 3]) + 1)

t = torch.tensor([[4, 5, 6], [7, 8, 9]])

print(t)

print( t + torch.tensor([1, 2, 3]))
```

```
tensor([2, 3, 4])
tensor([[4, 5, 6],
        [7, 8, 9]])
```

(continues on next page)

(continued from previous page)

```
tensor([[ 5,  7,  9],
        [ 8, 10, 12]])
```

1.22.3 Notational Linear Algebra

```
X = torch.arange(50, dtype=torch.float).view(10, 5)

print(X)

fc = torch.nn.Linear(in_features=5, out_features=3)

print(fc.weight)

print(fc.bias)

print(f"X dim: {X.size()}")
print(f"Weights dim: {fc.weight.size()}")
print(f"bias dim: {fc.bias.size()}")

A = fc(X)

print(f"A dim: {A.size()}")
```

```
tensor([[ 0.,  1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.,  9.],
        [10., 11., 12., 13., 14.],
        [15., 16., 17., 18., 19.],
        [20., 21., 22., 23., 24.],
        [25., 26., 27., 28., 29.],
        [30., 31., 32., 33., 34.],
        [35., 36., 37., 38., 39.],
        [40., 41., 42., 43., 44.],
        [45., 46., 47., 48., 49.]])
Parameter containing:
tensor([[ 0.2740,  0.2539, -0.3403, -0.2706,  0.3701],
        [ 0.1425,  0.3340,  0.3780,  0.0567, -0.3553],
        [ 0.3390,  0.0116,  0.3906,  0.3216, -0.3848]], requires_grad=True)
Parameter containing:
tensor([-0.3861, -0.3613,  0.3297], requires_grad=True)
X dim: torch.Size([10, 5])
Weights dim: torch.Size([3, 5])
bias dim: torch.Size([3])
A dim: torch.Size([10, 3])
```

1.23 Loss Functions

1.23.1 Introduction

Imagine the scenario, Once you developed your machine learning model that you believe, successfully identifying the cats and dogs but how do you know this is the best result?

we are looking for the metrics or a function that we can use to optimize our model performance.The loss function tells how good your model is in predictions.

- If the model predictions are closer to the actual values the Loss will be minimum.
- if the predictions are totally away from the original values the loss value will be the maximum.

$$Loss = abs(predict\breve{}actual)$$

On the basis of the Loss value, you can update your model until you get the best result.

1.23.2 Classification

Cross-Entropy or Log Loss

Cross entropy loss is used mostly when we have a binary classification problem; that is, where the network outputs either 1 or 0.

Suppose we are given a training dataset, $\mathbb{D} = \{(x_i, y_i), \cdots, (x_N, y_N)\}$ and $y_i \in \{0, 1\}$.
We can then write this in the following form:

$$\hat{y}_i = f(x_i; \theta)$$

Here, θ is the parameters of the network (weights and biases). We can express this in terms of a Bernoulli distribution, as follows:

$$P(x_i \to y_i \mid \theta) = \hat{y}_i^{y_i}(1 - \hat{y}_i)^{1-y_i}$$

The probability, given the entire dataset, is then as follows:

$$P(x_1, \cdots, x_N, y_1, \cdots, y_N) = \prod_{i=1}^{N} P(x_i \to y_i \mid \theta) = \prod_{i=1}^{N} \hat{y}_i^{y_i}(1 - \hat{y}_i)^{1-y_i}$$

If we take its negative-log likelihood, we get the following:

$$-\log P(x_1, \cdots, x_N, y_1, \cdots, y_N) = -\log \prod_{i=1}^{N} \hat{y}_i^{y_i}(1 - \hat{y}_i)^{1-y_i}$$

So, we have the following:

$$L(\hat{y}, y) = -\sum_{i=1}^{N} y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)$$

Cross-entropy loss

Cross entropy loss is a metric used to measure how well a classification model in machine learning performs. The loss (or error) is measured as a number between 0 and 1, with 0 being a perfect model. The goal is generally to get your model as close to 0 as possible.

Cross entropy loss measures the difference between the discovered probability distribution of a machine learning classification model and the predicted distribution.

1.24 Evaluation Metrics

Evaluation metrics are used to evaluate the performance of a machine learning model. They provide a way to quantitatively measure how well the model is performing on a given task.

1.24.1 Classification

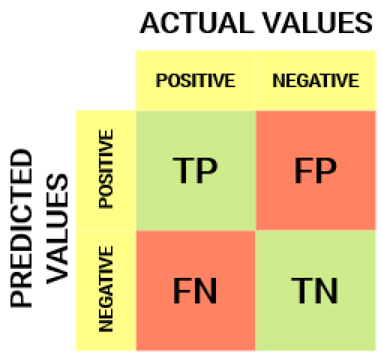
In machine learning, classification is a supervised learning problem in which the model is trained to predict the class of an input data point.

Note: It is important to choose an appropriate evaluation metric for your problem. For example, in a binary classification problem, you may be more interested in minimizing false negatives than false positives, in which case you would want to use a metric like recall rather than precision.

Confusion Matrix

Confusion matrix is a performance measurement for machine learning classification problem where output can be two or more classes.

In a confusion matrix, the rows represent the actual class labels, and the columns represent the predicted class labels.

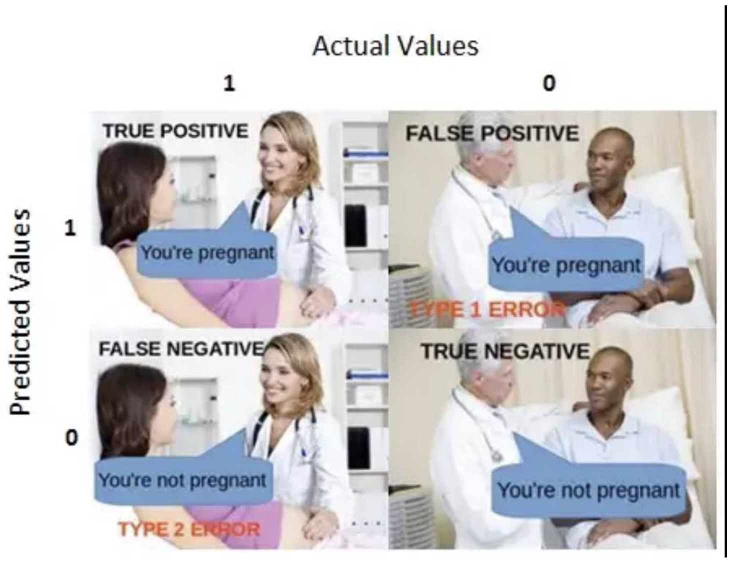


Here, TP (true positive) is the number of times the classifier predicted "positive" and the actual label was "positive". FP (false positive) is the number of times the classifier predicted "positive" and the actual label was "negative". FN (false negative) is the number of times the classifier predicted "negative" and the actual label was "positive". TN (true negative) is the number of times the classifier predicted "negative" and the actual label was "negative".

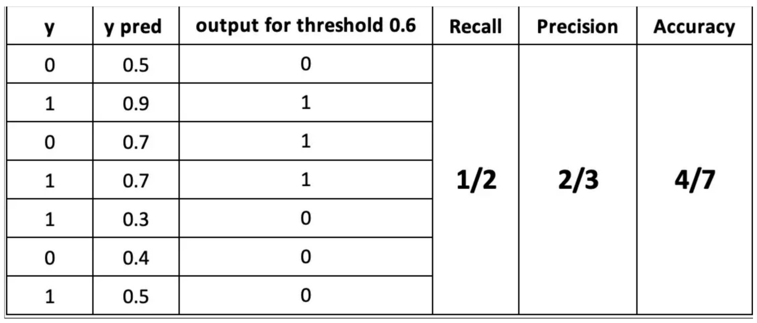
Here is an example of how to compute the values in a confusion matrix:

```
              Predicted
              Positive  Negative
Actual
Positive        10        5
Negative         3        2
```

The values in the confusion matrix can be used to compute various performance metrics, such as precision, recall, and accuracy.



Calculate Confusion Matrix for a 2 classes problem



y	y pred	output for threshold 0.6	Recall	Precision	Accuracy
0	0.5	0	1/2	2/3	4/7
1	0.9	1			
0	0.7	1			
1	0.7	1			
1	0.3	0			
0	0.4	0			
1	0.5	0			

Individual Number	1	2	3	4	5	6	7	8	9	10	11	12
Actual Classification	1	1	1	1	1	1	1	1	0	0	0	0
Predicted Classification	0	0	1	1	1	1	1	1	1	0	0	0
Result	FN	FN	TP	TP	TP	TP	TP	TP	FP	TN	TN	TN

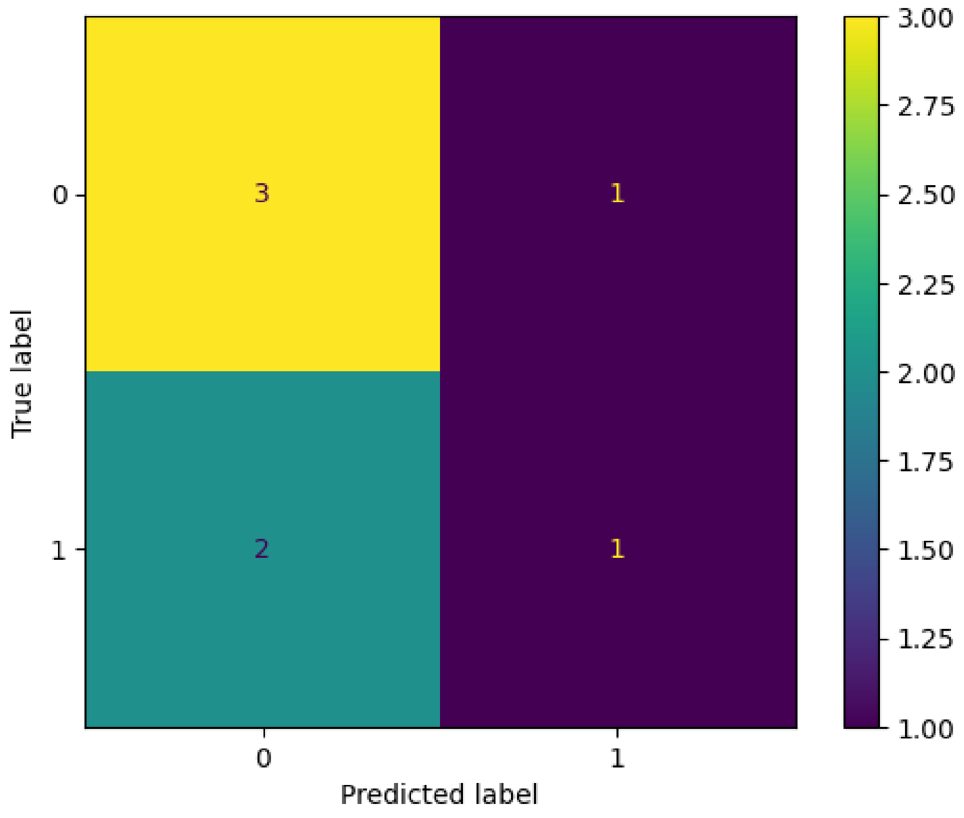
```python
import torch
from sklearn import metrics
import matplotlib.pyplot as plt
import torchmetrics
```

(continues on next page)

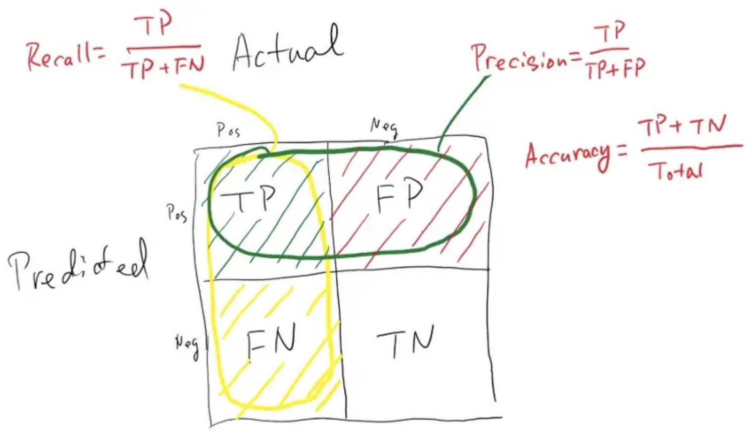
(continued from previous page)

```
# simulate a classification problem
y_true = torch.randint(0,2, (7,))
y_pred = torch.randint(0,2, (7,))

print(f"{y_true=}")
print(f"{y_pred=}")
print(f"confusion_matrix {metrics.confusion_matrix(y_true, y_pred)} ")

tn, fp, fn, tp = metrics.confusion_matrix(y_true, y_pred).ravel()
print(tn, fp, fn, tp)

disp = metrics.ConfusionMatrixDisplay(confusion_matrix=metrics.confusion_matrix(y_true,␣
↪y_pred) )
disp.plot()
plt.show()

print(f"{torchmetrics.functional.confusion_matrix(y_true, y_pred,task='binary')=}")
```

```
y_true=tensor([0, 0, 1, 1, 1, 0, 0])
y_pred=tensor([0, 0, 0, 0, 1, 1, 0])
confusion_matrix [[3 1]
 [2 1]]
3 1 2 1
```



```
torchmetrics.functional.confusion_matrix(y_true, y_pred,task='binary')=tensor([[3, 2],
        [1, 1]])
```



Precision

The above equation can be explained by saying, from all the classes we have predicted as positive, how many are actually positive. Precision should be high as possible.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Recall / Sensitivity / True Positive Rate

Sensitivity tells us what proportion of the positive class got correctly classified.

The above equation can be explained by saying, from all the positive classes, how many we predicted correctly. A simple example would be to determine what proportion of the actual sick people were correctly detected by the model.

Recall should be high as possible.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Note: Precision is about your prediction. Recall is about reality.

If your job is to identify thieves.

False Negative Rate

False Negative Rate (FNR) tells us what proportion of the positive class got incorrectly classified by the classifier.

A higher TPR and a lower FNR is desirable since we want to correctly classify the positive class.

$$FNR = \frac{\text{FN}}{\text{TP} + \text{FN}}$$

Specificity / True Negative Rate

Specificity tells us what proportion of the negative class got correctly classified.

Taking the same example as in Sensitivity, Specificity would mean determining the proportion of healthy people who were correctly identified by the model.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

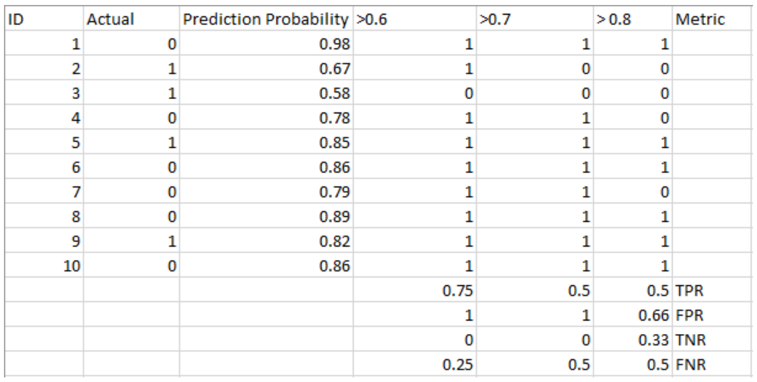
False Positive Rate

FPR tells us what proportion of the negative class got incorrectly classified by the classifier.

A higher TNR and a lower FPR is desirable since we want to correctly classify the negative class.

Out of these metrics, Sensitivity and Specificity are perhaps the most important and we will see later on how these are used to build an evaluation metric. But before that, let's understand why the probability of prediction is better than predicting the target class directly.

$$FPR = \frac{\text{FP}}{\text{FP} + \text{TN}}$$



ID	Actual	Prediction Probability	>0.6	>0.7	> 0.8	Metric
1	0	0.98	1	1	1	
2	1	0.67	1	0	0	
3	1	0.58	0	0	0	
4	0	0.78	1	1	0	
5	1	0.85	1	1	1	
6	0	0.86	1	1	1	
7	0	0.79	1	1	0	
8	0	0.89	1	1	1	
9	1	0.82	1	1	1	
10	0	0.86	1	1	1	
			0.75	0.5	0.5	TPR
			1	1	0.66	FPR
			0	0	0.33	TNR
			0.25	0.5	0.5	FNR

The metrics change with the changing threshold values. We can generate different confusion matrices and compare the various metrics that we discussed in the previous section. But that would not be a prudent thing to do. Instead, what we can do is generate a plot between some of these metrics so that we can easily visualize which threshold is giving us a better result.

The AUC-ROC curve solves just that problem!

```python
print(f"{metrics.precision_score(y_true, y_pred)=}")
print(f"{metrics.recall_score(y_true, y_pred)=}")
print(f"{metrics.accuracy_score(y_true, y_pred)=}")
print(f"{metrics.f1_score(y_true, y_pred)=}")
print(f"{metrics.fbeta_score(y_true, y_pred, beta=0.5)=}")
```

```
metrics.precision_score(y_true, y_pred)=0.5
metrics.recall_score(y_true, y_pred)=0.3333333333333333
metrics.accuracy_score(y_true, y_pred)=0.5714285714285714
metrics.f1_score(y_true, y_pred)=0.4
metrics.fbeta_score(y_true, y_pred, beta=0.5)=0.45454545454545453
```

F1-score

The F1-score is the harmonic mean of the precision and the recall. Using the harmonic mean has the effect that a good F1-score requires both a good precision and a good recall.

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

It is difficult to compare two models with low precision and high recall or vice versa. So to make them comparable, we use F-Score. F-score helps to measure Recall and Precision at the same time. It uses Harmonic Mean in place of Arithmetic Mean by punishing the extreme values more.

Drawbacks

One potential drawback of the F1 score as an evaluation metric is that it is sensitive to imbalanced class distributions. This means that if one class is much more prevalent in the data than the other, the F1 score may not be a reliable indicator of the classifier's performance.

For example, consider a binary classification problem where the positive class is rare, with only 1% of the samples belonging to that class. In this case, a classifier that simply predicts the negative class all the time would have an F1 score of 0, even though it is making the correct prediction 99% of the time. On the other hand, a classifier that makes a small number of correct predictions for the positive class (e.g., 5 out of 100) would have a relatively high F1 score, even though it is performing poorly overall.

ROC Curve and AUC

Best explaintion

- https://www.analyticsvidhya.com/blog/2020/06/auc-roc-curve-machine-learning/
- https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc
- https://stephenallwright.com/metric-choice/
- https://mlu-explain.github.io/

1.24.2 Ranking | Recommendation | Information Retrieval

None

1.24.3 Language Model

ROUGE

ROUGE is actually a set of metrics that are used to evaluate the quality of a text summarization system.

ROUGE-N Overlap of n-grams[2] between the system and reference summaries.

ROUGE-1 refers to the overlap of unigram (each word) between the system and reference summaries.

ROUGE-2 refers to the overlap of bigrams between the system and reference summaries.

1.25 Linear Algebra

1.25.1 Multiplying Matrices and Vectors

Multiplication of two A x B matrices a third matrix C.

$$C = AB$$

The product operation is defined by

$$C_{i,j} = \sum_{k=1} A_{i,k} B_{k,j}$$

The value at index i,j of result matrix C is given by dot product of ith row of Matrix A with jth column of Matrix

E.g

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$
$$B = \begin{bmatrix} 2 \\ 6 \end{bmatrix}$$
$$C_{1,1} = \sum_{k=1} A_{1,k} B_{k,1} = 1 * 2 + 2 * 6 = 14$$

1.25.2 Hadamard product & element-wise product

Element wise of multiplication to generate another matrix of same dimension.

$$C = A * B$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \circ \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}\, b_{11} & a_{12}\, b_{12} & a_{13}\, b_{13} \\ a_{21}\, b_{21} & a_{22}\, b_{22} & a_{23}\, b_{23} \\ a_{31}\, b_{31} & a_{32}\, b_{32} & a_{33}\, b_{33} \end{bmatrix}.$$

1.25.3 Dot product

The dot product for two vectors to generate scalar value.

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

1.25.4 Identity and Inverse Matrices

Identity Matrix

An identity matrix is a matrix that does not change any vector when we multiply that vector by that matrix.

$$I = \begin{bmatrix} 1 & 0 & 0 \\ \\ 0 & 1 & 0 \\ \\ 0 & 0 & 1 \end{bmatrix}$$

When 'apply' the identity matrix to a vector the result is this same vector:

$$I \cdot v = v$$
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \times x_1 + 0 \times x_2 + 0 \times x_3 \\ 0 \times x_1 + 1 \times x_2 + 0 \times x_3 \\ 0 \times x_1 + 0 \times x_2 + 1 \times x_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Inverse Matrix

The inverse of a matrix A is written as A^{-1}.

$$A^{-1}A = I_n$$

A matrix A is invertible if and only if there exists a matrix B such that $AB = BA = I$.

The inverse can be found using:

- Gaussian elimination
- LU decomposition
- Gauss-Jordan elimination

Singular Matrix

A square matrix that is not invertible is called singular or degenerate. A square matrix is singular if and only if its determinant is zero

1.25.5 Norm

Norm is function which measure the size of vector.

- Norms are non-negative values. If you think of the norms as a length, you easily see why it can't be negative.
- Norms are 0 if and only if the vector is a zero vector
- The triangle inequality** $u + v \leq u + v$

The norm is what is generally used to evaluate the error of a model.

Example

$$u = \begin{bmatrix} 1 & 6 \end{bmatrix}$$
$$v = \begin{bmatrix} 4 & 2 \end{bmatrix}$$
$$u + v = \sqrt{(1+4)^2 + (6+2)^2} = \sqrt{89} \approx 9.43$$
$$u + v = \sqrt{1^2 + 6^2} + \sqrt{4^2 + 2^2} = \sqrt{37} + \sqrt{20} \approx 10.55$$

The p-norm (also called ℓ_p) of vector x. Let p 1 be a real number.

$$\|x\|_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$
$$\|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p}$$

- L1 norm, Where p = 1 $\|x\|_1 = \sum_{i=1}^{n} |x_i|$
- L2 norm and euclidean norm, Where p = 2 $\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$
- L-max norm, Where p = infinity

$$u = \begin{bmatrix} 3 \\ \\ 4 \end{bmatrix}$$
$$u = \sqrt{|3|^2 + |4|^2} = \sqrt{25} = 5$$

Frobenius norm

Sometimes we may also wish to measure the size of a matrix. In the context of deep learning, the most common way to do this is with the Frobenius norm.

The Frobenius norm is the square root of the sum of the squares of all the elements of a matrix.

$$\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}^2}$$
$$\|A\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$$

The squared Euclidean norm

The squared L^2 norm is convenient because it removes the square root and we end up with the simple sum of every squared values of the vector.

The squared Euclidean norm is widely used in machine learning partly because it can be calculated with the vector operation $x^T x$. There can be performance gain due to the optimization

$$x = \begin{bmatrix} 2 \\ \\ 5 \\ \\ 3 \\ \\ 3 \end{bmatrix}$$
$$x^T = \begin{bmatrix} 2 & 5 & 3 & 3 \end{bmatrix}$$
$$x^T x = \begin{bmatrix} 2 & 5 & 3 & 3 \end{bmatrix} \times \begin{bmatrix} 2 \\ \\ 5 \\ \\ 3 \\ \\ 3 \end{bmatrix}$$

$$= 2 \times 2 + 5 \times 5 + 3 \times 3 + 3 \times 3 = 47$$

1.25.6 The Trace Operator

The sum of the elements along the main diagonal of a square matrix.

$$\mathrm{Tr}(A) = \sum_{i=1}^{n} a_{ii} = a_{11} + a_{22} + \cdots + a_{nn}$$

$$A = \begin{bmatrix} 2 & 9 & 8 \\ 4 & 7 & 1 \\ 8 & 2 & 5 \end{bmatrix}$$

$$\mathrm{Tr}(A) = 2 + 7 + 5 = 14$$

Satisfies the following properties:

$$\begin{aligned} \text{tr}(A) &= \text{tr}(A^T) \\ \text{tr}(A+B) &= \text{tr}(A) + \text{tr}(B) \\ \text{tr}(cA) &= c\text{tr}(A) \end{aligned}$$

1.25.7 Transpose

$$(A^T)_{ij} = A_{ji}$$

Satisfies the following properties:

$$(A+B)^T = A^T + B^T (AB)^T = B^T A^T (A^T)^{-1} = (A^{-1})^T$$

1.25.8 Diagonal matrix

A matrix where $A_{ij} = 0$ if $i \neq j$.

Can be written as diag(a) where a is a vector of values specifying the diagonal entries.

Diagonal matrices have the following properties:

$$\begin{aligned} \text{diag}(a) + \text{diag}(b) &= \text{diag}(a+b) \\ \text{diag}(a) \cdot \text{diag}(b) &= \text{diag}(a*b) \\ \text{diag}(a)^{-1} &= \text{diag}(a_1^{-1}, ..., a_n^{-1}) \\ \text{det}(\text{diag}(a)) &= \prod_i a_i \end{aligned}$$

Example

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & -3 \\ 0 & 0 & 0 \end{bmatrix} \mathit{or} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & -3 & 0 & 0 \end{bmatrix}$$

The eigenvalues of a diagonal matrix are the set of its values on the diagonal.

1.25.9 Symmetric matrix

A square matrix A where $A = A^T$.

$$\begin{bmatrix} 1 & 7 & 3 \\ 7 & 4 & 5 \\ 3 & 5 & 0 \end{bmatrix} = A^T = A$$

Some properties of symmetric matrices are:

- All the eigenvalues of the matrix are real.

1.25.10 Unit Vector

A unit vector has unit Euclidean norm.

$$\|x\|_2 := \sqrt{x_1^2 + \cdots + x_n^2} = 1$$
$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \sqrt{1^2 + 0^2 + 0^2} = 1$$

1.25.11 Orthogonal Matrix or Orthonormal Vectors

Orthogonal Vectors

Two vector x and y are orthogonal if they are perpendicular to each other or dot product is equal to zero.

$$x = \begin{bmatrix} 2 \\ \\ 2 \end{bmatrix}$$
$$y = \begin{bmatrix} 2 \\ \\ -2 \end{bmatrix}$$
$$x^T y = \begin{bmatrix} 2 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \times 2 + 2 \times -2 \end{bmatrix} = 0$$

Orthonormal Vectors

when the norm of orthogonal vectors is the unit norm they are called orthonormal.

Orthonormal Matrix

Orthogonal matrices are important because they have interesting properties. A matrix is orthogonal if columns are mutually orthogonal and have a unit norm (orthonormal) and rows are mutually orthonormal and have unit norm.

An orthogonal matrix is a square matrix whose columns and rows are orthonormal vectors.

$$A^{\mathrm{T}} A = A A^{\mathrm{T}} = I$$
$$A^{\mathrm{T}} = A^{-1}$$

where AT is the transpose of A and I is the identity matrix. This leads to the equivalent characterization: matrix A is orthogonal if its transpose is equal to its inverse.

so orthogonal matrices are of interest because their inverse is very cheap to compute.

Property 1

A orthogonal matrix has this property: $A^TA = I$.

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} A^T = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

$$A^TA = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} aa + cc & ab + cd \\ ab + cd & bb + dd \end{bmatrix}$$

$$= \begin{bmatrix} a^2 + c^2 & ab + cd \\ ab + cd & b^2 + d^2 \end{bmatrix}$$

$$A^TA = \begin{bmatrix} 1 & ab + cd \\ ab + cd & 1 \end{bmatrix}$$

$$\begin{bmatrix} a & c \end{bmatrix} \begin{bmatrix} b \\ d \end{bmatrix} = ab + cd$$

$$\begin{bmatrix} a & c \end{bmatrix} \begin{bmatrix} b \\ d \end{bmatrix} = 0$$

$$A^TA = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

that the norm of the vector $\begin{bmatrix} a & c \end{bmatrix}$ is equal to $a^2 + c^2$ (squared L^2). In addtion, we saw that the rows of A have a unit norm because A is orthogonal. This means that $a^2 + c^2 = 1$ and $b^2 + d^2 = 1$.

Property 2

We can show that if $A^TA = I$ then $A^T = A^{-1}$

$$\begin{aligned} (A^TA)A^{-1} &= IA^{-1} \\ (A^TA)A^{-1} &= A^{-1} \\ A^TAA^{-1} &= A^{-1} \\ A^TI &= A^{-1} \\ A^T &= A^{-1} \end{aligned}$$

You can refer to [this question](https://math.stackexchange.com/questions/1936020/why-is-the-inverse-of-an-orthogonal-matrix-equal-to-its-transpose).

Sine and cosine are convenient to create orthogonal matrices. Let's take the following matrix:

$$A = \begin{bmatrix} cos(50) & -sin(50) \\ sin(50) & cos(50) \end{bmatrix}$$

1.25.12 Eigendecomposition

The eigendecomposition is one form of matrix decomposition (only square matrices). Decomposing a matrix means that we want to find a product of matrices that is equal to the initial matrix. In the case of the eigendecomposition, we decompose the initial matrix into the product of its eigenvectors and eigenvalues.

$$Av = \lambda v$$

Eigenvectors and eigenvalues

Now imagine that the transformation of the initial vector gives us a new vector that has the exact same direction. The scale can be different but the direction is the same. Applying the matrix didn't change the direction of the vector. This special vector is called an eigenvector of the matrix. We will see that finding the eigenvectors of a matrix can be very useful. Imagine that the transformation of the initial vector by the matrix gives a new vector with the exact same direction. This vector is called an eigenvector of . This means that is a eigenvector of if and are in the same direction or to rephrase it if the vectors and are parallel. The output vector is just a scaled version of the input vector. This scalling factor is which is called the eigenvalue of .

$$A = \begin{bmatrix} 5 & 1 \\ 3 & 3 \end{bmatrix}$$
$$v = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$
$$Av = \lambda v$$
$$\begin{bmatrix} 5 & 1 \\ 3 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 6 \end{bmatrix}$$
$$6 \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 6 \end{bmatrix}$$

which means that v is well an eigenvector of A. Also, the corresponding eigenvalue is lambda=6.

Another eigenvector of is

$$v = \begin{bmatrix} 1 \\ -3 \end{bmatrix}$$
$$\begin{bmatrix} 5 & 1 \\ 3 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ -3 \end{bmatrix} = \begin{bmatrix} 2 \\ -6 \end{bmatrix}$$
$$2 \times \begin{bmatrix} 1 \\ -3 \end{bmatrix} = \begin{bmatrix} 2 \\ -6 \end{bmatrix}$$

which means that v is an eigenvector of A. Also, the corresponding eigenvalue is lambda=2.

Rescaled vectors if v is an eigenvector of A, then any rescaled vector sv is also an eigenvector of A. The eigenvalue of the rescaled vector is the same.

$$3v = \begin{bmatrix} 3 \\ -9 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 1 \\ 3 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ -9 \end{bmatrix} = \begin{bmatrix} 6 \\ -18 \end{bmatrix} = 2 \times \begin{bmatrix} 3 \\ -9 \end{bmatrix}$$

We have well A X 3v = lambda v and the eigenvalue is still lambda = 2 .

Concatenating eigenvalues and eigenvectors

Now that we have an idea of what eigenvectors and eigenvalues are we can see how it can be used to decompose a matrix. All eigenvectors of a matrix can be concatenated in a matrix with each column corresponding to each eigenvector.

$$v = \begin{bmatrix} 1 & 1 \\ 1 & -3 \end{bmatrix}$$

The first column [1 1] is the eigenvector of with lambda=6 and the second column [1 -3] with lambda=2.

The vector λ can be created from all eigenvalues:

$$\lambda = \begin{bmatrix} 6 \\ 2 \end{bmatrix}$$

Then the eigendecomposition is given by

$$A = V \cdot diag(\lambda) \cdot V^{-1}$$

Converting eigenvalues and eigenvectors to a matrix A.

$$V^{-1} = \begin{bmatrix} 0.75 & 0.25 \\ 0.25 & -0.25 \end{bmatrix}$$

$$V \cdot diag(\lambda) \cdot V^{-1}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & -3 \end{bmatrix} \begin{bmatrix} 6 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 0.75 & 0.25 \\ 0.25 & -0.25 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 \\ 1 & -3 \end{bmatrix} \begin{bmatrix} 6 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 6 & 2 \\ 6 & -6 \end{bmatrix}$$

$$\begin{bmatrix} 6 & 2 \\ 6 & -6 \end{bmatrix} \begin{bmatrix} 0.75 & 0.25 \\ 0.25 & -0.25 \end{bmatrix}$$

$$= \begin{bmatrix} 6 \times 0.75 + (2 \times 0.25) & 6 \times 0.25 + (2 \times -0.25) \\ 6 \times 0.75 + (-6 \times 0.25) & 6 \times 0.25 + (-6 \times -0.25) \end{bmatrix}$$

$$= \begin{bmatrix} 5 & 1 \\ 3 & 3 \end{bmatrix} = A$$

Real symmetric matrix

In the case of real symmetric matrices, the eigendecomposition can be expressed as

$$A = Q\Lambda Q^T$$

where Q is the matrix with eigenvectors as columns and Λ is $diag(\lambda)$.

$$A = \begin{bmatrix} 6 & 2 \\ 2 & 3 \end{bmatrix}$$

This matrix is symmetric because $A = A^T$. Its eigenvectors are:

$$Q = \begin{bmatrix} 0.89442719 & -0.4472136 \\ 0.4472136 & 0.89442719 \end{bmatrix}$$

$$\Lambda = \begin{bmatrix} 7 & 0 \\ 0 & 2 \end{bmatrix}$$

$$Q\Lambda = \begin{bmatrix} 0.89442719 & -0.4472136 \\ 0.4472136 & 0.89442719 \end{bmatrix} \begin{bmatrix} 7 & 0 \\ 0 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 0.89442719 \times 7 & -0.4472136 \times 2 \\ 0.4472136 \times 7 & 0.89442719 \times 2 \end{bmatrix}$$

$$= \begin{bmatrix} 6.26099033 & -0.8944272 \\ 3.1304952 & 1.78885438 \end{bmatrix}$$

$$Q^T = \begin{bmatrix} 0.89442719 & 0.4472136 \\ -0.4472136 & 0.89442719 \end{bmatrix}$$

$$Q\Lambda Q^T = \begin{bmatrix} 6.26099033 & -0.8944272 \\ 3.1304952 & 1.78885438 \end{bmatrix} \begin{bmatrix} 0.89442719 & 0.4472136 \\ -0.4472136 & 0.89442719 \end{bmatrix}$$

$$= \begin{bmatrix} 6 & 2 \\ 2 & 3 \end{bmatrix}$$

1.25.13 Singular Value Decomposition

The eigendecomposition can be done only for square matrices. The way to go to decompose other types of matrices that can't be decomposed with eigendecomposition is to use Singular Value Decomposition (SVD).

SVD decompose into 3 matrices.

$A = UDV^T$

U,D,V where U is a matrix with eigenvectors as columns and D is a diagonal matrix with eigenvalues on the diagonal and V is the transpose of U.

The matrices U,D,V have the following properties:

- U and V are orthogonal matrices U^T=U^{-1} and V^T=V^{-1}
- D is a diagonal matrix However D is not necessarily square.
- The columns of U are called the left-singular vectors of A while the columns of V are the right-singular vectors of A.The values along the diagonal of D are the singular values of A.

Intuition

I think that the intuition behind the singular value decomposition needs some explanations about the idea of matrix transformation. For that reason, here are several examples showing how the space can be transformed by 2D square matrices. Hopefully, this will lead to a better understanding of this statement: is a matrix that can be seen as a linear transformation. This transformation can be decomposed in three sub-transformations: 1. rotation, 2. re-scaling, 3. rotation. These three steps correspond to the three matrices , , and .

SVD and eigendecomposition

Now that we understand the kind of decomposition done with the SVD, we want to know how the sub-transformations are found. The matrices , and can be found by transforming in a square matrix and by computing the eigenvectors of this square matrix. The square matrix can be obtain by multiplying the matrix by its transpose in one way or the other:

 corresponds to the eigenvectors of ^T corresponds to the eigenvectors of ^T corresponds to the eigenvalues ^T or ^T which are the same.

1.25.14 The Moore-Penrose Pseudoinverse

We saw that not all matrices have an inverse because the inverse is used to solve system of equations. The Moore-Penrose pseudoinverse is a direct application of the SVD. the inverse of a matrix A can be used to solve the equation Ax=b.

$$A^{-1}Ax = A^{-1}bI_nx = A^{-1}bx = A^{-1}b$$

But in the case where the set of equations have 0 or many solutions the inverse cannot be found and the equation cannot be solved. The pseudoinverse is A^+ where A^+ is the pseudoinverse of A.

$$AA^+ \approx I_n$$
$$||AA^+ - I_n||$$

The following formula can be used to find the pseudoinverse:

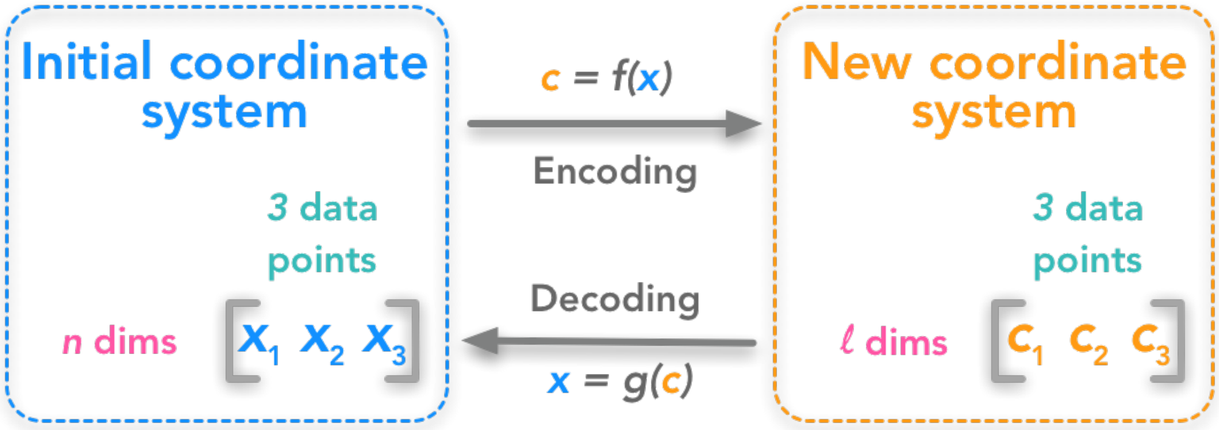
$$A^+ = VD^+U^T$$

1.25.15 Principal Components Analysis (PCA)

The aim of principal components analysis (PCA) is generaly to reduce the number of dimensions of a dataset where dimensions are not completely decorelated.

Describing the problem

The problem can be expressed as finding a function that converts a set of data points from to . This means that we change the number of dimensions of our dataset. We also need a function that can decode back from the transformed dataset to the initial one.



The first step is to understand the shape of the data. () is one data point containing dimensions. Let's have data points organized as column vectors (one column per point):

$x = \left[x^{(1)} x^{(2)} \cdots x^{(m)} \right]$

If we deploy the n dimensions of our data points we will have:

$$x = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} \\ \cdots & \cdots & \cdots & \cdots \\ x_n^{(1)} & x_n^{(2)} & \cdots & x_n^{(m)} \end{bmatrix}$$

We can also write:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix}$$

c will have the shape:

$$c = \begin{bmatrix} c_1 \\ c_2 \\ \cdots \\ c_l \end{bmatrix}$$

Adding some constraints: the decoding function

The encoding function f(x) transforms x into c and the decoding function transforms back c into an approximation of x. To keep things simple, PCA will respect some constraints:

Constraint 1

The decoding function has to be a simple matrix multiplication:

$$g(c)=Dc$$

By applying the matrix D to the dataset from the new coordinates system we should get back to the initial coordinate system.

Constraint 2

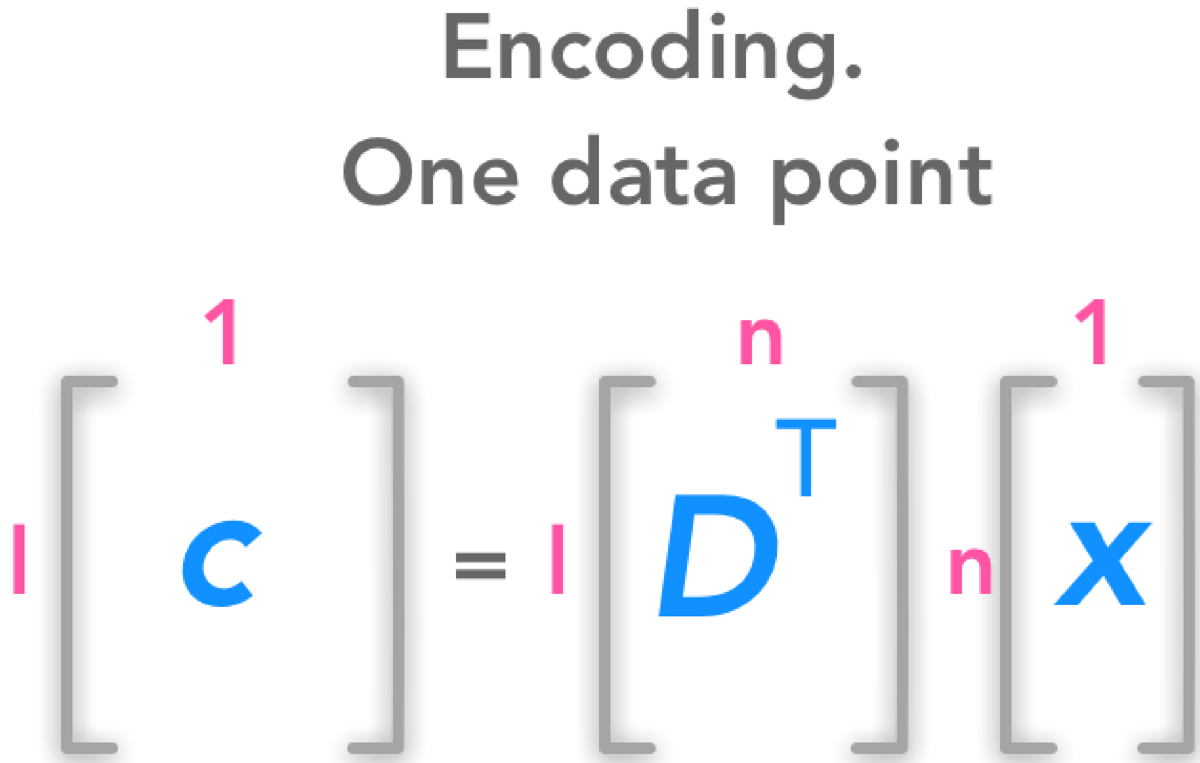
The columns of D must be orthogonal.

Constraint 3

The columns of D must have unit norm.

Finding the encoding function

For now we will consider only **one data point**. Thus we will have the following dimensions for these matrices (note that x and c are column vectors)



We want a decoding function which is a simple matrix multiplication. For that reason, we have g(c)=Dc.

We will then find the encoding function from the decoding function. We want to minimize the error between the decoded data point and the actual data point.

With our previous notation, this means reducing the distance between x and g(c). As an indicator of this distance, we will use the squared L^2 norm.

$$||x - g(c)||_2^2$$

This is what we want to minimize. Let's call c^* the optimal c. Mathematically it can be written:

$$ c^* = argmin ||x-g(c)||_2^2 $$

This means that we want to find the values of the vector c such that $||x - g(c)||_2^2$ is as small as possible.

the squared L^2 norm can be expressed as:

$$ ||y||_2^2 = y^Ty $$

We have named the variable y to avoid confusion with our x. Here $y = x - g(c)$ Thus the equation that we want to minimize becomes:

$$ (x - g(c))^T(x - g(c)) $$

Since the transpose respects addition we have:

$$ (x^T - g(c)^T)(x - g(c)) $$

By the distributive property we can develop:

$$ x^Tx - x^Tg(c) - g(c)^Tx + g(c)^Tg(c) $$

The commutative property tells us that $x^Ty = y^Tx$. We can use that in the previous equation: we have $g(c)^Tx = x^Tg(c)$. So the equation becomes:

$$ x^Tx -x^Tg(c) -x^Tg(c) + g(c)^Tg(c) = x^Tx -2x^Tg(c) + g(c)^Tg(c) $$

The first term x^Tx does not depends on c and since we want to minimize the function according to c we can just get off this term. We simplify to:

$$ c^* = arg min -2x^T g(c) + g(c)^T g(c) $$

Since $g(c) = Dc$:

$$ c^* = arg min -2x^T Dc + (Dc)^T Dc $$

With $(Dc)^T = c^TD^T$, we have:

$$ c^* = arg min -2x^T Dc + c^T D^T Dc $$

As we knew, $D^TD = I_l$ because D is orthogonal. and their columns have unit norm. We can replace in the equation:

$$ c^* = arg min -2x^T Dc + c^T I_l c $$

$$ c^* = arg min -2x^T Dc + c^T c $$

Minimizing the function

Now the goal is to find the minimum of the function 2T+T. One widely used way of doing that is to use the gradient descent algorithm. The main idea is that the sign of the derivative of the function at a specific value of tells you if you need to increase or decrease to reach the minimum. When the slope is near 0 , the minimum should have been reached.

Its mathematical notation is ().

Here we want to minimize through each dimension of c. We are looking for a slope of 0.

1.26 Statistics

1.26.1 Mean, Variance and Standard Deviation

Mean

The mean of a vector, usually denoted as μ , is the mean of its elements, that is to say the sum of the components divided by the number of components

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

Variance

The variance is the mean of the squared differences to the mean.

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

with $var(x)$ being the variance of the variable x, n the number of data samples, x_i the ith data sample and $\bar{x}$ the mean of x.

Standard Deviation

The *standard deviation* is simply the square root of the variance. It is usually denoted as σ:

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

We square root the variance to go back to the units of the observations.

Both the variance and the standard deviation are *dispersion indicators*: they tell you if the observations are clustered around the mean.

Note also that the variance and the standard deviation are always positive (it is like a distance, measuring how far away the data points are from the mean):

$$var(x) \geq 0$$
$$\sigma(x) \geq 0$$

1.26.2 Covariance and Correlation

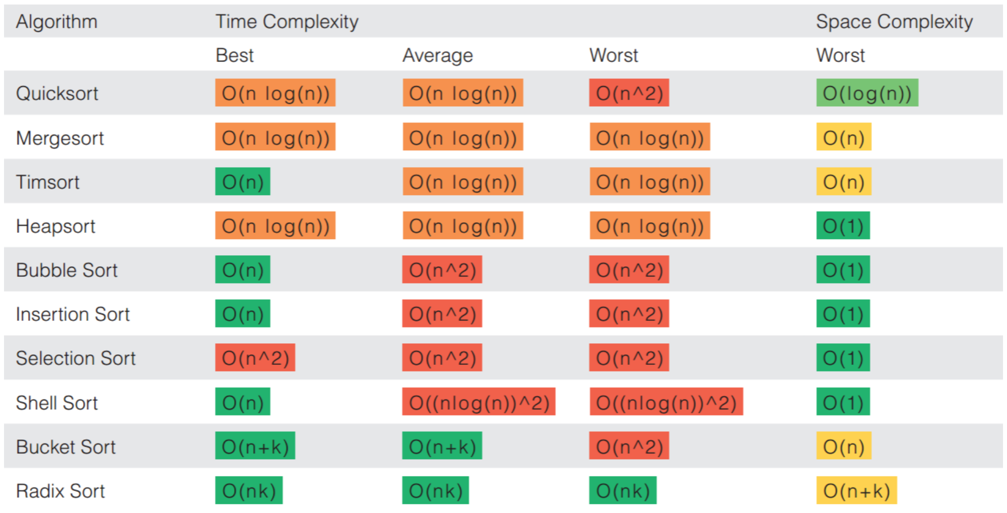
$$cov(x, y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

Correlation

The *correlation*, usually refering to the *Pearson's correlation coefficient*, is a normalized version of the covariance. It is scaled between -1 and 1

$$corr(x, y) = \frac{cov(x, y)}{\sigma_x \sigma_y}$$

1.27 Sorting Algorithms



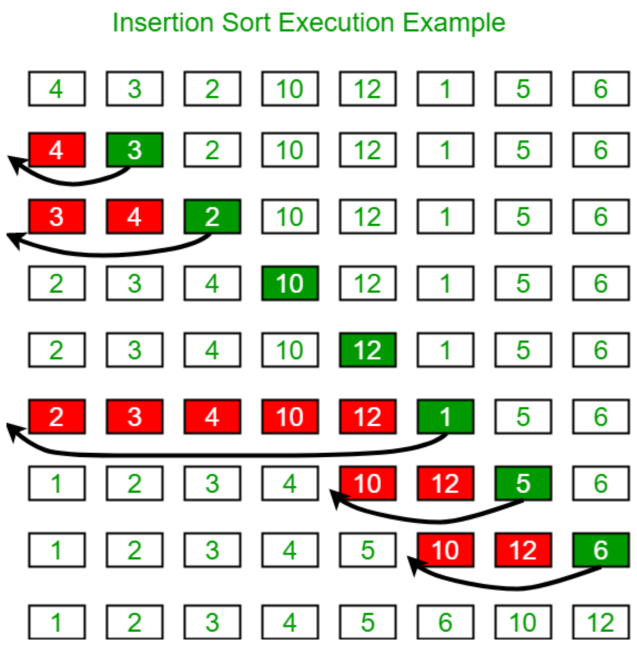
Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

1.27.1 Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Characteristics of Insertion Sort

- This algorithm is one of the simplest algorithm with simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.



https://www.swtestacademy.com/wp-content/uploads/2021/11/insertion-sort.gif

Implmentation

```
procedure insertionSort(A: list of sortable items)
   n = length(A)
   for i = 1 to n - 1 do
       j = i
       while j > 0 and A[j-1] > A[j] do
           swap(A[j], A[j-1])
           j = j - 1
       end while
   end for
end procedure
```

```
def insertion_sort(data: list):

    print(f"Unsorted List: {data}")

    for i in range(1, len(data)):
        print("===" * 10)
        print(f"Iteration: {i} & Current Element: {data[i]}")

        key = data[i]
        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1

        while j >= 0 and key < data[j] :
```

(continues on next page)

(continued from previous page)

```
                print(f"j = {j}")
                print(f"Swapping {data[j]} with {data[j + 1]}")
                data[j + 1] = data[j]
                j -= 1
                print(f"Swapped List: {data}")

        print(f"Inserting {key} at position {j + 1}")
        data[j + 1] = key
        print(f"New List: {data}")
        print("===" * 10)

    return data

insertion_sort([12, 11, 13, 5, 6])

print("####" * 10)

insertion_sort([6, 11, 6, 44, 6,7,9,22,0])

# This code is contributed by Mohit Kumra
```

```
Unsorted List: [12, 11, 13, 5, 6]
==============================
Iteration: 1 & Current Element: 11
j = 0
Swapping 12 with 11
Swapped List: [12, 12, 13, 5, 6]
Inserting 11 at position 0
New List: [11, 12, 13, 5, 6]
==============================
==============================
Iteration: 2 & Current Element: 13
Inserting 13 at position 2
New List: [11, 12, 13, 5, 6]
==============================
==============================
Iteration: 3 & Current Element: 5
j = 2
Swapping 13 with 5
Swapped List: [11, 12, 13, 13, 6]
j = 1
Swapping 12 with 13
Swapped List: [11, 12, 12, 13, 6]
j = 0
Swapping 11 with 12
Swapped List: [11, 11, 12, 13, 6]
Inserting 5 at position 0
New List: [5, 11, 12, 13, 6]
==============================
==============================
Iteration: 4 & Current Element: 6
j = 3
```

(continues on next page)

(continued from previous page)

```
Swapping 13 with 6
Swapped List: [5, 11, 12, 13, 13]
j = 2
Swapping 12 with 13
Swapped List: [5, 11, 12, 12, 13]
j = 1
Swapping 11 with 12
Swapped List: [5, 11, 11, 12, 13]
Inserting 6 at position 1
New List: [5, 6, 11, 12, 13]
==============================
########################################
Unsorted List: [6, 11, 6, 44, 6, 7, 9, 22, 0]
==============================
Iteration: 1 & Current Element: 11
Inserting 11 at position 1
New List: [6, 11, 6, 44, 6, 7, 9, 22, 0]
==============================
==============================
Iteration: 2 & Current Element: 6
j = 1
Swapping 11 with 6
Swapped List: [6, 11, 11, 44, 6, 7, 9, 22, 0]
Inserting 6 at position 1
New List: [6, 6, 11, 44, 6, 7, 9, 22, 0]
==============================
==============================
Iteration: 3 & Current Element: 44
Inserting 44 at position 3
New List: [6, 6, 11, 44, 6, 7, 9, 22, 0]
==============================
==============================
Iteration: 4 & Current Element: 6
j = 3
Swapping 44 with 6
Swapped List: [6, 6, 11, 44, 44, 7, 9, 22, 0]
j = 2
Swapping 11 with 44
Swapped List: [6, 6, 11, 11, 44, 7, 9, 22, 0]
Inserting 6 at position 2
New List: [6, 6, 6, 11, 44, 7, 9, 22, 0]
==============================
==============================
Iteration: 5 & Current Element: 7
j = 4
Swapping 44 with 7
Swapped List: [6, 6, 6, 11, 44, 44, 9, 22, 0]
j = 3
Swapping 11 with 44
Swapped List: [6, 6, 6, 11, 11, 44, 9, 22, 0]
Inserting 7 at position 3
New List: [6, 6, 6, 7, 11, 44, 9, 22, 0]
```

(continues on next page)

(continued from previous page)

```
==============================
==============================
Iteration: 6 & Current Element: 9
j = 5
Swapping 44 with 9
Swapped List: [6, 6, 6, 7, 11, 44, 44, 22, 0]
j = 4
Swapping 11 with 44
Swapped List: [6, 6, 6, 7, 11, 11, 44, 22, 0]
Inserting 9 at position 4
New List: [6, 6, 6, 7, 9, 11, 44, 22, 0]
==============================
==============================
Iteration: 7 & Current Element: 22
j = 6
Swapping 44 with 22
Swapped List: [6, 6, 6, 7, 9, 11, 44, 44, 0]
Inserting 22 at position 6
New List: [6, 6, 6, 7, 9, 11, 22, 44, 0]
==============================
==============================
Iteration: 8 & Current Element: 0
j = 7
Swapping 44 with 0
Swapped List: [6, 6, 6, 7, 9, 11, 22, 44, 44]
j = 6
Swapping 22 with 44
Swapped List: [6, 6, 6, 7, 9, 11, 22, 22, 44]
j = 5
Swapping 11 with 22
Swapped List: [6, 6, 6, 7, 9, 11, 11, 22, 44]
j = 4
Swapping 9 with 11
Swapped List: [6, 6, 6, 7, 9, 9, 11, 22, 44]
j = 3
Swapping 7 with 9
Swapped List: [6, 6, 6, 7, 7, 9, 11, 22, 44]
j = 2
Swapping 6 with 7
Swapped List: [6, 6, 6, 6, 7, 9, 11, 22, 44]
j = 1
Swapping 6 with 6
Swapped List: [6, 6, 6, 6, 7, 9, 11, 22, 44]
j = 0
Swapping 6 with 6
Swapped List: [6, 6, 6, 6, 7, 9, 11, 22, 44]
Inserting 0 at position 0
New List: [0, 6, 6, 6, 7, 9, 11, 22, 44]
==============================
```

```
[0, 6, 6, 6, 7, 9, 11, 22, 44]
```

1.27.2 Merge sort

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

The “Merge Sort” uses a recursive algorithm to achieve its results.

Advantages of the Merge Sort

- Merge sort can efficiently sort a list in O(n*log(n)) time.
- Merge sort can be used with linked lists without taking up any more space.
- A merge sort algorithm is used to count the number of inversions in the list.
- Merge sort is employed in external sorting.

Drawbacks of the Merge Sort

- For small datasets, merge sort is slower than other sorting algorithms.
- For the temporary array, mergesort requires an additional space of O(n).
- Even if the array is sorted, the merge sort goes through the entire process.

Python implementation of MergeSort

```python
def mergeSort(arr):
        if len(arr) > 1:

                # Finding the mid of the array
                mid = len(arr)//2

                # Dividing the array elements
                L = arr[:mid]

                # into 2 halves
                R = arr[mid:]

                # Sorting the first half
                mergeSort(L)

                # Sorting the second half
                mergeSort(R)

                i = j = k = 0

                # Copy data to temp arrays L[] and R[]
                while i < len(L) and j < len(R):
                        if L[i] <= R[j]:
                                arr[k] = L[i]
```

(continues on next page)

(continued from previous page)

```
                                i += 1
                        else:
                                arr[k] = R[j]
                                j += 1
                        k += 1

                # Checking if any element was left
                while i < len(L):
                        arr[k] = L[i]
                        i += 1
                        k += 1

                while j < len(R):
                        arr[k] = R[j]
                        j += 1
                        k += 1

arr = [12, 11, 13, 5, 6, 7]
print(arr)
mergeSort(arr)
print(arr)
```

```
[12, 11, 13, 5, 6, 7]
[5, 6, 7, 11, 12, 13]
```

1.27.3 Binary Search

Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

https://blog.penjee.com/wp-content/uploads/2015/04/binary-and-linear-search-animations.gif

python Implmentation of Binary Search

```
def binarySearchHelper(lst, elt, left, right):
    n = len(lst)
    if (left > right):
        return None # Search region is empty -- let us bail since we cannot find the␣
↪element elt in the list.
    else:
        # If elt exists in the list, it must be between left and right indices.
        mid = (left + right)//2 # Note that // is integer division
        if lst[mid] == elt:
            return mid # BINGO -- we found it. Return its index signalling that we found␣
↪it.
        elif lst[mid] < elt:
            # We search in the right part of the list
            return binarySearchHelper(lst, elt, mid+1, right)
        else: # lst[mid] > elt
            # We search in the left part of the list.
```

(continues on next page)

(continued from previous page)

```
            return binarySearchHelper(lst, elt, left, mid-1)

def binarySearch(lst, elt):
    n = len(lst)
    if (elt < lst[0] or elt > lst[n-1]):
        return None
    else: # Note: we will only get here if
          # lst[0] <= elt <= lst[n-1]
        return binarySearchHelper(lst, elt, 0, n-1)

print("Searching for 9 in list [0,2,3,4,6,9,12]")
print(binarySearch([0,2,3,4,6,9,12], 9))

print("Searching for 8 in list [1, 3, 4, 6, 8, 9,10, 11, 12, 15]")
print(binarySearch([1, 3, 4, 6, 8, 9,10, 11, 12, 15], 8))

print("Searching for 5 in list [1, 3, 4, 6, 8, 9,10, 11, 12, 15]")
print(binarySearch([1, 3, 4, 6, 8, 9,10, 11, 12, 15], 5))

print("Searching for 0 in list [0,2]")
print(binarySearch([0,2], 0))

print("Searching for 1 in list [0,2]")
print(binarySearch([0,2], 1))

print("Searching for 2 in list [0,2]")
print(binarySearch([0,2], 2))

print("Searching for 1 in list [1]")
print(binarySearch([1], 1))

print("Searching for 2 in list [1]")
print(binarySearch([1], 2))
```

```
Searching for 9 in list [0,2,3,4,6,9,12]
5
Searching for 8 in list [1, 3, 4, 6, 8, 9,10, 11, 12, 15]
4
Searching for 5 in list [1, 3, 4, 6, 8, 9,10, 11, 12, 15]
None
Searching for 0 in list [0,2]
0
Searching for 1 in list [0,2]
None
Searching for 2 in list [0,2]
1
Searching for 1 in list [1]
0
Searching for 2 in list [1]
None
```

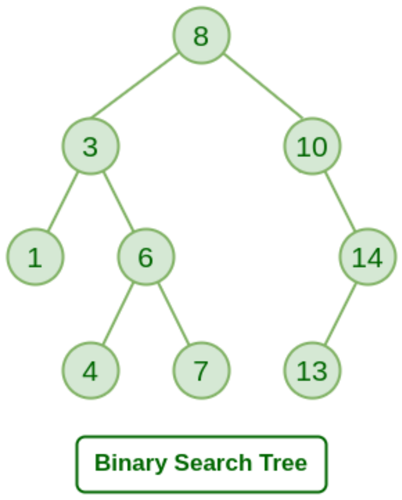
1.28 Graphs Data Structure

1.28.1 Binary Search Tree

A Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree.

In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

Left node > Parent node > Right node



Advantages of Binary search tree

Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.

As compared to array and linked lists, insertion and deletion operations are faster in BST.

```
class Node:
    # Implement a node of the binary search tree.
    # Constructor for a node with key and a given parent
    # parent can be None for a root node.
    def __init__(self, key, parent = None):
        self.key = key
        self.parent = parent
        self.left = None # We will set left and right child to None
        self.right = None
        # Make sure that the parent's left/right pointer
        # will point to the newly created node.
        if parent != None:
            if key < parent.key:
                assert(parent.left == None), 'parent already has a left child -- unable␣
↪to create node'
```

(continues on next page)

(continued from previous page)

```
                parent.left = self
            else:
                assert key > parent.key, 'key is same as parent.key. We do not allow␣
↪duplicate keys in a BST since it breaks some of the algorithms.'
                assert(parent.right == None ), 'parent already has a right child --␣
↪unable to create node'
                parent.right = self

    # Utility function that keeps traversing left until it finds
    # the leftmost descendant
    def get_leftmost_descendant(self):
        if self.left != None:
            return self.left.get_leftmost_descendant()
        else:
            return self


    # You can call search recursively on left or right child
    # as appropriate.
    # If search succeeds: return a tuple True and the node in the tree
    # with the key we are searching for.
    # Also note that if the search fails to find the key
    # you should return a tuple False and the node which would
    # be the parent if we were to insert the key subsequently.
    def search(self, key):
        if self.key == key:
            return (True, self)
        # your code here
        if self.key < key and self.right != None:
            return self.right.search(key)

        if self.key > key and self.left != None:
            return self.left.search(key)

        return (False, self)



    # To insert first search for it and find out
    # the parent whose child the currently inserted key will be.
    # Create a new node with that key and insert.
    # return None if key already exists in the tree.
    # return the new node corresponding to the inserted key otherwise.
    def insert(self, key):
        # your code here
        (b, found_node) = self.search(key)
        if b is not False:
            return None
        else:
            return Node(key, found_node)
```

(continues on next page)

(continued from previous page)

```
    # height of a node whose children are both None is defined
    # to be 1.
    # height of any other node is 1 + maximum of the height
    # of its children.
    # Return a number that is th eheight.
    def height(self):
        # your code here
        if self.left is None and self.right is None:
            return 1
        elif self.left is None:
            return 1 + self.right.height()
        elif self.right is None:
            return 1 + self.left.height()
        else:
            return 1 + max(self.left.height(), self.right.height())


    # programming.
    # Case 1: both children of the node are None
    #   -- in this case, deletion is easy: simply find out if the node with key is its
    #      parent's left/right child and set the corr. child to None in the parent node.
    # Case 2: one of the child is None and the other is not.
    #   -- replace the node with its only child. In other words,
    #      modify the parent of the child to be the to be deleted node's parent.
    #      also change the parent's left/right child appropriately.
    # Case 3: both children of the parent are not None.
    #    -- first find its successor (go one step right and all the way to the left).
    #    -- function get_leftmost_descendant may be helpful here.
    #    -- replace the key of the node by its successor.
    #    -- delete the successor node.
    # return: no return value specified

    def delete(self, key):
        (found, node_to_delete) = self.search(key)
        assert(found == True), f"key to be deleted:{key}- does not exist in the tree"
        # your code here
        if node_to_delete.left is None and node_to_delete.right is None:
            if node_to_delete.parent.left == node_to_delete:
                node_to_delete.parent.left = None
            else:
                node_to_delete.parent.right = None
        elif node_to_delete.left is None:
            if node_to_delete.parent.left == node_to_delete:
                node_to_delete.parent.left = node_to_delete.right
            else:
                node_to_delete.parent.right = node_to_delete.right
        elif node_to_delete.right is None:
            if node_to_delete.parent.left == node_to_delete:
                node_to_delete.parent.left = node_to_delete.left
            else:
                node_to_delete.parent.right = node_to_delete.left
        else:
```

(continues on next page)

(continued from previous page)

```
            successor = node_to_delete.right.get_leftmost_descendant()
            node_to_delete.key = successor.key
            successor.delete(successor.key)

t1 = Node(25, None)
t2 = Node(12, t1)
t3 = Node(18, t2)
t4 = Node(40, t1)

print('-- Testing basic node construction (originally provided code) -- ')
assert(t1.left == t2), 'test 1 failed'
assert(t2.parent == t1),  'test 2 failed'
assert(t2.right == t3), 'test 3 failed'
assert (t3.parent == t2), 'test 4 failed'
assert(t1.right == t4), 'test 5 failed'
assert(t4.left == None), 'test 6 failed'
assert(t4.right == None), 'test 7 failed'
# The tree should be :
#             25
#             /\
#         12     40
#         /\
#     None   18
#

print('-- Testing search -- ')
(b, found_node) = t1.search(18)
assert b and found_node.key == 18, 'test 8 failed'
(b, found_node) = t1.search(25)
assert b and found_node.key == 25, 'test 9 failed -- you should find the node with key␣
↪25 which is the root'
(b, found_node) = t1.search(26)
assert(not b), 'test 10 failed'
assert(found_node.key == 40), 'test 11 failed -- you should be returning the leaf node␣
↪which would be the parent to the node you failed to find if it were to be inserted in␣
↪the tree.'

print('-- Testing insert -- ')
ins_node = t1.insert(26)
assert ins_node.key == 26, ' test 12 failed '
assert ins_node.parent == t4,  ' test 13 failed '
assert t4.left == ins_node,  ' test 14 failed '

ins_node2 = t1.insert(33)
assert ins_node2.key == 33, 'test 15 failed'
assert ins_node2.parent == ins_node, 'test 16 failed'
assert ins_node.right == ins_node2, 'test 17 failed'

print('-- Testing height -- ')

assert t1.height() == 4, 'test 18 failed'
assert t4.height() == 3, 'test 19 failed'
```

(continues on next page)

(continued from previous page)

```
assert t2.height() == 2, 'test 20 failed'


# Testing deletion
t1 = Node(16, None)
# insert the nodes in the list
lst = [18,25,10, 14, 8, 22, 17, 12]
for elt in lst:
    t1.insert(elt)

# The tree should look like this
#               16
#            /     \
#          10      18
#        /  \     /  \
#       8   14   17  25
#          /         /
#         12        22


# Let us test the three deletion cases.
# case 1 let's delete node 8
# node 8 does not have left or right children.
t1.delete(8) # should have both children nil.
(b8,n8) = t1.search(8)
assert not b8, 'Test A: deletion fails to delete node.'
(b,n) = t1.search(10)
assert( b) , 'Test B failed: search does not work'
assert n.left == None, 'Test C failed: Node 8 was not properly deleted.'

# Let us test deleting the node 14 whose right child is none.
# n is still pointing to the node 10 after deleting 8.
# let us ensure that it's right child is 14
assert n.right != None, 'Test D failed: node 10 should have right child 14'
assert n.right.key == 14, 'Test E failed: node 10 should have right child 14'

# Let's delete node 14
t1.delete(14)
(b14, n14) = t1.search(14)
assert not b14, 'Test F: Deletion of node 14 failed -- it still exists in the tree.'
(b,n) = t1.search(10)
assert n.right != None , 'Test G failed: deletion of node 14 not handled correctly'
assert n.right.key == 12, f'Test H failed: deletion of node 14 not handled correctly: {n.
↪right.key}'

# Let's delete node 18 in the tree.
# It should be replaced by 22.

t1.delete(18)
(b18, n18) = t1.search(18)
assert not b18, 'Test I: Deletion of node 18 failed'
assert t1.right.key == 22 , ' Test J: Replacement of node with successor failed.'
```

(continues on next page)

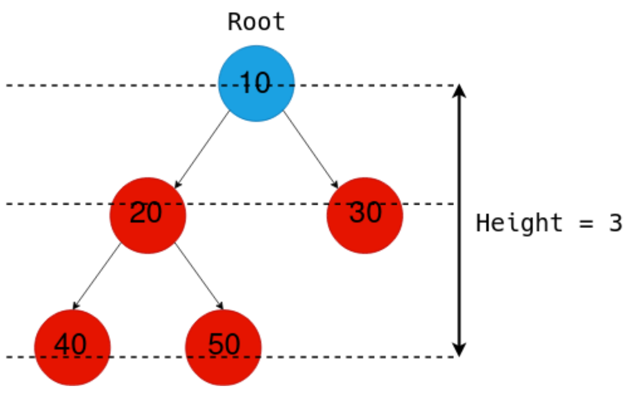
(continued from previous page)

```
assert t1.right.right.left == None, ' Test K: replacement of node with successor failed -
↪- you did not delete the successor leaf properly?'
```

```
-- Testing basic node construction (originally provided code) --
-- Testing search --
-- Testing insert --
-- Testing height --
```

Height of BST

The height of a Binary Tree is defined as the maximum depth of any leaf node from the root node. That is, it is the length of the longest path from the root node to any leaf node.



Find in BST

Complexity: O(log n) and O(n) in worst case

Insertion and Deleteion in BST

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if root.val == key:
            return root
        elif root.val < key:
            root.right = insert(root.right, key)
        else:
            root.left = insert(root.left, key)
    return root
```

(continues on next page)

(continued from previous page)

```
def inorder(root):
    if root:
        inorder(root.left)
        print(root.val, end =" ")
        inorder(root.right)


if __name__ == '__main__':

    # Let us create the following BST
    # 50
    #  /     \
    # 30     70
    #  / \ / \
    # 20 40 60 80

    r = Node(50)
    r = insert(r, 30)
    r = insert(r, 20)
    r = insert(r, 40)
    r = insert(r, 70)
    r = insert(r, 60)
    r = insert(r, 80)

    # Print inorder traversal of the BST
    inorder(r)
```

```
20 30 40 50 60 70 80
```

Delete a node from BST

```
# Python program to demonstrate delete operation
# in binary search tree

# A Binary Tree Node


class Node:

        # Constructor to create a new node
        def __init__(self, key):
                self.key = key
                self.left = None
                self.right = None


# A utility function to do inorder traversal of BST
def inorder(root):
        if root is not None:
                inorder(root.left)
                print(root.key, end=" ")
```

(continues on next page)

(continued from previous page)

```
                inorder(root.right)


# A utility function to insert a
# new node with given key in BST
def insert(node, key):

        # If the tree is empty, return a new node
        if node is None:
                return Node(key)

        # Otherwise recur down the tree
        if key < node.key:
                node.left = insert(node.left, key)
        else:
                node.right = insert(node.right, key)

        # return the (unchanged) node pointer
        return node

# Given a non-empty binary
# search tree, return the node
# with minimum key value
# found in that tree. Note that the
# entire tree does not need to be searched


def minValueNode(node):
        current = node

        # loop down to find the leftmost leaf
        while(current.left is not None):
                current = current.left

        return current

# Given a binary search tree and a key, this function
# delete the key and returns the new root


def deleteNode(root, key):

        # Base Case
        if root is None:
                return root

        # If the key to be deleted
        # is smaller than the root's
        # key then it lies in left subtree
        if key < root.key:
                root.left = deleteNode(root.left, key)
```

(continues on next page)

(continued from previous page)

```
        # If the kye to be delete
        # is greater than the root's key
        # then it lies in right subtree
        elif(key > root.key):
                root.right = deleteNode(root.right, key)

        # If key is same as root's key, then this is the node
        # to be deleted
        else:

                # Node with only one child or no child
                if root.left is None:
                        temp = root.right
                        root = None
                        return temp

                elif root.right is None:
                        temp = root.left
                        root = None
                        return temp

                # Node with two children:
                # Get the inorder successor
                # (smallest in the right subtree)
                temp = minValueNode(root.right)

                # Copy the inorder successor's
                # content to this node
                root.key = temp.key

                # Delete the inorder successor
                root.right = deleteNode(root.right, temp.key)

        return root


# Driver code
""" Let us create following BST
                        50
                /         \
                30         70
                / \ / \
        20 40 60 80 """

root = None
root = insert(root, 50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)
```

(continues on next page)

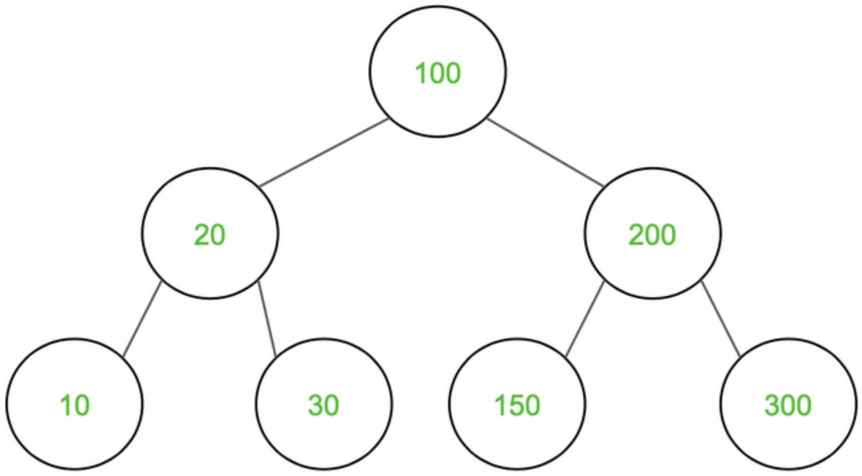
(continued from previous page)

```
print("Inorder traversal of the given tree")
inorder(root)

print("\nDelete 20")
root = deleteNode(root, 20)
print("Inorder traversal of the modified tree")
inorder(root)

print("\nDelete 30")
root = deleteNode(root, 30)
print("Inorder traversal of the modified tree")
inorder(root)

print("\nDelete 50")
root = deleteNode(root, 50)
print("Inorder traversal of the modified tree")
inorder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

```
Inorder traversal of the given tree
20 30 40 50 60 70 80
Delete 20
Inorder traversal of the modified tree
30 40 50 60 70 80
Delete 30
Inorder traversal of the modified tree
40 50 60 70 80
Delete 50
Inorder traversal of the modified tree
40 60 70 80
```

Traversals – Inorder, Preorder, Post Order

Given a Binary Search Tree, The task is to print the elements in inorder, preorder, and postorder traversal of the Binary Search Tree.



Binary Search Tree

Inorder Traversal: 10 20 30 100 150 200 300

Preorder Traversal: 100 20 10 30 200 150 300

Postorder Traversal: 10 30 20 150 300 200 100

Inorder Traversal:

Traverse left subtree Visit the root and print the data. Traverse the right subtree

```
class Node:
        def __init__(self, v):
                self.left = None
                self.right = None
                self.data = v

# Inorder Traversal
def printInorder(root):
        if root:
                # Traverse left subtree
                printInorder(root.left)

                # Visit node
                print(root.data,end=" ")

                # Traverse right subtree
                printInorder(root.right)

# Driver code
if __name__ == "__main__":
        # Build the tree
        root = Node(100)
        root.left = Node(20)
        root.right = Node(200)
        root.left.left = Node(10)
```

(continues on next page)

(continued from previous page)

```
        root.left.right = Node(30)
        root.right.left = Node(150)
        root.right.right = Node(300)

        # Function call
        print("Inorder Traversal:",end=" ")
        printInorder(root)

        # This code is contributed by ajaymakvana.
```

```
Inorder Traversal: 10 20 30 100 150 200 300
```

Preorder Traversal

At first visit the root then traverse left subtree and then traverse the right subtree.

Follow the below steps to implement the idea:

- Visit the root and print the data.
- Traverse left subtree
- Traverse the right subtree

```
class Node:
        def __init__(self, v):
                self.data = v
                self.left = None
                self.right = None

# Preorder Traversal
def printPreOrder(node):
        if node is None:
                return
        # Visit Node
        print(node.data, end = " ")

        # Traverse left subtree
        printPreOrder(node.left)

        # Traverse right subtree
        printPreOrder(node.right)

# Driver code
if __name__ == "__main__":
        # Build the tree
        root = Node(100)
        root.left = Node(20)
        root.right = Node(200)
        root.left.left = Node(10)
        root.left.right = Node(30)
        root.right.left = Node(150)
```

(continues on next page)

(continued from previous page)

```
        root.right.right = Node(300)

        # Function call
        print("Preorder Traversal: ", end = "")
        printPreOrder(root)
```

```
Preorder Traversal: 100 20 10 30 200 150 300
```

Postorder Traversal

At first traverse left subtree then traverse the right subtree and then visit the root.

Follow the below steps to implement the idea:

- Traverse left subtree
- Traverse the right subtree
- Visit the root and print the data.

```
class Node:
        def __init__(self, v):
                self.data = v
                self.left = None
                self.right = None

# Preorder Traversal
def printPostOrder(node):
        if node is None:
                return

        # Traverse left subtree
        printPostOrder(node.left)

        # Traverse right subtree
        printPostOrder(node.right)

        # Visit Node
        print(node.data, end = " ")

# Driver code
if __name__ == "__main__":
        # Build the tree
        root = Node(100)
        root.left = Node(20)
        root.right = Node(200)
        root.left.left = Node(10)
        root.left.right = Node(30)
        root.right.left = Node(150)
        root.right.right = Node(300)

        # Function call
```

(continues on next page)

(continued from previous page)

```
        print("Postorder Traversal: ", end = "")
        printPostOrder(root)
```

```
Postorder Traversal: 10 30 20 150 300 200 100
```

1.28.2 Red-Black Tree

When it comes to searching and sorting data, one of the most fundamental data structures is the binary search tree. However, the performance of a binary search tree is highly dependent on its shape, and in the worst case, it can degenerate into a linear structure with a time complexity of O(n). This is where Red Black Trees come in, they are a type of balanced binary search tree that use a specific set of rules to ensure that the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Red Black Trees are self-balancing, meaning that the tree adjusts itself automatically after each insertion or deletion operation. It uses a simple but powerful mechanism to maintain balance, by coloring each node in the tree either red or black.

Properties of Red Black Tree

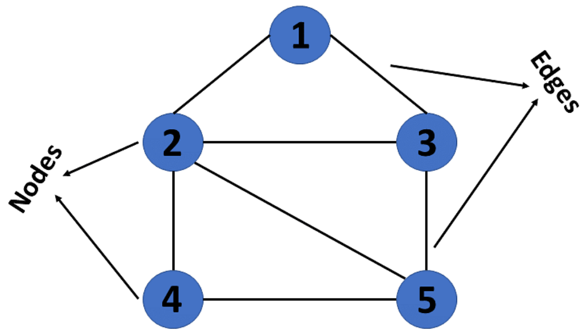
The Red-Black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties –

1. Root property: The root is black.
2. External property: Every leaf (Leaf is a NULL child of a node) is black in Red-Black tree.
3. Internal property: The children of a red node are black. Hence possible parent of red node is a black node.
4. Depth property: All the leaves have the same black depth.
5. Path property: Every simple path from root to descendant leaf node contains same number of black nodes.

The result of all these above-mentioned properties is that the Red-Black tree is roughly balanced.

1.28.3 Graph Data Structure

A graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as, a Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.



In graph theory, a graph is a mathematical structure consisting of a set of objects, called vertices or nodes, and a set of connections, called edges, which link pairs of vertices. The notation:

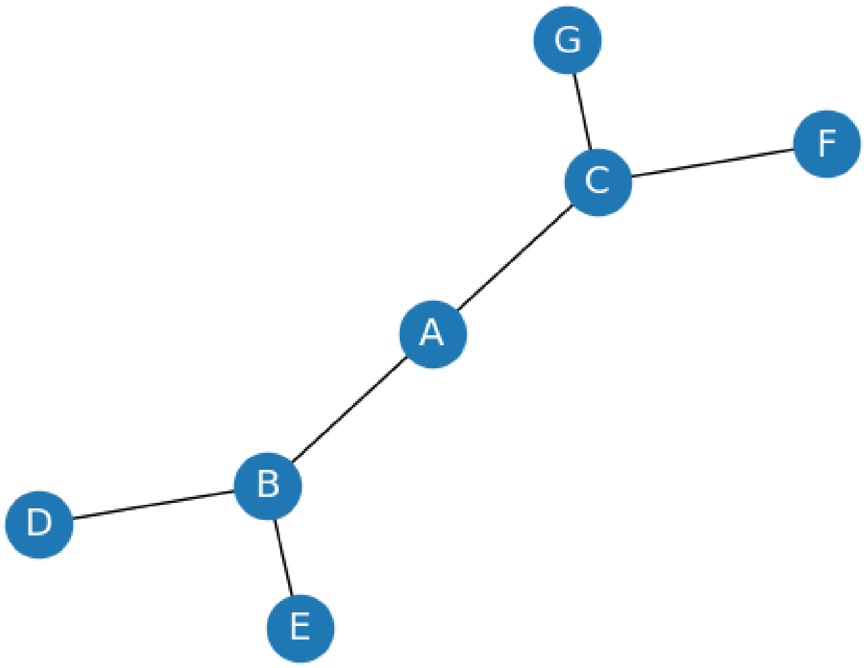
$$G = (V, E)$$

is used to represent a graph, where G is the graph, V is the set of vertices, and $\bigvee$ is the set of edges.

The nodes of a graph can represent any objects, such as cities, people, web pages, or molecules, and the edges represent the relationships or connections between them.

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()
G.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G
↪')])

plt.axis('off')
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=0),
                 node_size=600,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white'
                 )
```

```
/home/docs/checkouts/readthedocs.org/user_builds/ml-math/envs/latest/lib/python3.11/site-
↪packages/networkx/drawing/nx_pylab.py:450: UserWarning: No data for colormapping␣
↪provided via 'c'. Parameters 'cmap' will be ignored
  node_collection = ax.scatter(
```



Terminology

The following are the most commonly used terms in graph theory with respect to graphs:

1. Vertex - A vertex, also called a "node", is a fundamental part of a graph. In the context of graphs, a vertex is an object which may contain zero or more items called attributes.
2. Edge - An edge is a connection between two vertices. An edge may contain a weight/value/cost.
3. Path - A path is a sequence of edges connecting a sequence of vertices.
4. Cycle - A cycle is a path of edges that starts and ends on the same vertex.
5. Weighted Graph/Network - A weighted graph is a graph with numbers assigned to its edges. These numbers are called weights.
6. Unweighted Graph/Network - An unweighted graph is a graph in which all edges have equal weight.
7. Directed Graph/Network - A directed graph is a graph where all the edges are directed.
8. Undirected Graph/Network - An undirected graph is a graph where all the edges are not directed.
9. Adjacent Vertices - Two vertices in a graph are said to be adjacent if there is an edge connecting them.

Types of Graphs

There are two types of graphs:

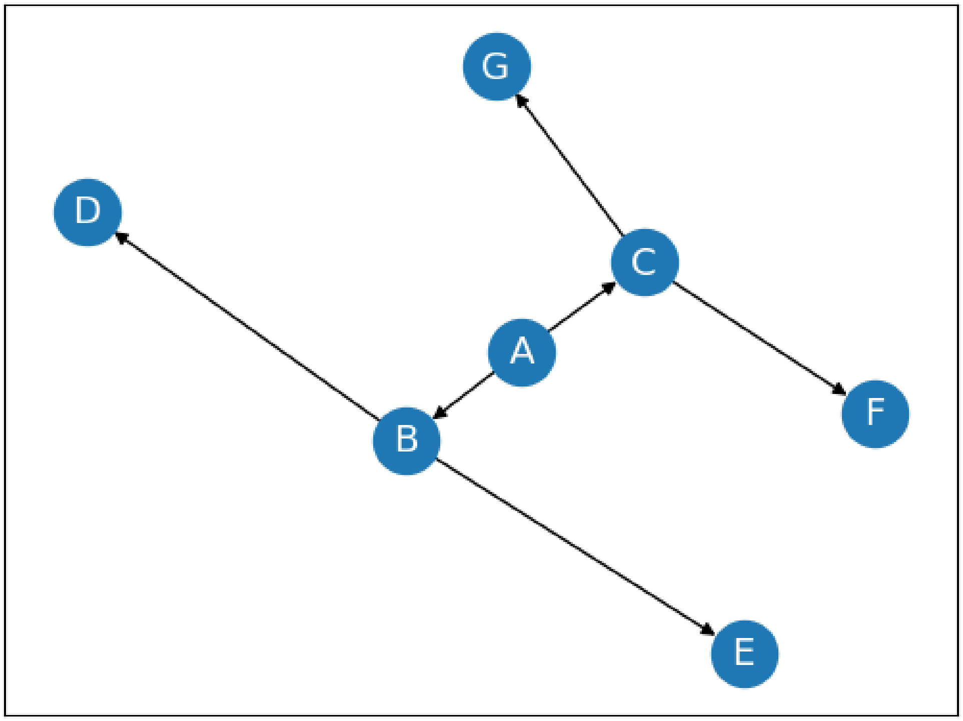
1. Directed Graphs
2. Undirected Graphs
3. Weighted Graph
4. Cyclic Graph
5. Acyclic Graph
6. Directed Acyclic Graph

Directed Graphs

In a directed graph, all the edges are directed. That means, each edge is associated with a direction. For example, if there is an edge from node A to node B, then the edge is directed from A to B and not the other way around.

Directed graph, also called a digraph.

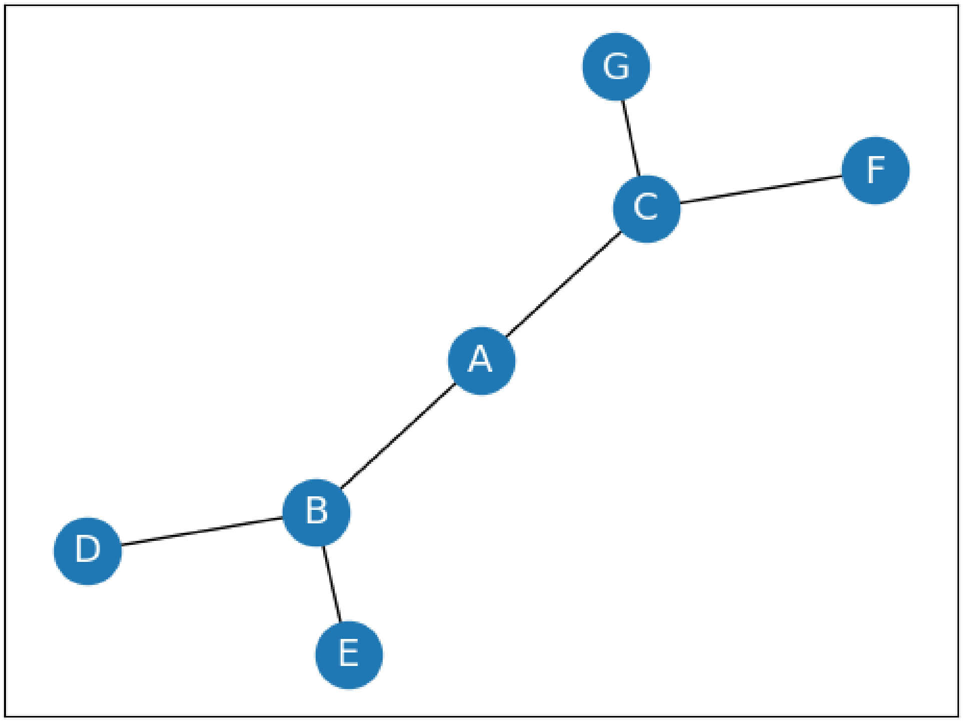
```
DG = nx.DiGraph()
DG.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'),
('B', 'E'), ('C', 'F'), ('C', 'G')])
nx.draw_networkx(DG, pos=nx.spring_layout(DG, seed=0), node_size=600, cmap='coolwarm',␣
↪font_size=14, font_color='white')
```



Undirected Graphs

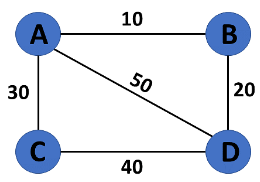
In an undirected graph, all the edges are undirected. That means, each edge is associated with a direction. For example, if there is an edge from node A to node B, then the edge is directed from A to B and not the other way around.

```
G = nx.Graph()
G.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'),
('B', 'E'), ('C', 'F'), ('C', 'G')])

nx.draw_networkx(G, pos=nx.spring_layout(G, seed=0), node_size=600, cmap='coolwarm',␣
↪font_size=14, font_color='white')
```



Weighted Graph

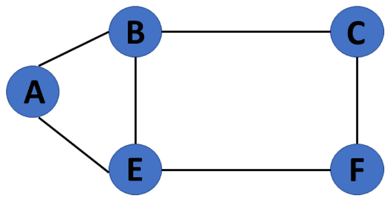
In a weighted graph, each edge is assigned a weight or a cost. The weight can be positive, negative or zero. The weight of an edge is represented by a number. A graph G= (V, E) is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



```
WG = nx.Graph()
WG.add_edges_from([('A', 'B', {"weight": 10}), ('A', 'C', {"weight": 20}), ('B', 'D', {
↪"weight": 30}), ('B', 'E', {"weight": 40}), ('C', 'F', {"weight": 50}), ('C', 'G', {
↪"weight": 60})])
labels = nx.get_edge_attributes(WG, "weight")
```

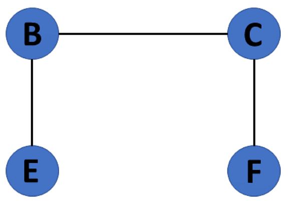
Cyclic Graph

A graph is said to be cyclic if it contains a cycle. A cycle is a path of edges that starts and ends on the same vertex. A graph that contains a cycle is called a cyclic graph.



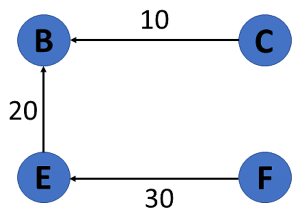
Acyclic Graph

When there are no cycles in a graph, it is called an acyclic graph.



Directed Acyclic Graph

It's also known as a directed acyclic graph (DAG), and it's a graph with directed edges but no cycle. It represents the edges using an ordered pair of vertices since it directs the vertices and stores some data.

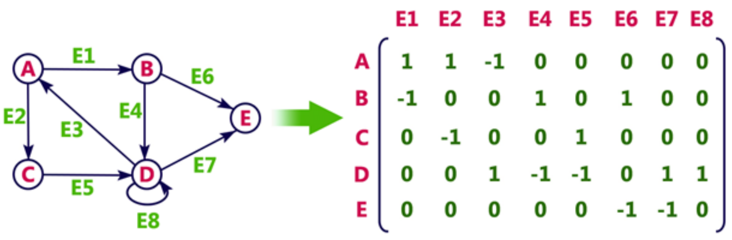


Trees

A tree is a special type of graph that has a root node, and every node in the graph is connected by edges. It's a directed acyclic graph with a single root node and no cycles. A tree is a special type of graph that has a root node, and every node in the graph is connected by edges. It's a directed acyclic graph with a single root node and no cycles.

degree of a vertex

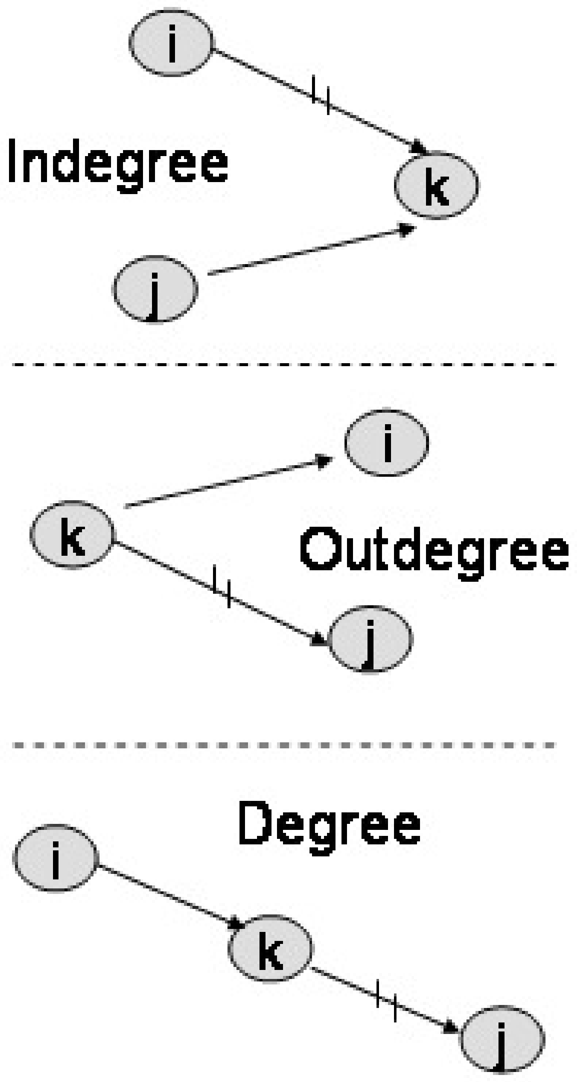
The degree of a vertex is the number of edges incident to it. In the following figure, the degree of vertex A is 3, the degree of vertex B is 4, and the degree of vertex C is 2.



$$\begin{array}{c|cccccccc} & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\ A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

In-Degree and Out-Degree of a Vertex

In a directed graph, the in-degree of a vertex is the number of edges that are incident to the vertex. The out-degree of a vertex is the number of edges that are incident to the vertex.



```
G = nx.Graph()
G.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G
↪')])
print(f"deg(A) = {G.degree['A']}")
DG = nx.DiGraph()
DG.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G
↪')])

print(f"deg^-(A) = {DG.in_degree['A']}")
print(f"deg^+(A) = {DG.out_degree['A']}")
```

```
deg(A) = 2
deg^-(A) = 0
deg^+(A) = 2
```

Path

A path is a sequence of edges that allows you to go from one vertex to another. The length of a path is the number of edges in it.

Cycle

A cycle is a path that starts and ends at the same vertex.

Graph measures

Degrees and paths can be used to determine the importance of a node in a network. This measure is referred to as **centrality**

Centrality quantifies the importance of a vertex or node in a network. It helps us to identify key nodes in a graph based on their connectivity and influence on the flow of information or interactions within the network.

Degree centrality

Degree centrality is one of the simplest and most commonly used measures of centrality. It is simply defined as the degree of the node. A high degree centrality indicates that a vertex is highly connected to other vertices in the graph, and thus significantly influences the network.

Closeness centrality

Closeness centrality measures how close a node is to all other nodes in the graph. It corresponds to the average length of the shortest path between the target node and all other nodes in the graph. A node with high closeness centrality can quickly reach all other vertices in the network.

Betweenness centrality

Betweenness centrality measures the number of times a node lies on the shortest path between pairs of other nodes in the graph. A node with high betweenness centrality acts as a bottleneck or bridge between different parts of the graph.

```python
print(f"Degree centrality      = {nx.degree_centrality(G)}")
print(f"Closeness centrality   = {nx.closeness_centrality(G)}")
print(f"Betweenness centrality = {nx.betweenness_centrality(G)}")
```

```
Degree centrality      = {'A': 0.3333333333333333, 'B': 0.5, 'C': 0.5, 'D': 0.
↪16666666666666666, 'E': 0.16666666666666666, 'F': 0.16666666666666666, 'G': 0.
↪16666666666666666}
Closeness centrality   = {'A': 0.6, 'B': 0.5454545454545454, 'C': 0.5454545454545454, 'D
↪': 0.375, 'E': 0.375, 'F': 0.375, 'G': 0.375}
Betweenness centrality = {'A': 0.6, 'B': 0.6, 'C': 0.6, 'D': 0.0, 'E': 0.0, 'F': 0.0, 'G
↪': 0.0}
```

The importance of nodes A, B and C in a graph depends on the type of centrality used. Degree centrality considers nodes B and C to be more important because they have more neighbors than node A . However, in closeness centrality, node A is the most important as it can reach any other node in the graph in the shortest possible path. On the other hand, nodes A,B and C have equal betweenness centrality, as they all lie on a large number of shortest paths between other nodes.

Density

The density of a graph is the ratio of the number of edges to the number of possible edges. A graph with high density is considered more connected and has more information flow compared to a graph with low density. A dense graph has a density closer to 1, while a sparse graph has a density closer to 0.

```
G = nx.Graph()
G.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G
↪')])
print(f"Density of G = {nx.density(G)}")
```

```
Density of G = 0.2857142857142857
```

Graph Representation

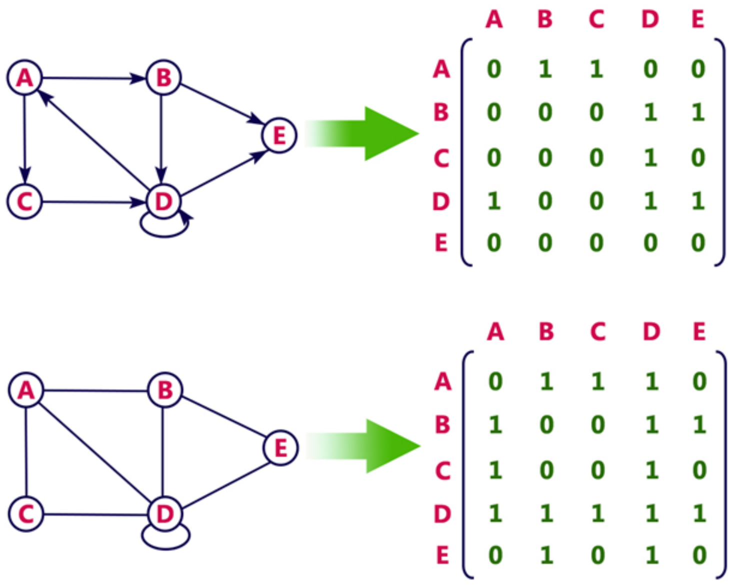
There are two ways to represent a graph:

1. Adjacency Matrix
2. Edge List
3. Adjacency List

Each data structure has its own advantages and disadvantages that depend on the specific application and requirements.

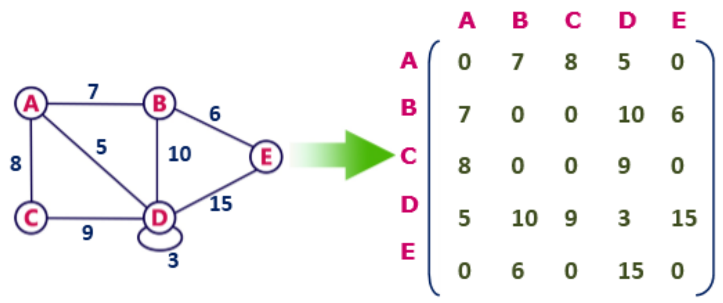
Adjacency Matrix

In an adjacency matrix, each row represents a vertex and each column represents another vertex. If there is an edge between the two vertices, then the corresponding entry in the matrix is 1, otherwise it is 0. The following figure shows an adjacency matrix for a graph with 4 vertices.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 7 & 8 & 5 & 0 \\ B & 7 & 0 & 0 & 10 & 6 \\ C & 8 & 0 & 0 & 9 & 0 \\ D & 5 & 10 & 9 & 3 & 15 \\ E & 0 & 6 & 0 & 15 & 0 \end{array}$$

drawbacks of adjacency matrix

1. The adjacency matrix representation of a graph is not suitable for a graph with a large number of vertices. This is because the number of entries in the matrix is proportional to the square of the number of vertices in the graph.
2. The adjacency matrix representation of a graph is not suitable for a graph with parallel edges. This is because the matrix can only store a single value for each pair of vertices.
3. One of the main drawbacks of using an adjacency matrix is its space complexity: as the number of nodes in the graph grows, the space required to store the adjacency matrix increases exponentially. adjacency matrix has a space complexity of $O\left(|V|^2\right)_{\text{, where}}$ $|V|_{\text{repre-}}$ sents the number of nodes in the graph.

Overall, while the adjacency matrix is a useful data structure for representing small graphs, it may not be practical for larger ones due to its space complexity. Additionally, the overhead of adding or removing nodes can make it inefficient for dynamically changing graphs.

Edge list

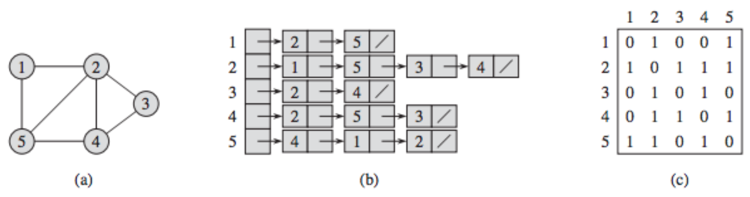
An edge list is a list of all the edges in a graph. Each edge is represented by a tuple or a pair of vertices. The edge list can also include the weight or cost of each edge. This is the data structure we used to create our graphs with networkx:

```
edge_list = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)]
```

checking whether two vertices are connected in an edge list requires iterating through the entire list, which can be time-consuming for large graphs with many edges. Therefore, edge lists are more commonly used in applications where space is a concern.

Adjacency List

In an adjacency list, each vertex stores a list of adjacent vertices. The following figure shows an adjacency list for a graph with 4 vertices.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

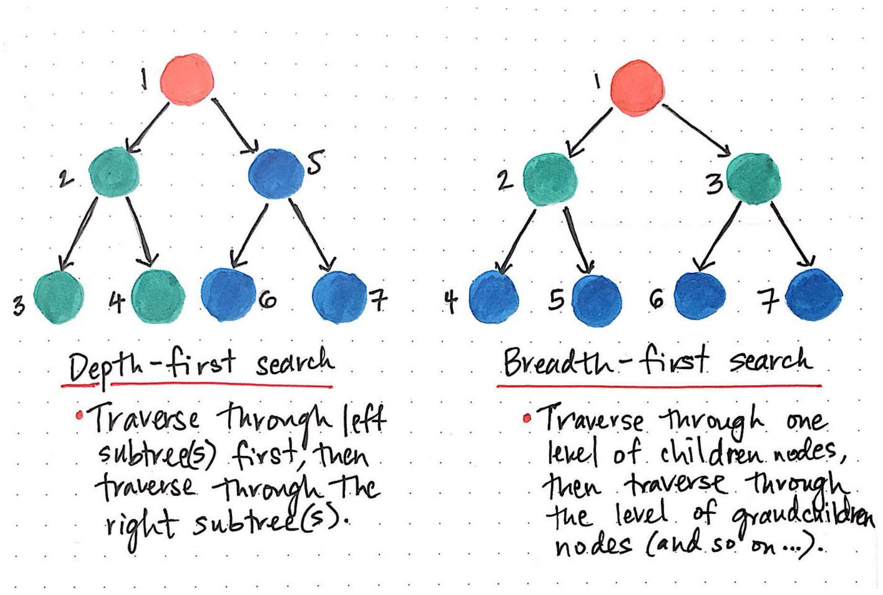
However, checking whether two vertices are connected can be slower than with an adjacency matrix. This is because it requires iterating through the adjacency list of one of the vertices, which can be time-consuming for large graphs.

Graph Traversal

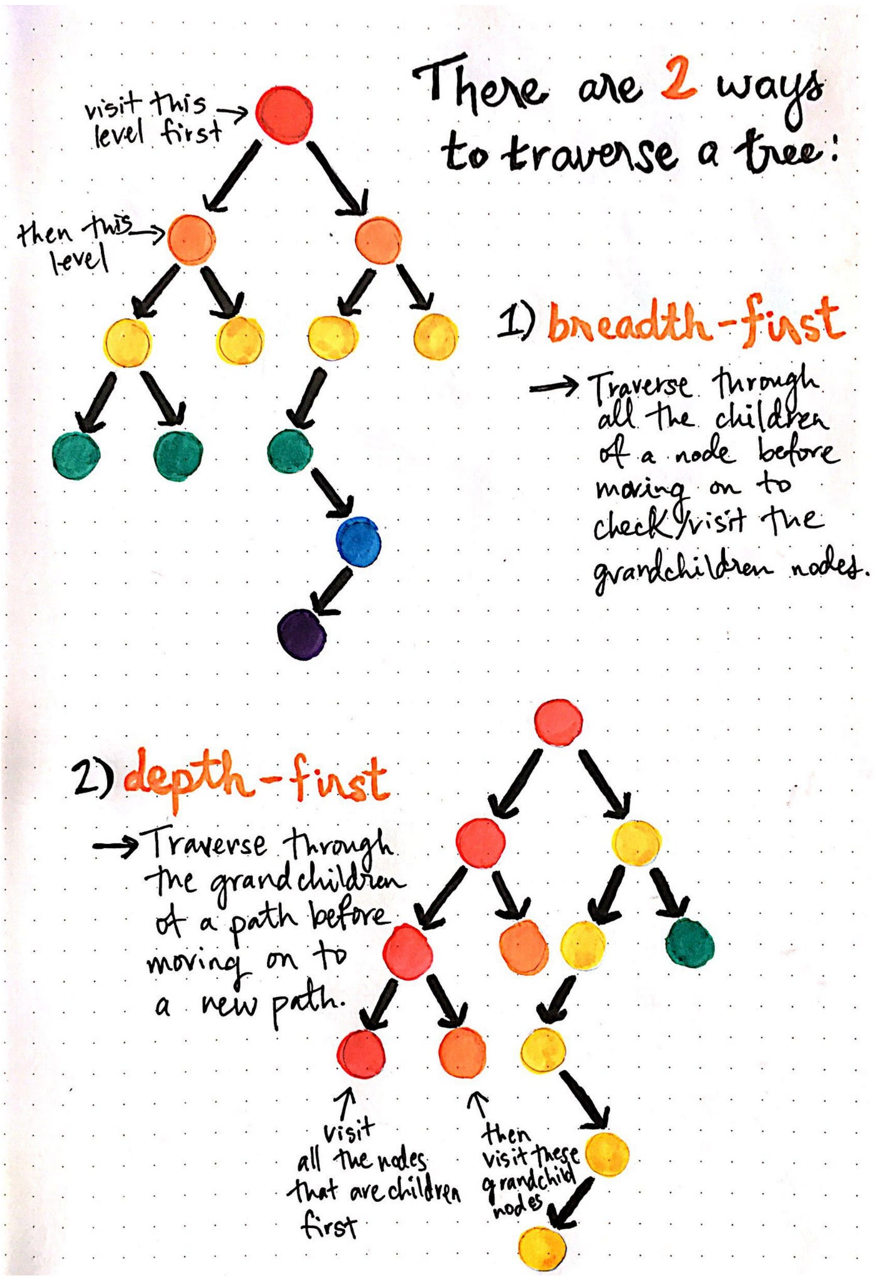
Graph algorithms are critical in solving problems related to graphs, such as finding the shortest path between two nodes or detecting cycles. This section will discuss two graph traversal algorithms: BFS and DFS.

Graph traversal is the process of visiting (checking and/or updating) each vertex in a graph, exactly once. Such traversals are classified by the order in which the vertices are visited. The order may be defined by a specific rule, for example, depth-first search and breadth-first search.

Link: https://medium.com/basecs/breaking-down-breadth-first-search-cebe696709d9


1
2
5
3
4
6
7
Depth-first search
• Traverse through left subtree(s) first, then traverse through the right subtree(s).
1
2
3
4
5
6
7
Breadth-first search
• Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

While DFS uses a stack data structure, BFS leans on the queue data structure.


There are 2 ways to traverse a tree:
visit this level first
then this level
1) breadth-first
→ Traverse through all the children of a node before moving on to check/visit the grandchildren nodes.
2) depth-first
→ Traverse through the grandchildren of a path before moving on to a new path.
visit all the nodes that are children first
then visit these grandchild nodes

Depth First Search

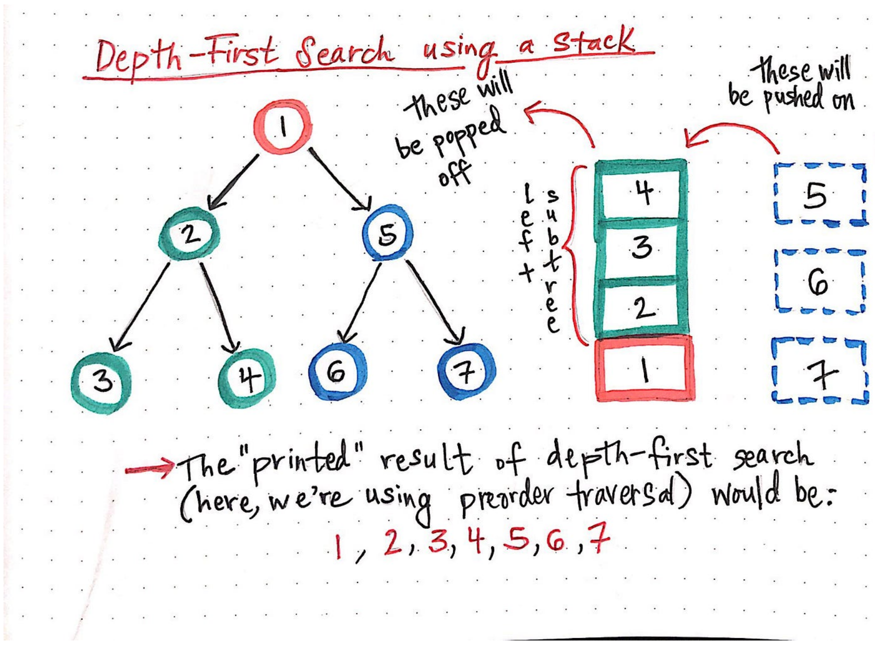
We know that depth-first search is the process of traversing down through one branch of a tree until we get to a leaf, and then working our way back to the "trunk" of the tree. In other words, implementing a DFS means traversing down through the subtrees of a binary search tree.

DFS is a recursive algorithm that starts at the root node and explores as far as possible along each branch before backtracking.

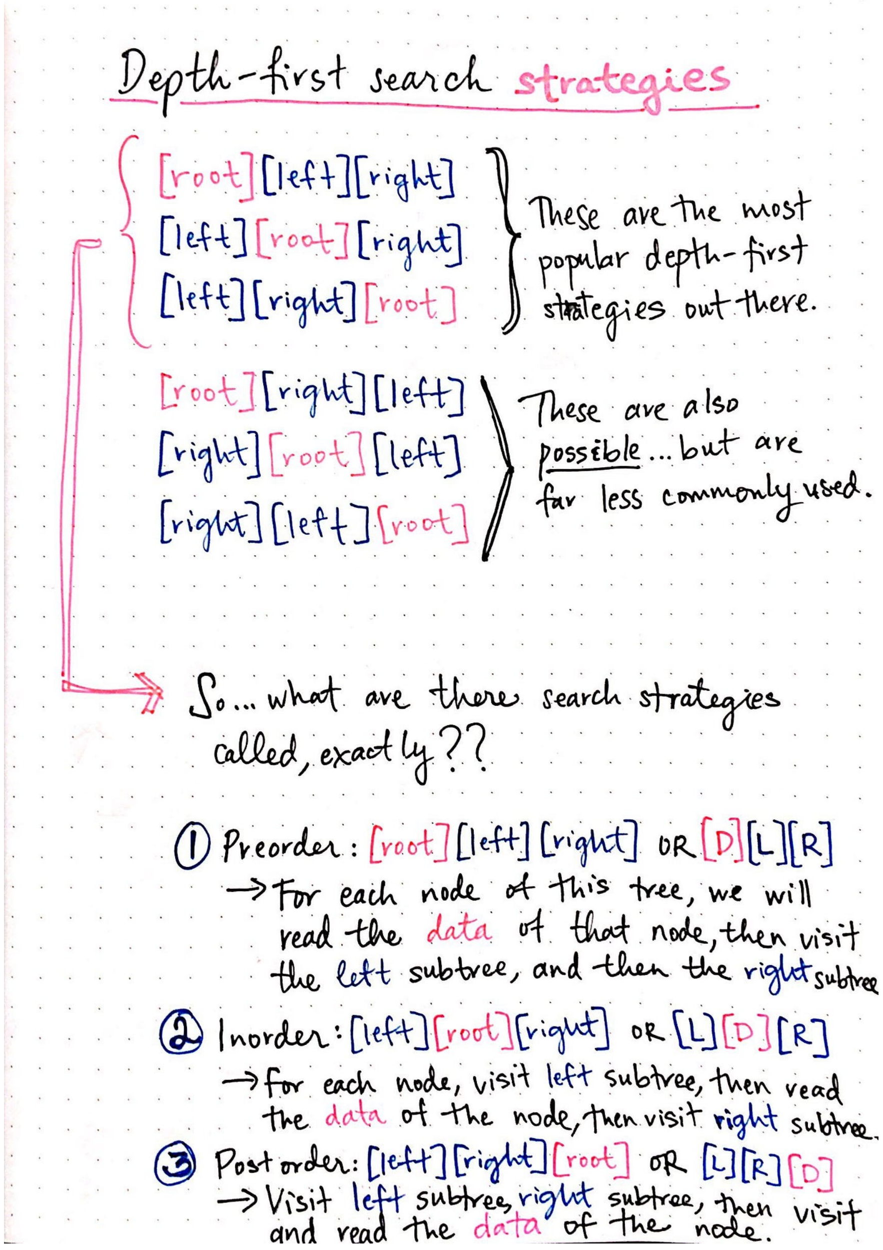
It chooses a node and explores all of its unvisited neighbors, visiting the first neighbor that has not been explored and backtracking only when all the neighbors have been visited. By doing so, it explores the graph by following as deep a path from the starting node as possible before backtracking to explore other branches. This continues until all nodes have been explored.

DFS Algorithm goes 'deep' instead of 'wide'

https://miro.medium.com/v2/resize:fit:1400/1*LUL63FWqneOfsLKqMtHKFw.gif


Depth-First Search using a Stack
these will be popped off
these will be pushed on
left subtree
1
2
5
3
4
6
7
→The "printed" result of depth-first search (here, we're using preorder traversal) would be:
1, 2, 3, 4, 5, 6, 7

In depth-first search, once we start down a path, we don't stop until we get to the end. In other words, we traverse through one branch of a tree until we get to a leaf, and then we work our way back to the trunk of the tree.



Depth-first search strategies

[root][left][right]
[left][root][right]
[left][right][root]

These are the most popular depth-first strategies out there.

[root][right][left]
[right][root][left]
[right][left][root]

These are also possible... but are far less commonly used.

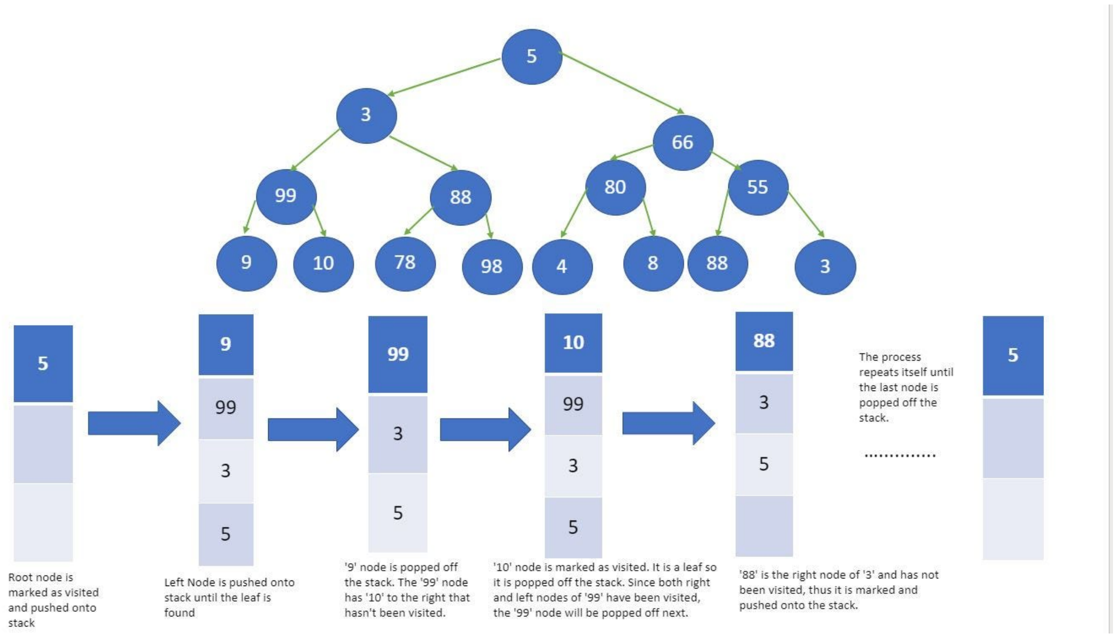
So... what are there search strategies called, exactly??

① Preorder: [root][left][right] OR [D][L][R]
→ For each node of this tree, we will read the data of that node, then visit the left subtree, and then the right subtree

② Inorder: [left][root][right] OR [L][D][R]
→ for each node, visit left subtree, then read the data of the node, then visit right subtree.

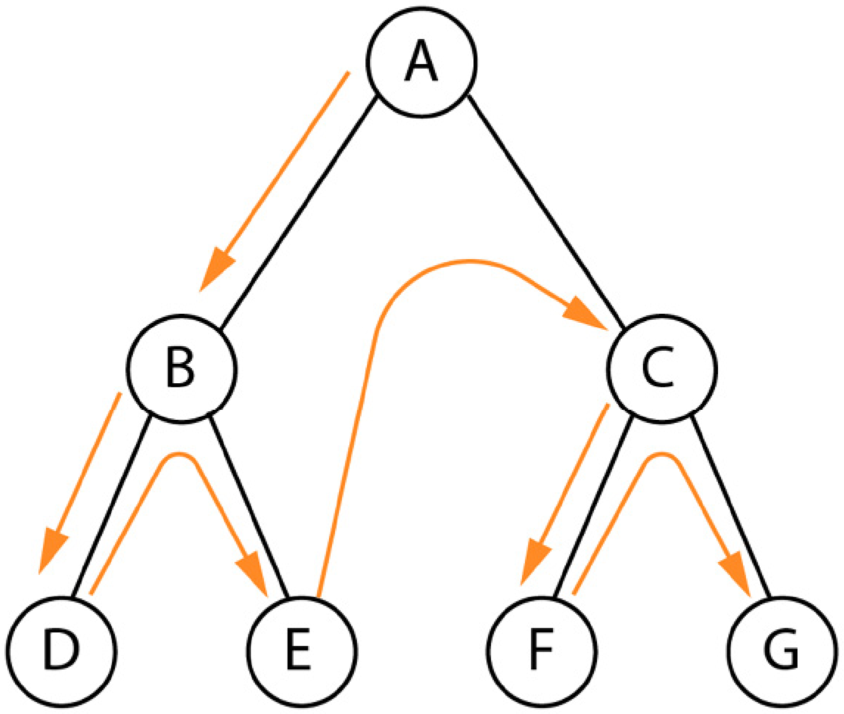
③ Postorder: [left][right][root] OR [L][R][D]
→ Visit left subtree, right subtree, then visit and read the data of the node.

Stack data structure is used to implement DFS. The algorithm starts with a particular node of a graph, then goes to any of its adjacent nodes and repeats this process until it finds the goal. If it reaches a node from which there is no unexplored edge leading to an unvisited node, then it backtracks to the last visited node and repeats the process.


5
3
66
99
88
80
55
9
10
78
98
4
8
88
3
5
9 99 3 5
99 3 5
10 99 3 5
88 3 5
5
Root node is marked as visited and pushed onto stack
Left Node is pushed onto stack until the leaf is found
'9' node is popped off the stack. The '99' node has '10' to the right that hasn't been visited.
'10' node is marked as visited. It is a leaf so it is popped off the stack. Since both right and left nodes of '99' have been visited, the '99' node will be popped off next.
'88' is the right node of '3' and has not been visited, thus it is marked and pushed onto the stack.
The process repeats itself until the last node is popped off the stack.
....

```
G = nx.Graph()
G.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G
↪')])

visited = []

def dfs(visited, graph, node):
    if node not in visited:
        visited.append(node)
    # We then iterate through each neighbor of the current node. For each neighbor, we␣
↪recursively call the dfs() function
    # passing in visited, graph, and the neighbor as arguments:
        for neighbor in graph[node]:
            visited = dfs(visited, graph, neighbor)
    # The dfs() function continues to explore the graph depth-first, visiting all the␣
↪neighbors of each node until there
    # are no more unvisited neighbors. Finally, the visited list is returned
    return visited

dfs(visited, G, 'A')
```

```
['A', 'B', 'D', 'E', 'C', 'F', 'G']
```



Once again, the order we obtained is the one we anticipated in Figure. DFS is useful in solving various problems, such as finding connected components, topological sorting, and solving maze problems. It is particularly useful in finding cycles in a graph since it traverses the graph in a depth-first order, and a cycle exists if, and only if, a node is visited twice during the traversal.

Additionally, many other algorithms in graph theory build upon BFS and DFS, such as Dijkstra's shortest path algorithm, Kruskal's minimum spanning tree algorithm, and Tarjan's strongly connected components algorithm. Therefore, a solid understanding of BFS and DFS is essential for anyone who wants to work with graphs and develop more advanced graph algorithms.

Breadth First Search

Breadth First Search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'[1]), and explores the neighbor nodes first, before moving to the next level neighbors.

It works by maintaining a queue of nodes to visit and marking each visited node as it is added to the queue. The algorithm then dequeues the next node in the queue and explores all its neighbors, adding them to the queue if they haven't been visited yet.

Let's now see how we can implement it in Python

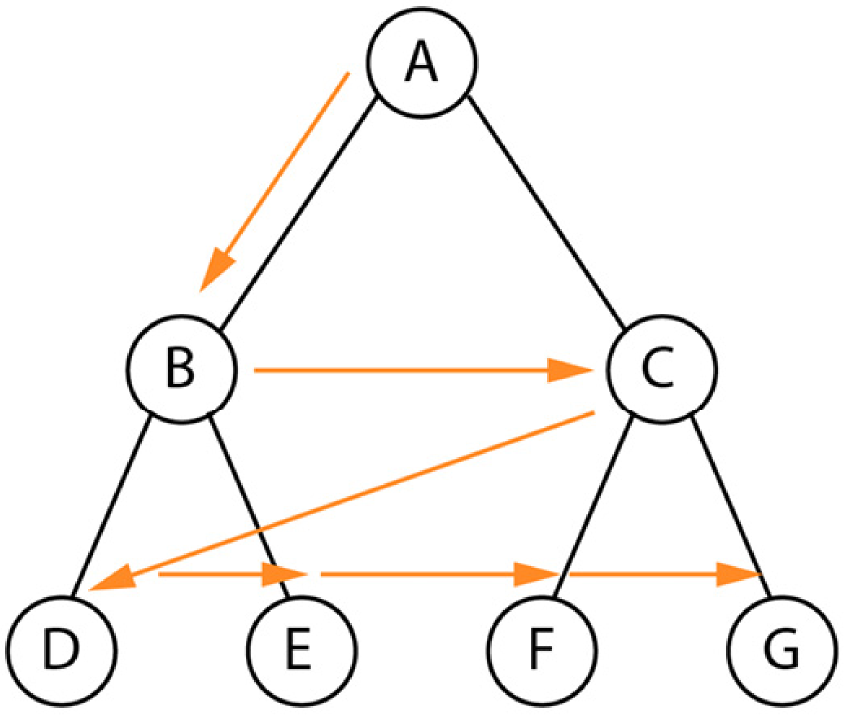
```
def bfs(graph, node):
   # We initialize two lists (visited and queue) and add the starting node. The visited␣
↪list keeps track of the nodes that have been
   # visited #during the search, while the queue list stores the nodes that need to be␣
↪visited:

   visited, queue = [node], [node]

```

(continues on next page)

(continued from previous page)

```
    while queue:
    # When We enter a while loop that continues until the queue list is empty.
    # Inside the loop, we remove the first node in the queue list using the pop(0) method␣
↪and store the result in the node variable

        node = queue.pop(0)
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)
    # We iterate through the neighbors of the node using a for loop. For each neighbor␣
↪that has not been visited yet,
    # we add it to the visited list and to the end of the queue list using the append()␣
↪method. When it's complete,
    # we return the visited list:
    return visited

bfs(G, 'A')
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

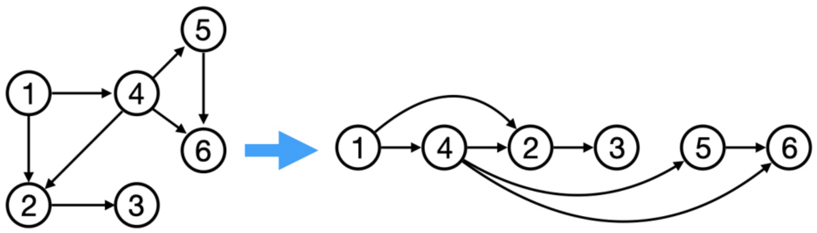


The order we obtained is the one we anticipated in Figure.

BFS is particularly useful in finding the shortest path between two nodes in an unweighted graph. This is because the algorithm visits nodes in order of their distance from the starting node, so the first time the target node is visited, it must be along the shortest path from the starting node.

Topological Sort

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.



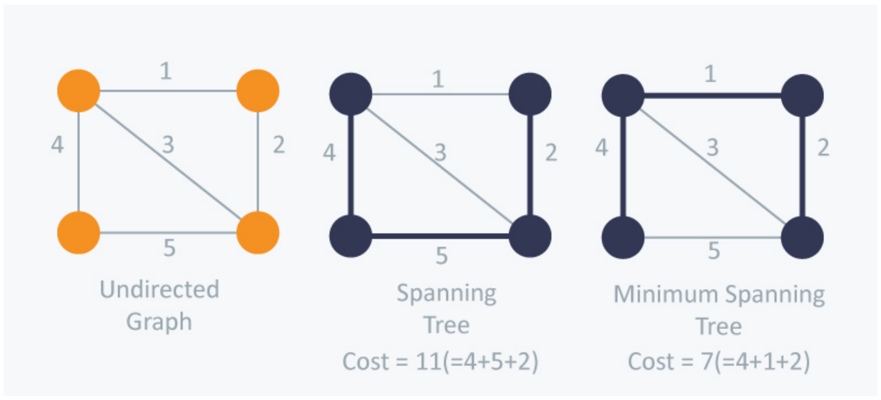
Graph Algorithms

Graph algorithms are used to solve problems that involve graphs. Graph algorithms are used to find the shortest path between two nodes, find the minimum spanning tree, find the strongly connected components, find the shortest path from a single node to all other nodes, find the bridges and articulation points, find the Eulerian path and circuit, find the maximum flow, find the maximum matching, find the biconnected components, find the Hamiltonian path and circuit, find the dominating set, find the shortest path from a single node to all other nodes, find the bridges and articulation points, find the Eulerian path and circuit, find the maximum flow, find the maximum matching, find the biconnected components, find the Hamiltonian path and circuit, find the dominating set, etc.

1.29 Tree Data Structure

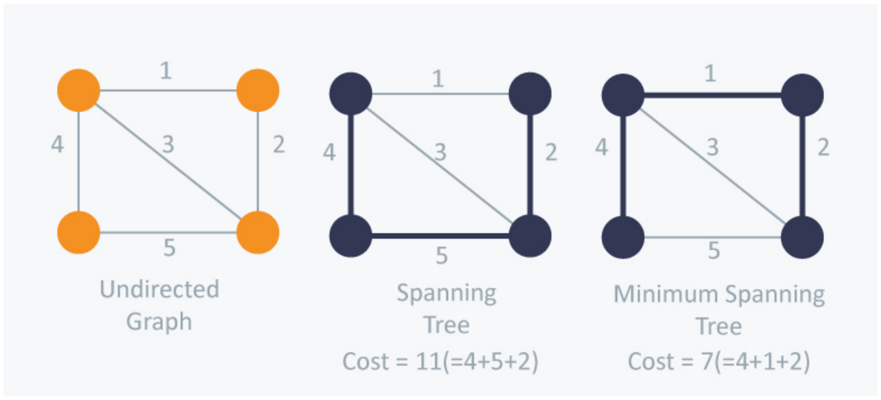
1.29.1 Spanning Trees

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree. The edges may or may not have weights assigned to them.



Minimum Spanning Tree

A minimum spanning tree is a spanning tree with the minimum possible sum of edge weights. The edges may or may not have weights assigned to them.



Finding Minimum Spanning Tree

There are many algorithms to find the minimum spanning tree. The most common ones are:

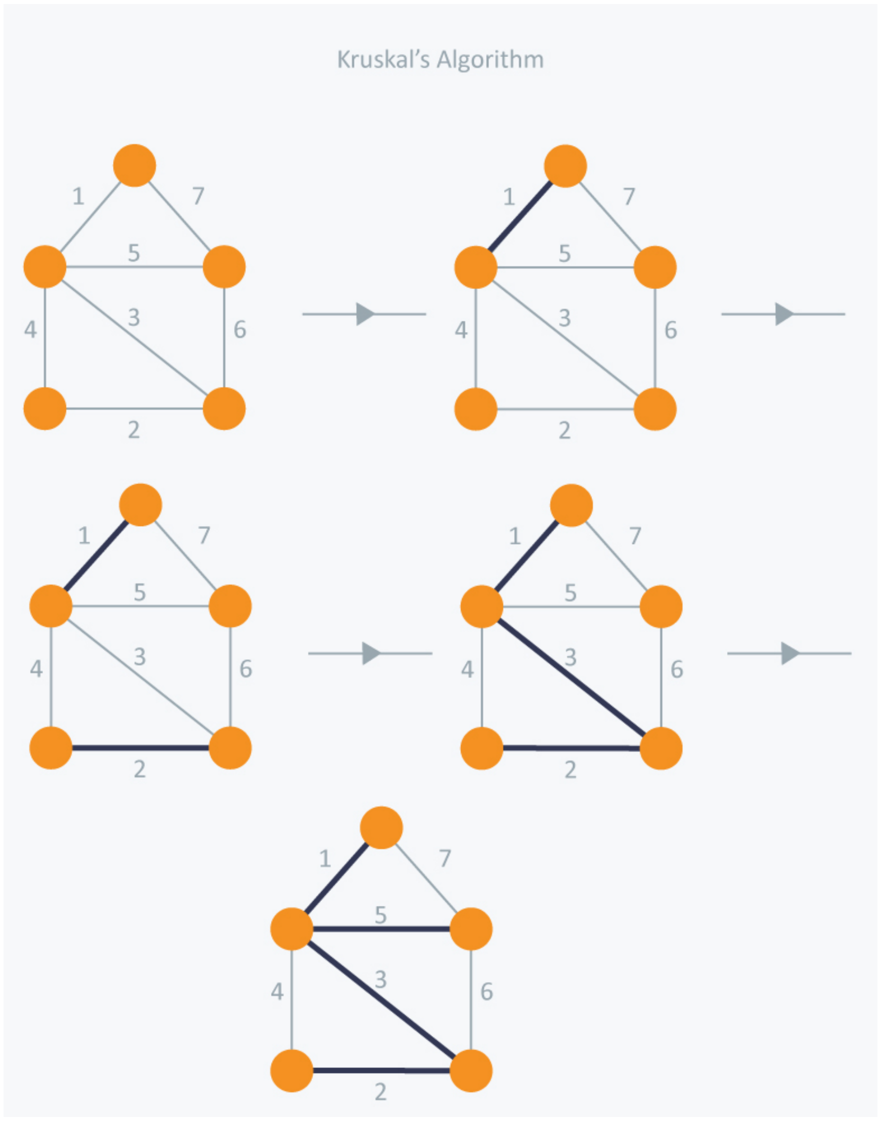
- Kruskal's Algorithm
- Prim's Algorithm

Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).

Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.



Union Find Data Structure

Union Find is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. It supports two operations:

- Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.
- Union: Join two subsets into a single subset.

1.29.2 Amortized Analysis

Amortized analysis is a method of analyzing the costs associated with a data structure that averages the worst operations out over time.

1.30 Shortest Path Algorithms

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum. This problem could be solved easily using (BFS) if all edge weights were 1.

1.30.1 Bellman Ford's Algorithm

Bellman Ford's algorithm is a dynamic programming algorithm that solves the shortest path problem in graphs with negative edge weights. It is a generalization of Dijkstra's algorithm that works on graphs with negative edge weights.

Shortest path contains at most n -1 edges, because the shortest path couldn't have a cycle.

So why shortest path shouldn't have a cycle ? There is no need to pass a vertex again, because the shortest path to all other vertices could be found without the need for a second visit for any vertices.

Algorithm Steps:

1. Initialize the distance of all vertices to infinity.
2. Set the distance of the source vertex to 0.
3. Relax all edges n - 1 times.
4. Check for negative-weight cycles.

1.30.2 Dijkstra's Algorithm

Dijkstra's algorithm is a dynamic programming algorithm that solves the shortest path problem in graphs with non-negative edge weights. It is a generalization of BFS that works on graphs with non-negative edge weights.

Basics of Dijkstra's Algorithm

- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Requirements

Dijkstra's Algorithm can only work with graphs that have positive weights. This is because, during the process, the weights of the edges have to be added to find the shortest path.

If there is a negative weight in the graph, then the algorithm will not work properly. Once a node has been marked as "visited", the current path to that node is marked as the shortest path to reach that node. And negative weights can alter this if the total weight can be decremented after this step has occurred.

1.31 Greedy Algorithms

1.31.1 Divide and Conquer

Divide and conquer is a general algorithm design paradigm: divide the problem into smaller subproblems, solve the subproblems recursively, and then combine the solutions to the subproblems to solve the original problem.

Largest pair sum in an unsorted array

Given an unsorted of distinct integers, find the largest pair sum in it. For example, the largest pair sum in {12, 34, 10, 6, 40} is 74.

Brute force solution

```
numbers = [2,1,0,8,15,7,-1,6]
print(numbers)

max_pair_sum = 0

for i in numbers:
    for j in numbers:
        if i != j:
            max_pair_sum = max(max_pair_sum, i + j)

print(max_pair_sum)

# time Complexity = n^2
# space Complexity = 1
```

```
[2, 1, 0, 8, 15, 7, -1, 6]
23
```

Best solution

1. Initialize the first = Integer.MIN_VALUE second = Integer.MIN_VALUE
2. Loop through the elements a) If the current element is greater than the first max element, then update second max to the first max and update the first max to the current element.
3. Return (first + second)

```
def findLargestSumPair(arr, n):

    # Initialize first and second
```

(continues on next page)

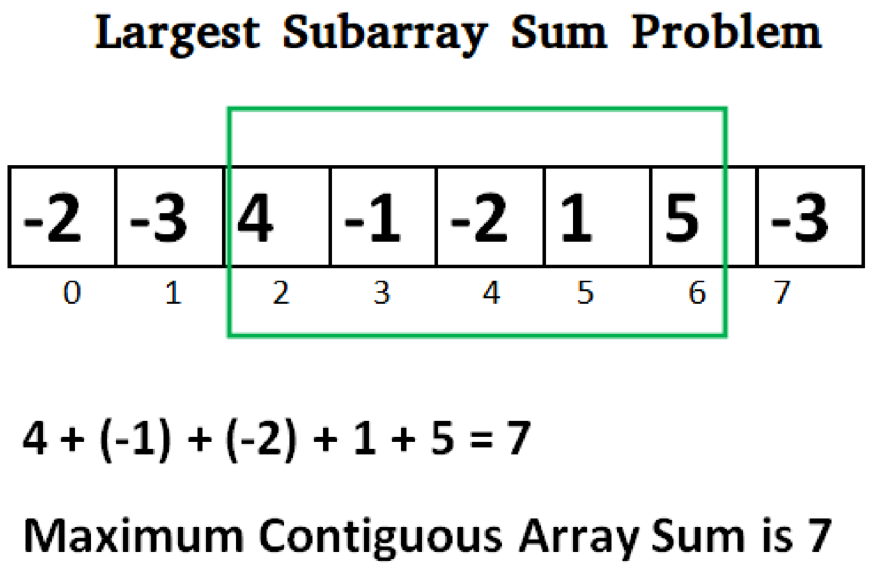
(continued from previous page)

```
    # largest element
    if arr[0] > arr[1]:
        first_big_number = arr[0]
        second_big_number = arr[1]

    else:
        first_big_number = arr[1]
        second_big_number = arr[0]

    # Traverse remaining array and
    # find first and second largest
    # elements in overall array
    for i in range(2, n):

        # If current element is greater
        # than first then update both
        # first and second
        if arr[i] > first_big_number:
            second_big_number = first_big_number
            first_big_number = arr[i]

        # If arr[i] is in between first
        # and second then update second
        elif arr[i] > second_big_number and arr[i] != first_big_number:
            second_big_number = arr[i]

    return (first_big_number , second_big_number)

first, second = findLargestSumPair(numbers, len(numbers))

print(first, second)

print(first + second)

# time Complexity = n
# space Complexity = 1
```

```
15 8
23
```

Max subarray problem

Given an array of integers, find the contiguous subarray that has the largest sum. Return the sum.

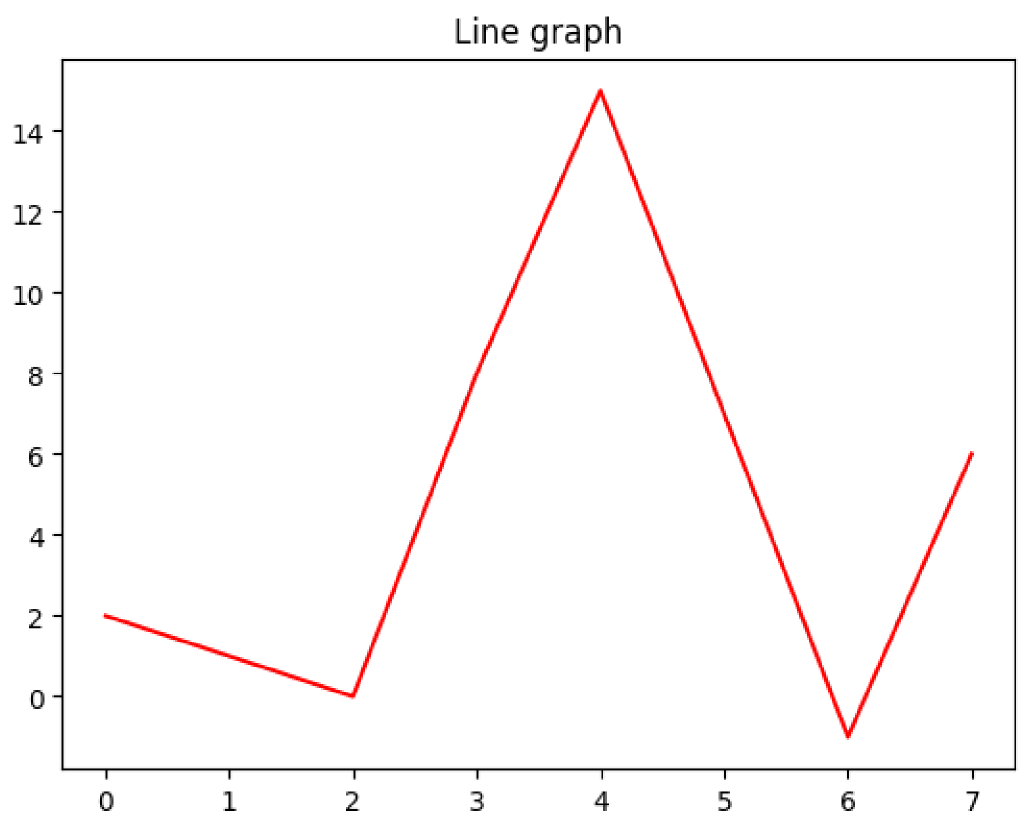


Brute force solution

Draw graph which show as stock values.

```python
import numpy as np
import matplotlib.pyplot as plt

numbers = [2,1,0,8,15,7,-1,6]
print(numbers)

x = range(len(numbers))
y = numbers

plt.title("Line graph")
plt.plot(x, y, color="red")

plt.show()
```

```
[2, 1, 0, 8, 15, 7, -1, 6]
```



```
max_contiguous_sum = 0

for index,v in enumerate(numbers):
    for j in range(index ,len(numbers)):
        # print(index, j)
        # print(numbers[index:j+1])
        max_contiguous_sum = max(max_contiguous_sum, sum(numbers[index:j+1]))

print(max_contiguous_sum)
```

```
38
```

Divide and Conquer solution

```
import sys

def maxSubArraySum(arr):
    # Base case: when there is only one element in the array
    if len(arr) == 1:
        return arr[0]

    # Recursive case: divide the problem into smaller sub-problems
    m = len(arr) // 2

    # Find the maximum subarray sum in the left half
    left_max = maxSubArraySum(arr[:m])
```

(continues on next page)

(continued from previous page)

```
    # Find the maximum subarray sum in the right half
    right_max = maxSubArraySum(arr[m:])

    # Find the maximum subarray sum that crosses the middle element
    left_sum = -sys.maxsize - 1
    right_sum = -sys.maxsize - 1
    sum = 0

    # Traverse the array from the middle to the right
    for i in range(m, len(arr)):
        sum += arr[i]
        right_sum = max(right_sum, sum)

    sum = 0

    # Traverse the array from the middle to the left
    for i in range(m - 1, -1, -1):
        sum += arr[i]
        left_sum = max(left_sum, sum)

    cross_max = left_sum + right_sum

    # Return the maximum of the three subarray sums
    return max(cross_max, max(left_max, right_max))

print(maxSubArraySum(numbers))
```

```
38
```

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

```
def max_subarray(arr):
    if len(arr) == 1:
        return arr[0]
    else:
        mid = len(arr) // 2
        left = max_subarray(arr[:mid])
        right = max_subarray(arr[mid:])
        cross = max_crossing_subarray(arr, mid)
        return max(left, right, cross)
```

```
def max_crossing_subarray(arr, mid):
    left_sum = -np.inf
    right_sum = -np.inf
    sum = 0
    for i in range(mid - 1, -1, -1):
        sum += arr[i]
```

(continues on next page)

(continued from previous page)

```
        if sum > left_sum:
            left_sum = sum
            max_left = i
    sum = 0
    for j in range(mid, len(arr)):
        sum += arr[j]
        if sum > right_sum:
            right_sum = sum
            max_right = j
    return left_sum + right_sum
```

```
arr = np.random.randint(-10, 10, 10)
arr = np.array(numbers)
```

```
max_subarray(arr)
```

```
38
```

```
def max_subarray(arr):
    max_sum = -np.inf
    for i in range(len(arr)):
        sum = 0
        for j in range(i, len(arr)):
            sum += arr[j]
            if sum > max_sum:
                max_sum = sum
                max_left = i
                max_right = j
    return max_sum
```

```
max_subarray(arr)
```

```
38
```

Fast Fourier Transform Algorithm

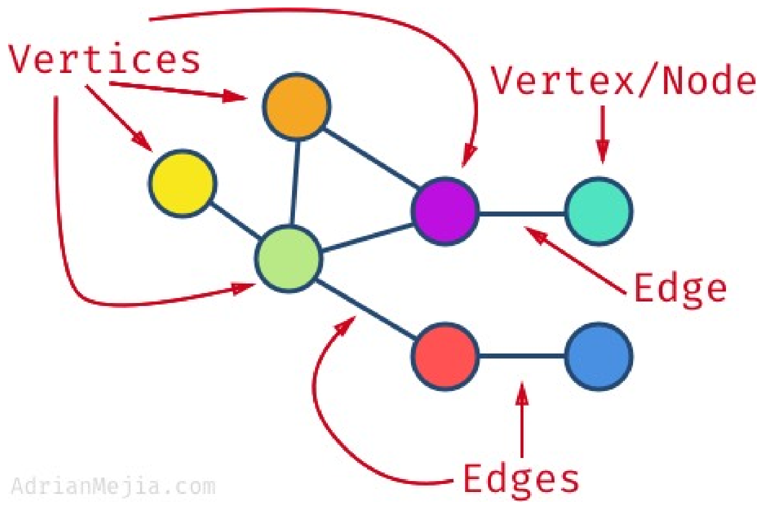
1.32 What is Graphs Theory

Mathematics

In mathematics, graph theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects. A graph in this context is made up of vertices (also called nodes or points) which are connected by edges (also called links or lines).

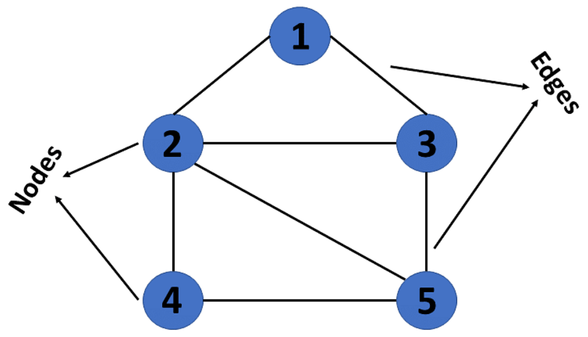
Computer Science

it is considered an abstract data type that is really good for representing connections or relations – unlike the tabular data structures of relational database systems, which are ironically very limited in expressing relations.



1.32.1 Graph Data Structure

A graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as, a Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.



In graph theory, a graph is a mathematical structure consisting of a set of objects, called vertices or nodes, and a set of connections, called edges, which link pairs of vertices. The notation:

$$G = (V, E)$$

is used to represent a graph, where G is the graph, V is the set of vertices, and $\bigvee$ is the set of edges.

The nodes of a graph can represent any objects, such as cities, people, web pages, or molecules, and the edges represent the relationships or connections between them.

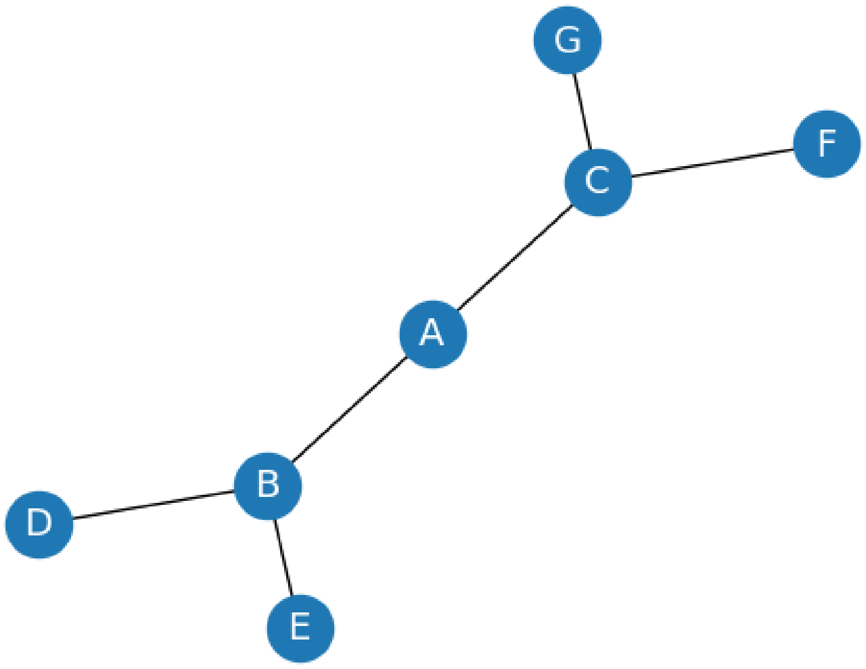
```python
import networkx as nx
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
G = nx.Graph()
G.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G
↪')])

plt.axis('off')
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=0),
                 node_size=600,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white',
                 )
plt.show()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/ml-math/envs/latest/lib/python3.11/site-
↪packages/networkx/drawing/nx_pylab.py:450: UserWarning: No data for colormapping␣
↪provided via 'c'. Parameters 'cmap' will be ignored
  node_collection = ax.scatter(
```



1.32.2 Graphs Terminology

The following are the most commonly used terms in graph theory with respect to graphs:

1. Vertex - A vertex, also called a "node", is a fundamental part of a graph. In the context of graphs, a vertex is an object which may contain zero or more items called attributes.
2. Edge - An edge is a connection between two vertices. An edge may contain a weight/value/cost.
3. Path - A path is a sequence of edges connecting a sequence of vertices.
4. Cycle - A cycle is a path of edges that starts and ends on the same vertex.
5. Weighted Graph/Network - A weighted graph is a graph with numbers assigned to its edges. These numbers are called weights.
6. Unweighted Graph/Network - An unweighted graph is a graph in which all edges have equal weight.
7. Directed Graph/Network - A directed graph is a graph where all the edges are directed.
8. Undirected Graph/Network - An undirected graph is a graph where all the edges are not directed.
9. Adjacent Vertices - Two vertices in a graph are said to be adjacent if there is an edge connecting them.

1.32.3 Types of Graphs

There are many types of graphs:

1. Directed Graphs
2. Undirected Graphs
3. Weighted Graph
4. Cyclic Graph
5. Acyclic Graph
6. Directed Acyclic Graph

Graphs have many properties, including the direction of travel for each relationship. The decision of which graph type you use usually depends on the use case.

There may not be an explicit direction from one node to another, such as with friendships in a social network, but in others, there might be clearly defined directions, such as flights and airports dataset.

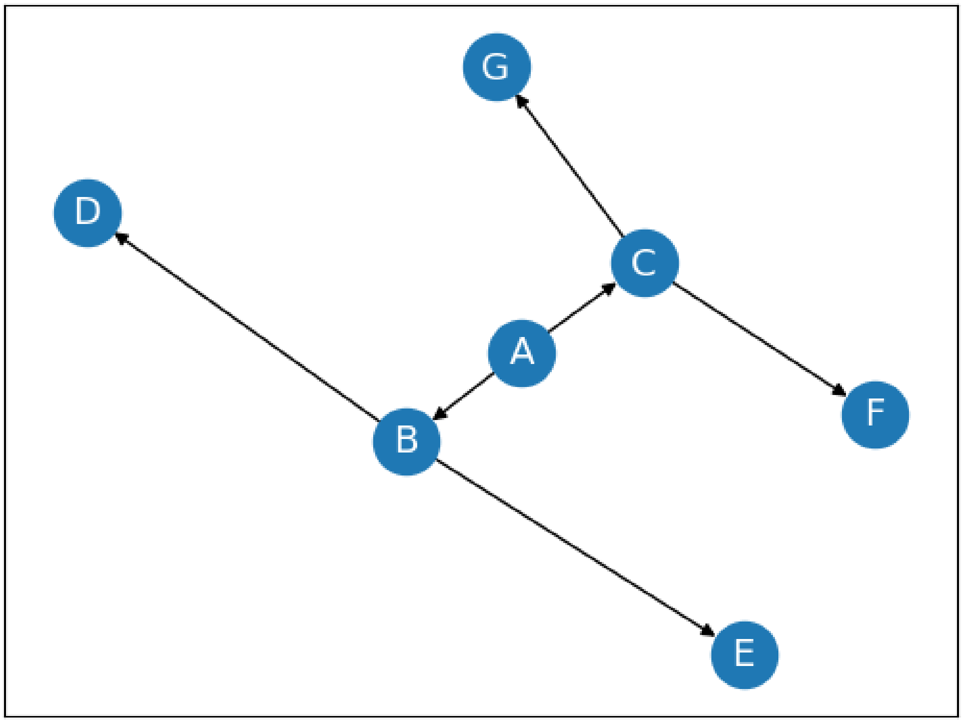
Directed Graphs

In a directed graph, all the edges are directed. That means, each edge is associated with a direction. For example, if there is an edge from node A to node B, then the edge is directed from A to B and not the other way around.

Directed graph, also called a digraph.

https://media.geeksforgeeks.org/wp-content/cdn-uploads/SCC1.png

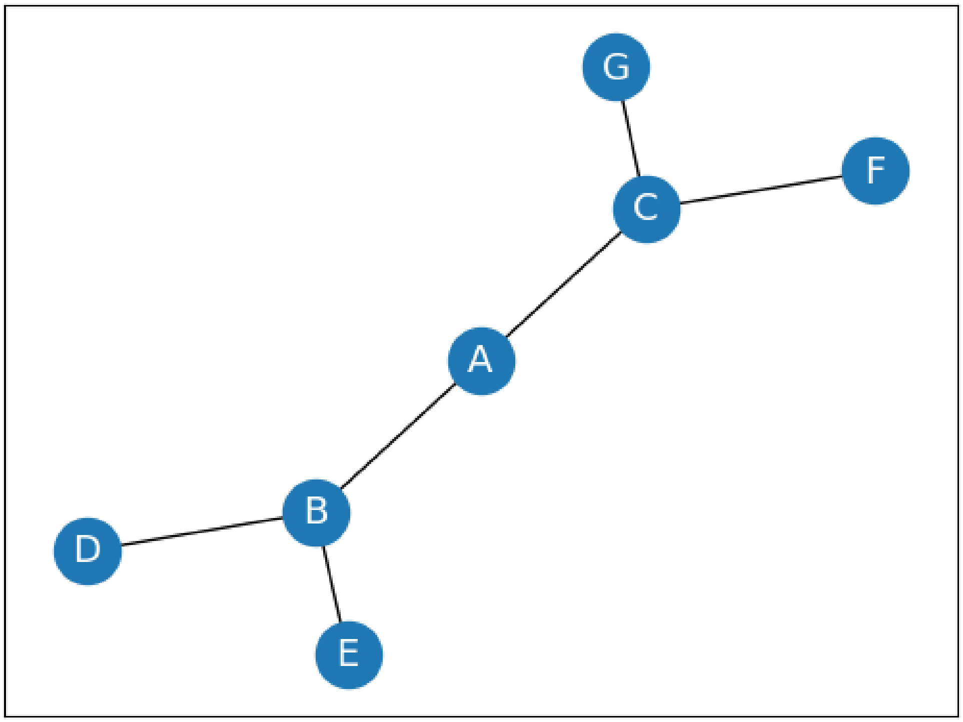
```
DG = nx.DiGraph()
DG.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'),
('B', 'E'), ('C', 'F'), ('C', 'G')])
nx.draw_networkx(DG, pos=nx.spring_layout(DG, seed=0), node_size=600, cmap='coolwarm',␣
↪font_size=14, font_color='white')
plt.show()
```



Undirected Graphs

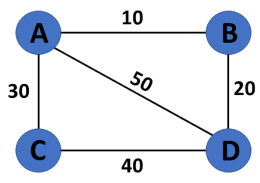
In an undirected graph, all the edges are undirected. That means, each edge is associated with a direction. For example, if there is an edge from node A to node B, then the edge is directed from A to B and not the other way around.

```
G = nx.Graph()
G.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'),
('B', 'E'), ('C', 'F'), ('C', 'G')])

nx.draw_networkx(G, pos=nx.spring_layout(G, seed=0), node_size=600, cmap='coolwarm',␣
↪font_size=14, font_color='white')
plt.show()
```



Weighted Graph

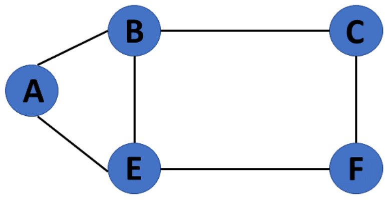
In a weighted graph, each edge is assigned a weight or a cost. The weight can be positive, negative or zero. The weight of an edge is represented by a number. A graph G= (V, E) is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



```
WG = nx.Graph()
WG.add_edges_from([('A', 'B', {"weight": 10}), ('A', 'C', {"weight": 20}), ('B', 'D', {
↪"weight": 30}), ('B', 'E', {"weight": 40}), ('C', 'F', {"weight": 50}), ('C', 'G', {
↪"weight": 60})])
labels = nx.get_edge_attributes(WG, "weight")
```

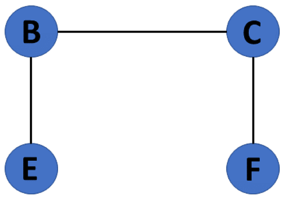
Cyclic Graph

A graph is said to be cyclic if it contains a cycle. A cycle is a path of edges that starts and ends on the same vertex. A graph that contains a cycle is called a cyclic graph.



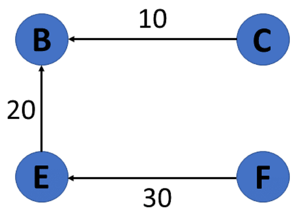
Acyclic Graph

When there are no cycles in a graph, it is called an acyclic graph.



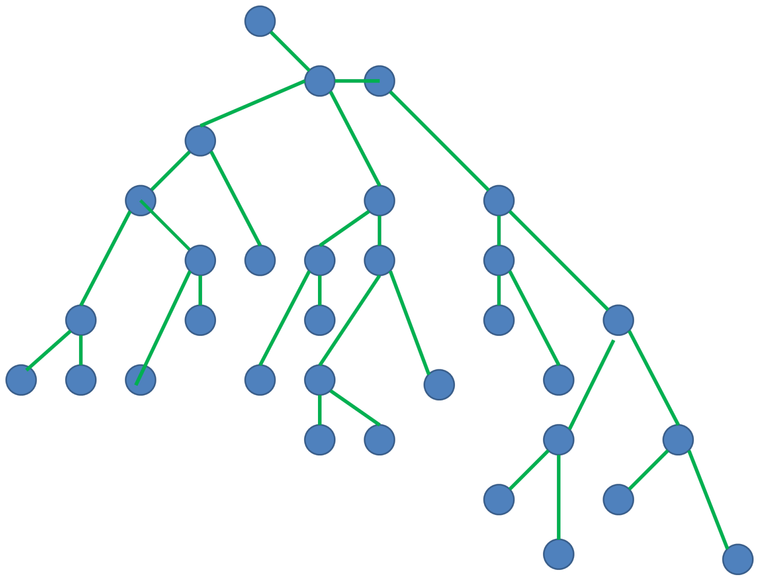
Directed Acyclic Graph

It's also known as a directed acyclic graph (DAG), and it's a graph with directed edges but no cycle. It represents the edges using an ordered pair of vertices since it directs the vertices and stores some data.



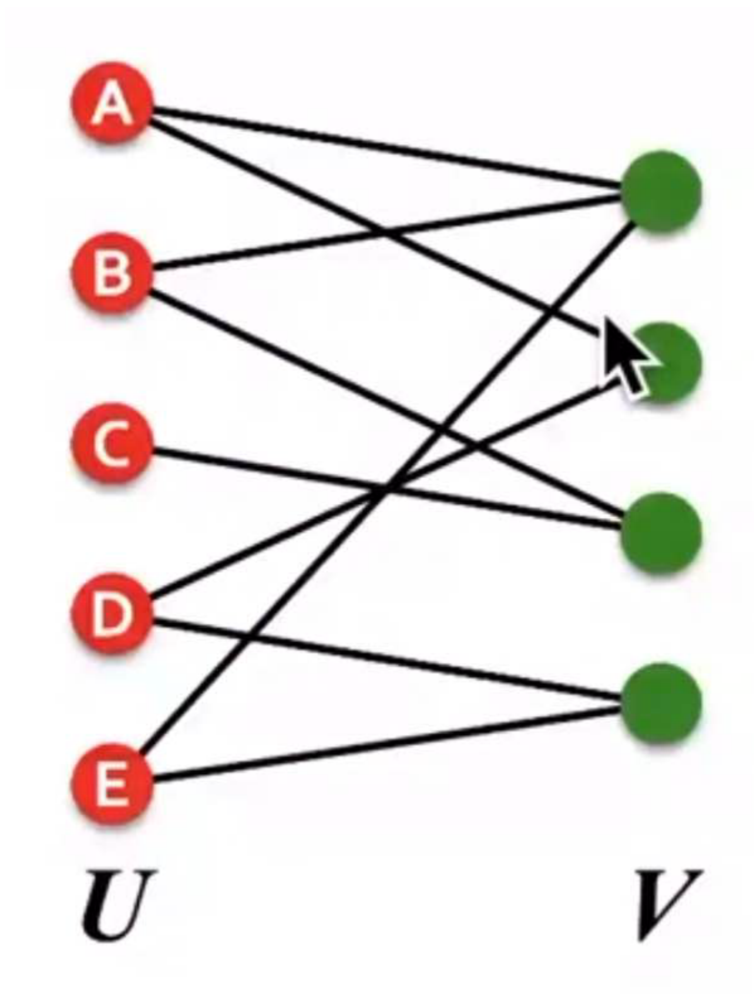
Trees

A tree is a special type of graph that has a root node, and every node in the graph is connected by edges. It's a directed acyclic graph with a single root node and no cycles. A tree is a special type of graph that has a root node, and every node in the graph is connected by edges. It's a directed acyclic graph with a single root node and no cycles.



Biprartite Graph

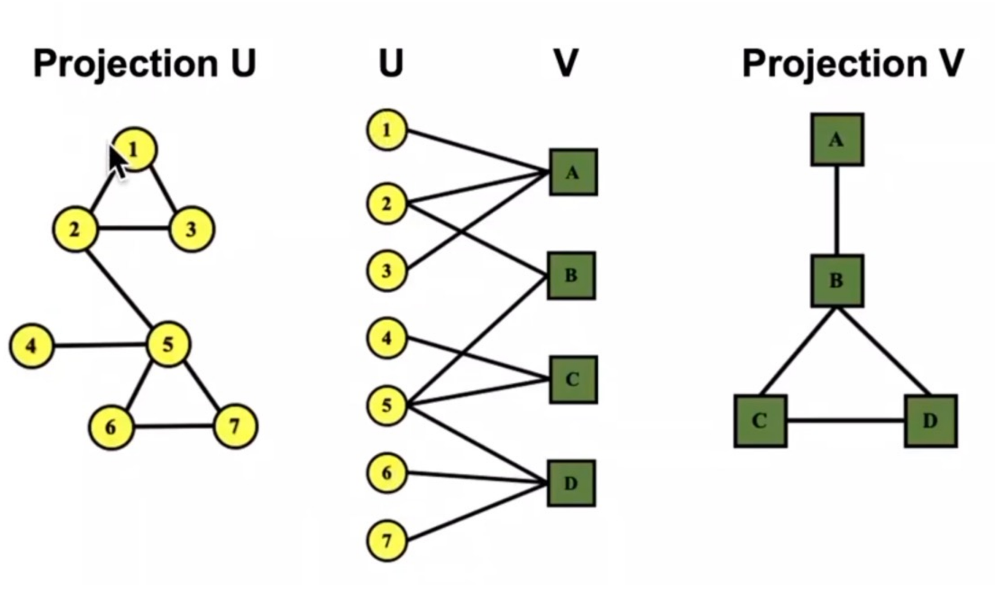
A bipartite graph is a graph whose vertices can be divided into two independent sets, U and V, such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.



Examples of Bipartite Graphs

- Authors-to-Papers (they authored)
- Actors-to-Movies (they appeared in)
- Users-to-Movies (they rated)
- Recipes-to-Ingredients (they contain)
- Author collaboration networks
- Movie co-rating networks

Projections of Bipartite Graphs



Homogeneous graph

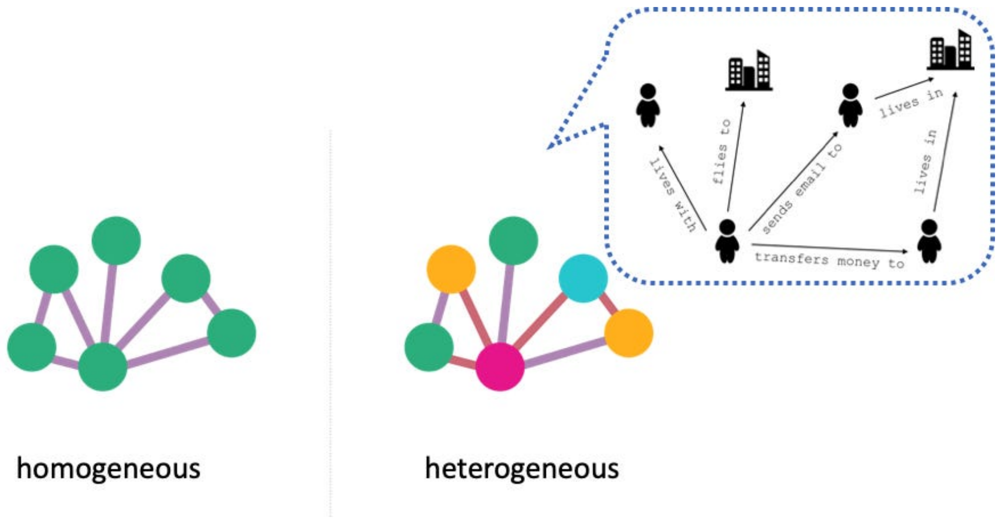
Homogeneous graph consists of one type of nodes and edges, and heterogeneous graph has multiple types of nodes or edges.

An example of a homogeneous graph is an online social network with nodes representing people and edges representing friendship. means we have same node for all types of node and similary all edges are same.

Heterogeneous graph

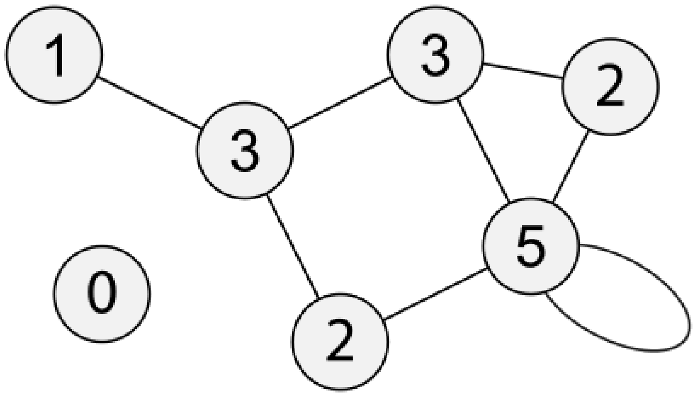
Heterogeneous graphs come with different types of information attached to nodes and edges. Thus, a single node or edge feature tensor cannot hold all node or edge features of the whole graph, due to differences in type and dimensionality.

A graph with two or more types of node and/or two or more types of edge is called heterogeneous. An online social network with edges of different types, say ‘friendship’ and ‘co-worker’, between nodes of ‘person’ type is an example of a heterogeneous



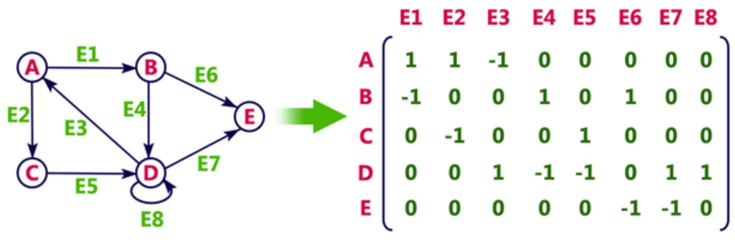
1.32.4 Node degrees

The degree of a node is the number of edges connected to it. A node with no edges is called an isolated node. A node with only one edge is called a leaf node.



Degree of a vertex/Node

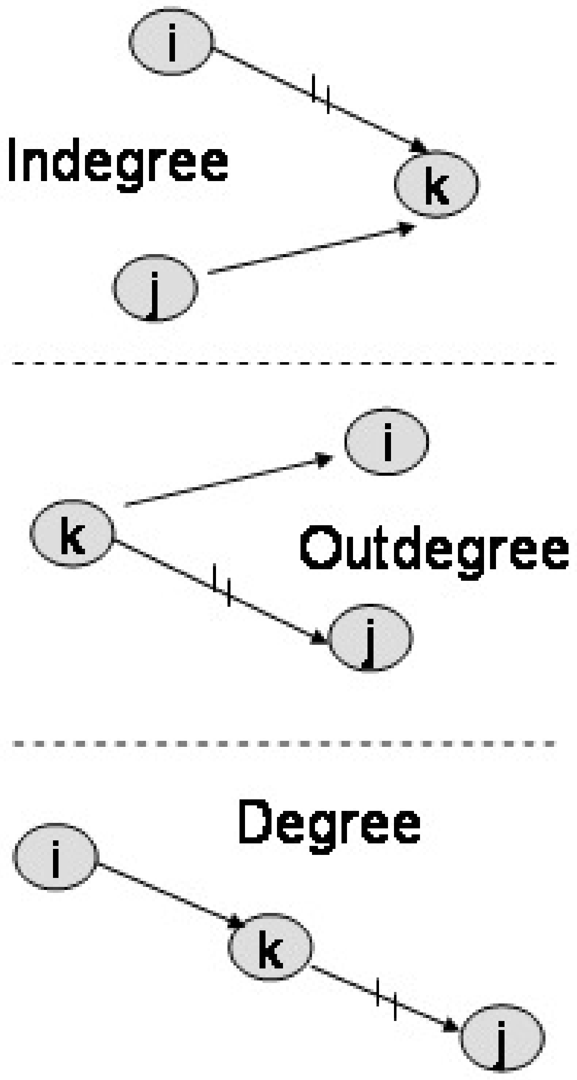
The degree of a vertex is the number of edges incident to it. In the following figure, the degree of vertex A is 3, the degree of vertex B is 4, and the degree of vertex C is 2.



$$\begin{array}{c|cccccccc} & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\ \hline A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

In-Degree and Out-Degree of a Vertex/node

In a directed graph, the in-degree of a vertex is the number of edges that are incident to the vertex. The out-degree of a vertex is the number of edges that are incident to the vertex.



```
G = nx.Graph()
G.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G
↪')])
print(f"deg(A) = {G.degree['A']}")
DG = nx.DiGraph()
DG.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G
↪')])

print(f"deg^-(A) = {DG.in_degree['A']}")
print(f"deg^+(A) = {DG.out_degree['A']}")
```

(continues on next page)

(continued from previous page)

Danger: Drawsback of Node Degree feature

- The degree k of node v is the number of edges (neighboring nodes) the node has
- Treats all neighboring nodes equally.

1.32.5 Node Centrality

Centrality quantifies the importance of a vertex or node in a network. It helps us to identify key nodes in a graph based on their connectivity and influence on the flow of information or interactions within the network.

There are several measures of centrality, each providing a different perspective on the importance of a node:

- Degree centrality
- Engienvector centrality
- Betweenness centrality
- Closeness centrality

Note

- Node degree counts the neighboring nodes without capturing their importance.
- Node centrality c_v takes the node importance in a graph into account

1.32.6 Graph Representation

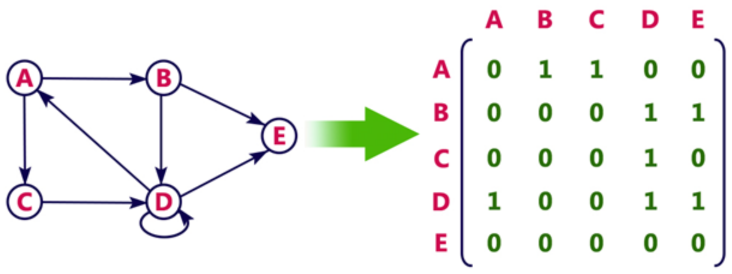
There are two ways to represent a graph:

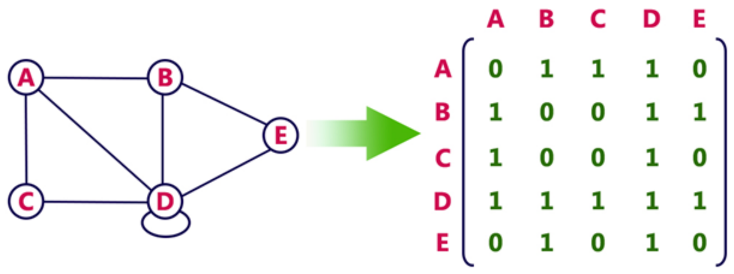
1. Adjacency Matrix
2. Edge List
3. Adjacency List

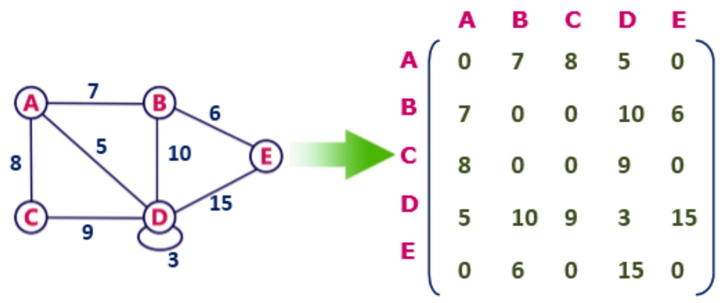
Each data structure has its own advantages and disadvantages that depend on the specific application and requirements.

Adjacency Matrix

In an adjacency matrix, each row represents a vertex and each column represents another vertex. If there is an edge between the two vertices, then the corresponding entry in the matrix is 1, otherwise it is 0. The following figure shows an adjacency matrix for a graph with 4 vertices.

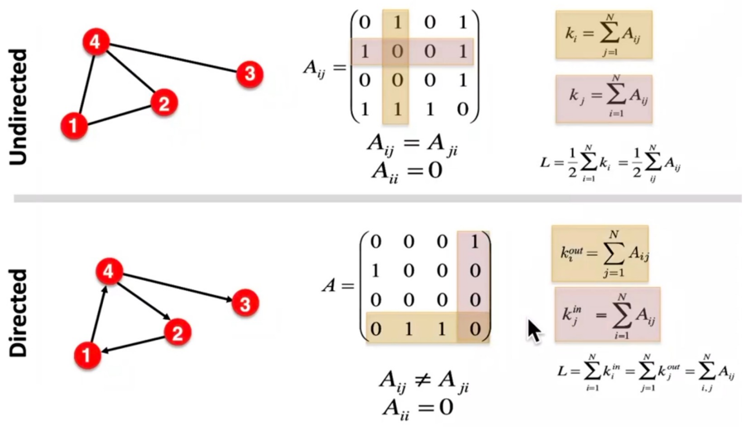






Calculating the degree of node in adjacency matrix

Sum all the 1 in the row corresponding to the node. The sum of the elements in the row corresponding to the node is the degree of the node.


Undirected
Directed
1
2
3
4
$A_{ij} = A_{ji}$
$A_{ii} = 0$
$k_i = \sum_{j=1}^{N} A_{ij}$
$k_j = \sum_{i=1}^{N} A_{ij}$
$L = \frac{1}{2}\sum_{i=1}^{N} k_i = \frac{1}{2}\sum_{ij}^{N} A_{ij}$
$A_{ij} \neq A_{ji}$
$k_i^{out} = \sum_{j=1}^{N} A_{ij}$
$k_j^{in} = \sum_{i=1}^{N} A_{ij}$
$L = \sum_{i=1}^{N} k_i^{in} = \sum_{j=1}^{N} k_j^{out} = \sum_{i,j}^{N} A_{ij}$

Drawbacks of adjacency matrix

1. The adjacency matrix representation of a graph is not suitable for a graph with a large number of vertices. This is because the number of entries in the matrix is proportional to the square of the number of vertices in the graph.
2. The adjacency matrix representation of a graph is not suitable for a graph with parallel edges. This is because the matrix can only store a single value for each pair of vertices.
3. One of the main drawbacks of using an adjacency matrix is its space complexity: as the number of nodes in the graph grows, the space required to store the adjacency matrix increases exponentially. adjacency matrix has a space complexity of $O\left(|V|^2\right)$, where $|V|$ repre- sents the number of nodes in the graph.

Overall, while the adjacency matrix is a useful data structure for representing small graphs, it may not be practical for larger ones due to its space complexity. Additionally, the overhead of adding or removing nodes can make it inefficient for dynamically changing graphs.

Edge list

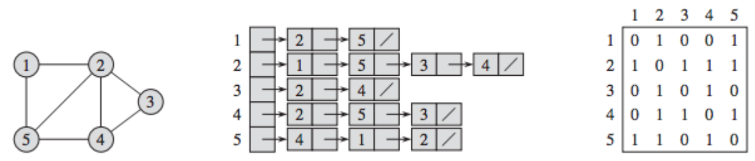
An edge list is a list of all the edges in a graph. Each edge is represented by a tuple or a pair of vertices. The edge list can also include the weight or cost of each edge. This is the data structure we used to create our graphs with networkx:

```
edge_list = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)]
```

checking whether two vertices are connected in an edge list requires iterating through the entire list, which can be time-consuming for large graphs with many edges. Therefore, edge lists are more commonly used in applications where space is a concern.

Adjacency List

In an adjacency list, each vertex stores a list of adjacent vertices. The following figure shows an adjacency list for a graph with 4 vertices.



However, checking whether two vertices are connected can be slower than with an adjacency matrix. This is because it requires iterating through the adjacency list of one of the vertices, which can be time-consuming for large graphs.

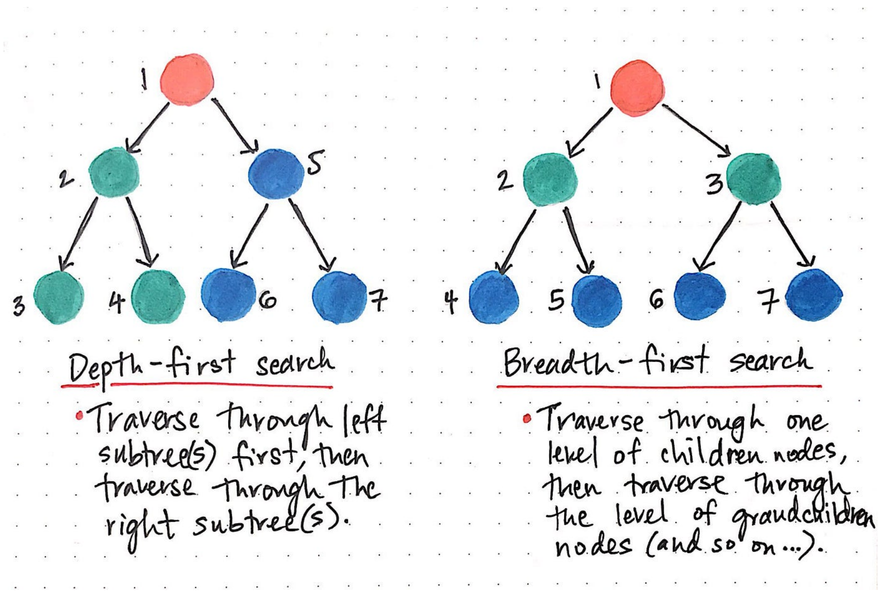
1.32.7 Graph Traversals

Traversal means to walk along the edges of a graph in specific ways.

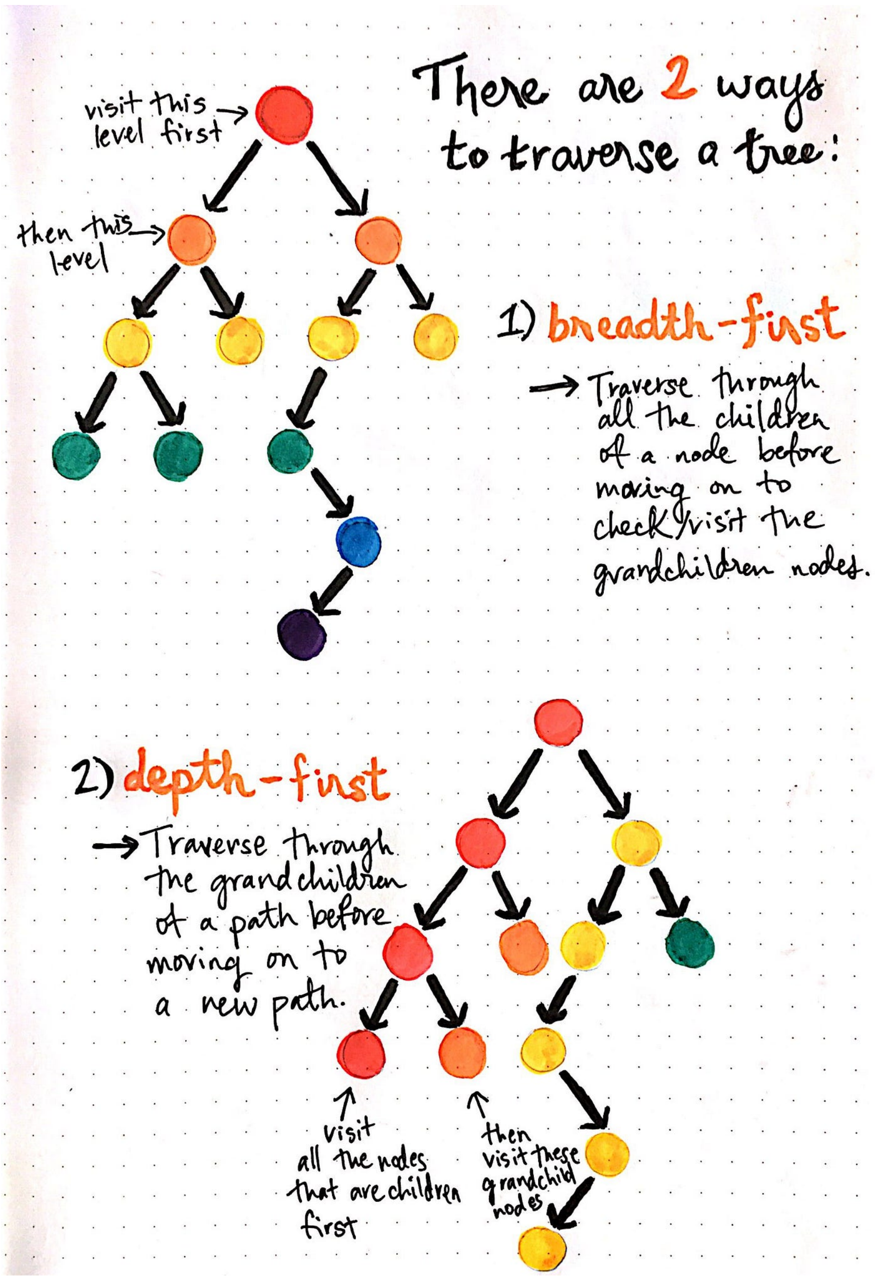
Graph algorithms are critical in solving problems related to graphs, such as finding the shortest path between two nodes or detecting cycles. This section will discuss two graph traversal algorithms: BFS and DFS.

Graph traversal is the process of visiting (checking and/or updating) each vertex in a graph, exactly once. Such traversals are classified by the order in which the vertices are visited. The order may be defined by a specific rule, for example, depth-first search and breadth-first search.

Link: https://medium.com/basecs/breaking-down-breadth-first-search-cebe696709d9


1
2
5
3
4
6
7
Depth-first search
• Traverse through left subtree(s) first, then traverse through the right subtree(s).
1
2
3
4
5
6
7
Breadth-first search
• Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

While DFS uses a stack data structure, BFS leans on the queue data structure.


There are 2 ways to traverse a tree:
visit this level first
then this level
1) breadth-first
→ Traverse through all the children of a node before moving on to check/visit the grandchildren nodes.
2) depth-first
→ Traverse through the grandchildren of a path before moving on to a new path.
visit all the nodes that are children first
then visit these grandchild nodes

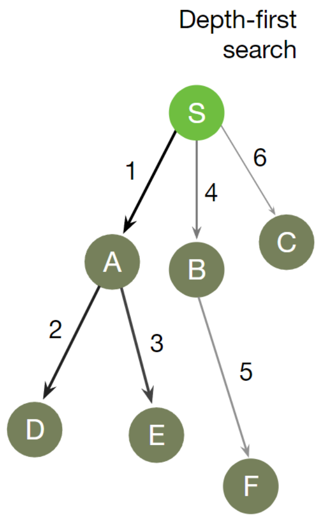
Depth First Search

We know that depth-first search is the process of traversing down through one branch of a tree until we get to a leaf, and then working our way back to the "trunk" of the tree. In other words, implementing a DFS means traversing down through the subtrees of a binary search tree.

DFS is a recursive algorithm that starts at the root node and explores as far as possible along each branch before backtracking.

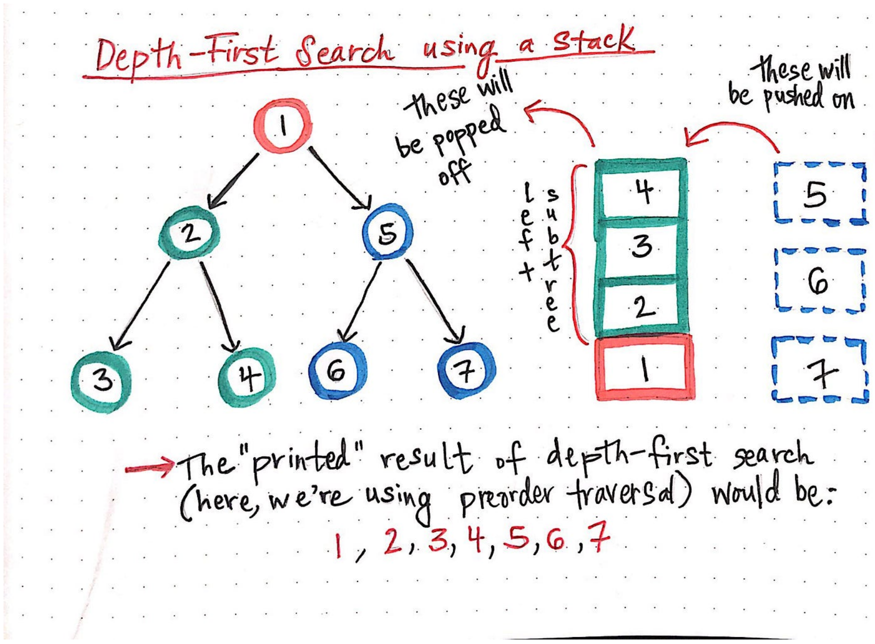
It chooses a node and explores all of its unvisited neighbors, visiting the first neighbor that has not been explored and backtracking only when all the neighbors have been visited. By doing so, it explores the graph by following as deep a path from the starting node as possible before backtracking to explore other branches. This continues until all nodes have been explored.

In depth first, Continue down the edges from the start vertex to the last vertex on that path or until the maximum traversal depth is reached, then walk down the other paths.

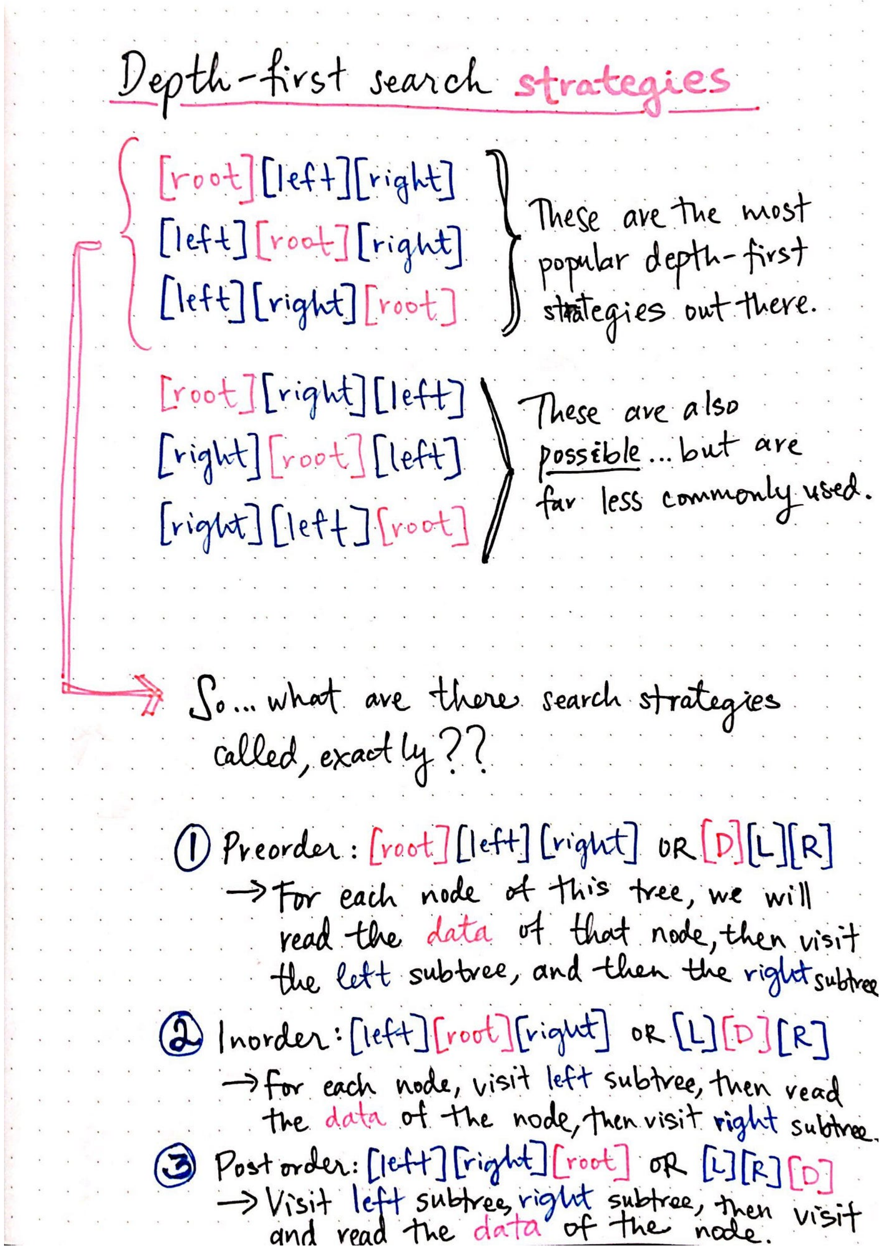


DFS Algorithm goes 'deep' instead of 'wide'

https://miro.medium.com/v2/resize:fit:1400/1*LUL63FWqneOfsLKqMtHKFw.gif


Depth-First Search using a Stack
these will be popped off
these will be pushed on
left subtree
1
2
5
3
4
6
7
The "printed" result of depth-first search (here, we're using preorder traversal) would be:
1, 2, 3, 4, 5, 6, 7

In depth-first search, once we start down a path, we don't stop until we get to the end. In other words, we traverse through one branch of a tree until we get to a leaf, and then we work our way back to the trunk of the tree.



Depth-first search strategies

[root][left][right]
[left][root][right]
[left][right][root]

These are the most popular depth-first strategies out there.

[root][right][left]
[right][root][left]
[right][left][root]

These are also possible... but are far less commonly used.

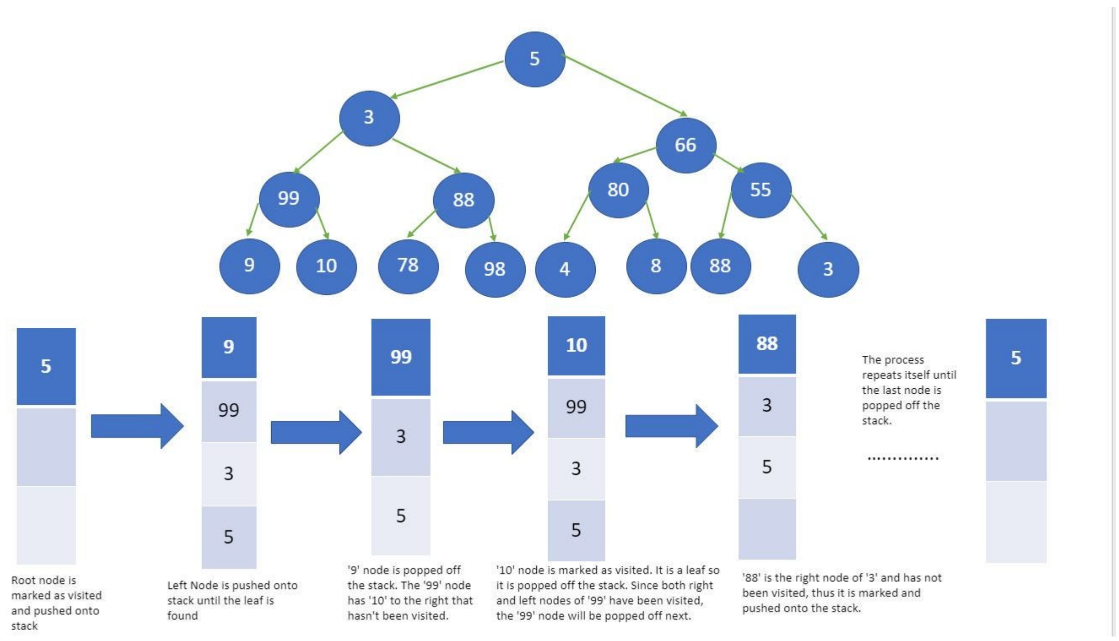
So... what are there search strategies called, exactly??

① Preorder: [root][left][right] OR [D][L][R]
→ For each node of this tree, we will read the data of that node, then visit the left subtree, and then the right subtree

② Inorder: [left][root][right] OR [L][D][R]
→ for each node, visit left subtree, then read the data of the node, then visit right subtree.

③ Post order: [left][right][root] OR [L][R][D]
→ Visit left subtree, right subtree, then visit and read the data of the node.

Stack data structure is used to implement DFS. The algorithm starts with a particular node of a graph, then goes to any of its adjacent nodes and repeats this process until it finds the goal. If it reaches a node from which there is no unexplored edge leading to an unvisited node, then it backtracks to the last visited node and repeats the process.



Root node is marked as visited and pushed onto stack

Left Node is pushed onto stack until the leaf is found

'9' node is popped off the stack. The '99' node has '10' to the right that hasn't been visited.

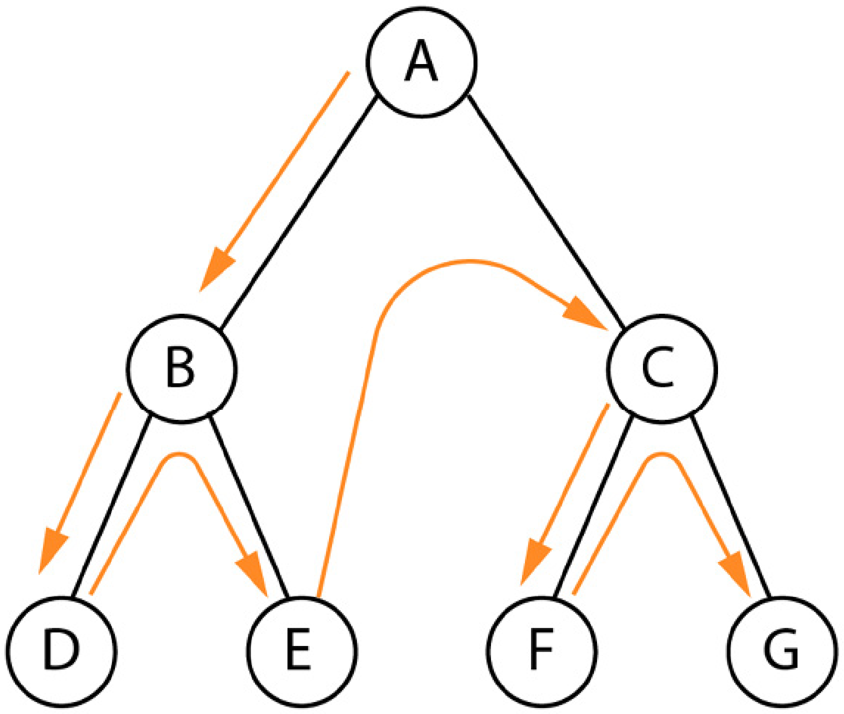
'10' node is marked as visited. It is a leaf so it is popped off the stack. Since both right and left nodes of '99' have been visited, the '99' node will be popped off next.

'88' is the right node of '3' and has not been visited, thus it is marked and pushed onto the stack.

The process repeats itself until the last node is popped off the stack.

```
G = nx.Graph()
G.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G
↪')])

visited = []

def dfs(visited, graph, node):
    if node not in visited:
        visited.append(node)
    # We then iterate through each neighbor of the current node. For each neighbor, we␣
↪recursively call the dfs() function
    # passing in visited, graph, and the neighbor as arguments:
        for neighbor in graph[node]:
            visited = dfs(visited, graph, neighbor)
    # The dfs() function continues to explore the graph depth-first, visiting all the␣
↪neighbors of each node until there
    # are no more unvisited neighbors. Finally, the visited list is returned
    return visited

dfs(visited, G, 'A')
```

```
['A', 'B', 'D', 'E', 'C', 'F', 'G']
```



Once again, the order we obtained is the one we anticipated in Figure. DFS is useful in solving various problems, such as finding connected components, topological sorting, and solving maze problems. It is particularly useful in finding cycles in a graph since it traverses the graph in a depth-first order, and a cycle exists if, and only if, a node is visited twice during the traversal.

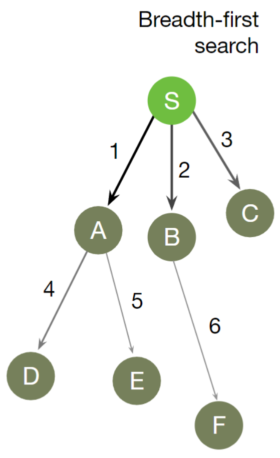
Additionally, many other algorithms in graph theory build upon BFS and DFS, such as Dijkstra's shortest path algorithm, Kruskal's minimum spanning tree algorithm, and Tarjan's strongly connected components algorithm. Therefore, a solid understanding of BFS and DFS is essential for anyone who wants to work with graphs and develop more advanced graph algorithms.

Breadth First Search

Breadth First Search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'[1]), and explores the neighbor nodes first, before moving to the next level neighbors.

It works by maintaining a queue of nodes to visit and marking each visited node as it is added to the queue. The algorithm then dequeues the next node in the queue and explores all its neighbors, adding them to the queue if they haven't been visited yet.

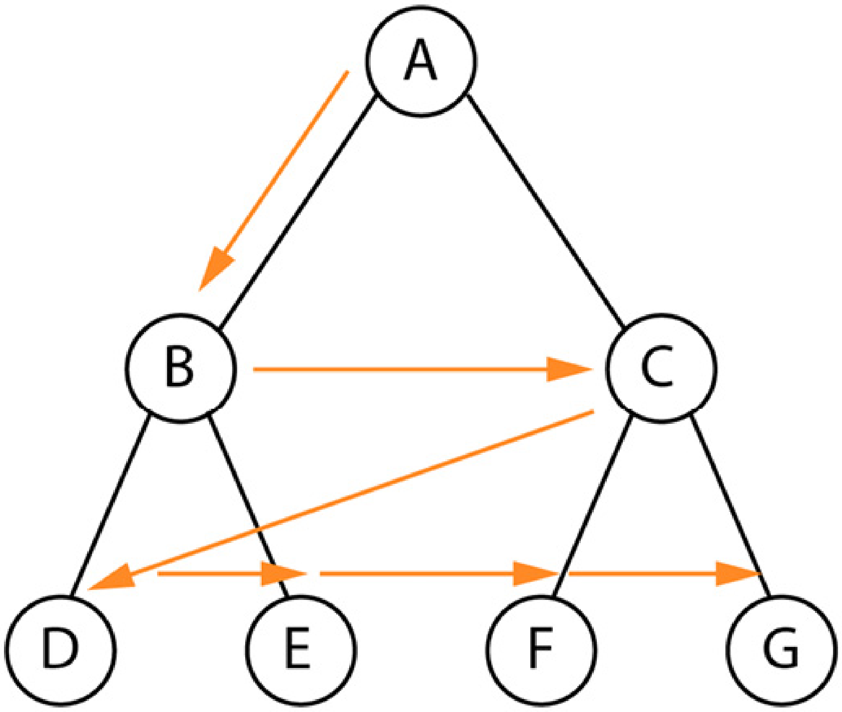
Follow all edges from the start vertex to the next level, then follow all edges of their neighbors by another level and continue this pattern until there are no more edges to follow or the maximum traversal depth is reached.



Let’s now see how we can implement it in Python

```
def bfs(graph, node):
    # We initialize two lists (visited and queue) and add the starting node. The visited␣
↪list keeps track of the nodes that have been
    # visited #during the search, while the queue list stores the nodes that need to be␣
↪visited:

    visited, queue = [node], [node]

    while queue:
    # When We enter a while loop that continues until the queue list is empty.
    # Inside the loop, we remove the first node in the queue list using the pop(0)␣
↪method and store the result in the node variable

        node = queue.pop(0)
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)
    # We iterate through the neighbors of the node using a for loop. For each neighbor␣
↪that has not been visited yet,
    # we add it to the visited list and to the end of the queue list using the append()␣
↪method. When it’s complete,
    # we return the visited list:
    return visited

bfs(G, 'A')
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```



The order we obtained is the one we anticipated in Figure.

BFS is particularly useful in finding the shortest path between two nodes in an unweighted graph. This is because the algorithm visits nodes in order of their distance from the starting node, so the first time the target node is visited, it must be along the shortest path from the starting node.

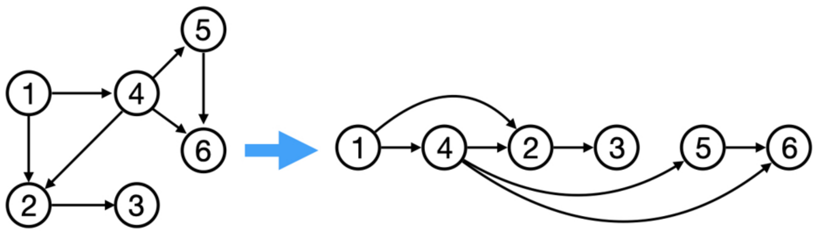
Both algorithms return the same amount of paths if all other traversal options are the same, but the order in which edges are followed and vertices are visited is different.

Depth-first	Breadth-first
$S \rightarrow A$	$S \rightarrow A$
$S \rightarrow A \rightarrow D$	$S \rightarrow B$
$S \rightarrow A \rightarrow E$	$S \rightarrow C$
$S \rightarrow B$	$S \rightarrow A \rightarrow D$
$S \rightarrow B \rightarrow F$	$S \rightarrow A \rightarrow E$
$S \rightarrow C$	$S \rightarrow B \rightarrow F$

Note that there is no particular order in which the edges of a single vertex are followed. Hence, S→C may be returned before S→A and S→B. Shorter paths are returned before longer paths using a breadth-first search still.

Topological Sort

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.



Graph Algorithms

Graph algorithms are used to solve problems that involve graphs. Graph algorithms are used to find the shortest path between two nodes, find the minimum spanning tree, find the strongly connected components, find the shortest path from a single node to all other nodes, find the bridges and articulation points, find the Eulerian path and circuit, find the maximum flow, find the maximum matching, find the biconnected components, find the Hamiltonian path and circuit, find the dominating set, find the shortest path from a single node to all other nodes, find the bridges and articulation points, find the Eulerian path and circuit, find the maximum flow, find the maximum matching, find the biconnected components, find the Hamiltonian path and circuit, find the dominating set, etc.

1.33 Graph Neural Networks

Graph Neural Networks (GNNs) are a class of neural networks that operate on graphs. They are a powerful tool for solving problems in domains such as social network analysis, recommender systems, and combinatorial optimization.

1.33.1 Node Representations

The goal of a GNN is to learn a function that maps a graph to a representation of its nodes. This representation can then be used for various downstream tasks, such as node classification, link prediction, and clustering.

DeepWalk

DeepWalk is a simple algorithm for learning node representations in a graph. It works by performing random walks on the graph, and using the SkipGram model from Word2Vec to learn the embeddings of the nodes.

It introduces important concepts such as embeddings that are at the core of GNNs. Unlike traditional neural networks, the goal of this architecture is to produce representations that are then fed to other models, which perform downstream tasks (for example, node classification).

DeepWalk architecture and its two major components: **Word2Vec** and **random walks**.

Word2Vec

The first step to comprehending the DeepWalk algorithm is to understand its major component: Word2Vec. it proposed a new technique to translate words into vectors (also known as embeddings) using large datasets of text. These representations can then be used in downstream tasks, such as sentiment classification.

One of the most surprising results of Word2Vec is its ability to solve analogies. A popular example is how it can answer the question "man is to woman, what king is to ___?" It can be calculated as follows:

CBOW versus skip-gram

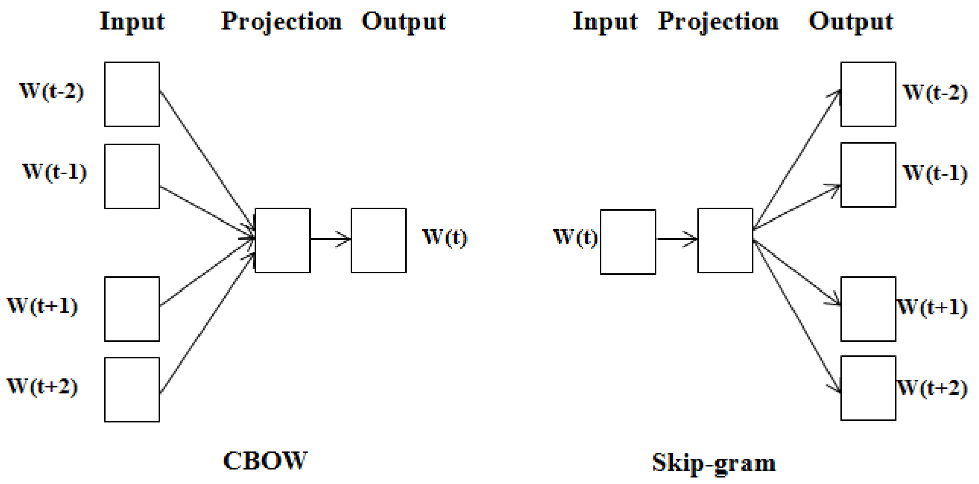
its only goal is to produce high-quality embeddings.

The continuous bag-of-words (CBOW) model:

This is trained to predict a word using its surrounding context (words coming before and after the target word). The order of context words does not matter since their embeddings are summed in the model. The authors claim to obtain better results using four words before and after the one that is predicted.

The continuous skip-gram model:

Here, we feed a single word to the model and try to predict the words around it. Increasing the range of context words leads to better embeddings but also increases the training time.



Creating skip-grams

For now, we will focus on the skip-gram model since it is the architecture used by DeepWalk. Skip-grams are implemented as pairs of words with the following structure (target word, context word). where target word is the input and context word is the word to predict. The number of skip grams for the same target word depends on a parameter called context size.

Context Size	Text	Skip-grams
1	the train was late.	(‘the’, ‘train’)
	the train was late	(’train’, ’the’)
		(’train’, ’was’)
	the train was late	(’was’, ’train’)
		(’was’, ’late’)
	the train was late	(‘late’, ‘was’)
2	the train was late	(’the’, ’train’)
		(’the’, ’was’)
	the train was late	(’train’, ’the’)
		(’train’, ’was’)
		(’train’, ’late’)
	the train was late	(’was’, ’the’)
		(’was’, ’train’)
		(’was’, ’late’)
	the train was late	(’late’, ’train’)
		(’late’, ’was’)

The same idea can be applied to a corpus of text instead of a single sentence.

```
import numpy as np
np.random.seed(0)

CONTEXT_SIZE = 2

text = """Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc eu sem
scelerisque, dictum eros aliquam, accumsan quam. Pellentesque tempus, lorem ut
semper fermentum, ante turpis accumsan ex, sit amet ultricies tortor erat quis
nulla. Nunc consectetur ligula sit amet purus porttitor, vel tempus tortor
scelerisque. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices
posuere cubilia curae; Quisque suscipit ligula nec faucibus accumsan. Duis
vulputate massa sit amet viverra hendrerit. Integer maximus quis sapien id
convallis. Donec elementum placerat ex laoreet gravida. Praesent quis enim
facilisis, bibendum est nec, pharetra ex. Etiam pharetra congue justo, eget
imperdiet diam varius non. Mauris dolor lectus, interdum in laoreet quis,
faucibus vitae velit. Donec lacinia dui eget maximus cursus. Class aptent taciti
sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Vivamus
tincidunt velit eget nisi ornare convallis. Pellentesque habitant morbi
tristique senectus et netus et malesuada fames ac turpis egestas. Donec
tristique ultrices tortor at accumsan.
""".split()

# Create skipgrams
skipgrams = []
for i in range(CONTEXT_SIZE, len(text) - CONTEXT_SIZE):
    array = [text[j] for j in np.arange(i - CONTEXT_SIZE, i + CONTEXT_SIZE + 1) if j !=␣
↪i]
    skipgrams.append((text[i], array))
```

(continues on next page)

(continued from previous page)

```
print(skipgrams[0:2])
```

```
[('dolor', ['Lorem', 'ipsum', 'sit', 'amet,']), ('sit', ['ipsum', 'dolor', 'amet,',
↪'consectetur'])]
```

```
vocab = set(text)
VOCAB_SIZE = len(vocab)
print(f"Length of vocabulary = {VOCAB_SIZE}")
```

```
Length of vocabulary = 121
```

```
from gensim.models.word2vec import Word2Vec

# Create Word2Vec
model = Word2Vec([text],
                 sg=1,   # Skip-gram
                 vector_size=10,
                 min_count=0,
                 window=2,
                 workers=1,
                 seed=0)

print(f'Shape of W_embed: {model.wv.vectors.shape}')

# Train model
model.train([text], total_examples=model.corpus_count, epochs=10)

# Print a word embedding
print('\nWord embedding =')
print(model.wv[0])
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In[3], line 1
----> 1 from gensim.models.word2vec import Word2Vec
      3 # Create Word2Vec
      4 model = Word2Vec([text],
      5                  sg=1,   # Skip-gram
      6                  vector_size=10,
   (...)
      9                  workers=1,
     10                  seed=0)

File ~/checkouts/readthedocs.org/user_builds/ml-math/envs/latest/lib/python3.11/site-
↪packages/gensim/__init__.py:11
      7 __version__ = '4.3.2'
      9 import logging
---> 11 from gensim import parsing, corpora, matutils, interfaces, models, similarities,␣
↪utils  # noqa:F401
     14 logger = logging.getLogger('gensim')
     15 if not logger.handlers:  # To ensure reload() doesn't add another one
```

(continues on next page)

(continued from previous page)

```
File ~/checkouts/readthedocs.org/user_builds/ml-math/envs/latest/lib/python3.11/site-
↪packages/gensim/corpora/__init__.py:6
      1 """
      2 This package contains implementations of various streaming corpus I/O format.
      3 """
      5 # bring corpus classes directly into package namespace, to save some typing
----> 6 from .indexedcorpus import IndexedCorpus  # noqa:F401 must appear before the␣
↪other classes
      8 from .mmcorpus import MmCorpus  # noqa:F401
      9 from .bleicorpus import BleiCorpus  # noqa:F401

File ~/checkouts/readthedocs.org/user_builds/ml-math/envs/latest/lib/python3.11/site-
↪packages/gensim/corpora/indexedcorpus.py:14
     10 import logging
     12 import numpy
---> 14 from gensim import interfaces, utils
     16 logger = logging.getLogger(__name__)
     19 class IndexedCorpus(interfaces.CorpusABC):

File ~/checkouts/readthedocs.org/user_builds/ml-math/envs/latest/lib/python3.11/site-
↪packages/gensim/interfaces.py:19
      7 """Basic interfaces used across the whole Gensim package.
      8
      9 These interfaces are used for building corpora, model transformation and␣
↪similarity queries.
   (...)
     14
     15 """
     17 import logging
---> 19 from gensim import utils, matutils
     22 logger = logging.getLogger(__name__)
     25 class CorpusABC(utils.SaveLoad):

File ~/checkouts/readthedocs.org/user_builds/ml-math/envs/latest/lib/python3.11/site-
↪packages/gensim/matutils.py:20
     18 import scipy.sparse
     19 from scipy.stats import entropy
---> 20 from scipy.linalg import get_blas_funcs, triu
     21 from scipy.linalg.lapack import get_lapack_funcs
     22 from scipy.special import psi  # gamma function utils

ImportError: cannot import name 'triu' from 'scipy.linalg' (/home/docs/checkouts/
↪readthedocs.org/user_builds/ml-math/envs/latest/lib/python3.11/site-packages/scipy/
↪linalg/__init__.py)
```

While this approach works well with small vocabularies, the computational cost of applying a full softmax function to millions of words (the vocabulary size) is too costly in most cases.

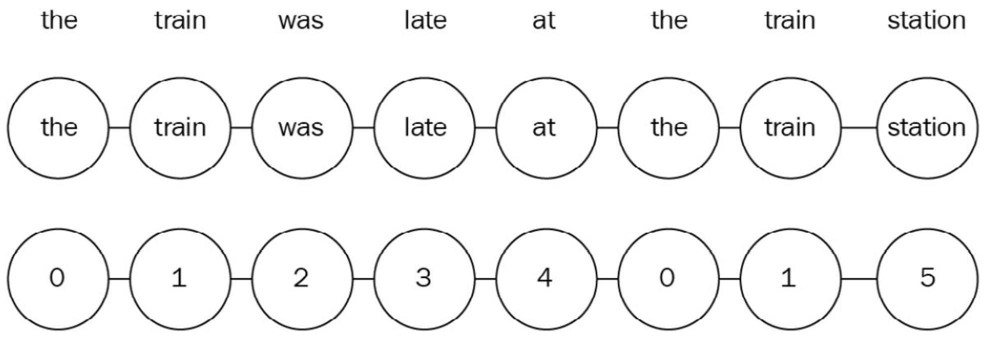
Word2Vec (and DeepWalk) implements one of these techniques, called H-Softmax. Instead of a flat softmax that directly calculates the probability of every word, this technique uses a binary tree structure where leaves are words. Even more interestingly, a Huffman tree can be used, where infrequent words are stored at deeper levels than common words. In most cases, this dramatically speeds up the word prediction by a factor of at least 50.

H-Softmax can be activated in gensim using hs=1.

DeepWalk and random walks

Proposed in 2014 by Perozzi et al., DeepWalk quickly became extremely popular among graph researchers. It is a simple algorithm that can be used to learn node representations in a graph. It works by performing random walks on the graph, and using the SkipGram model from Word2Vec to learn the embeddings of the nodes.

The goal of DeepWalk is to produce high-quality feature representations of nodes in an unsupervised way. This architecture is heavily inspired by Word2Vec in NLP. However, instead of words, our dataset is composed of nodes. This is why we use random walks to generate meaningful sequences of nodes that act like sentences. The following diagram illustrates the connection between sentences and graphs:



Sentences can be represented as graphs

Why are random walks important? Even if nodes are randomly selected, the fact that they often appear together in a sequence means that they are close to each other. Under the network homophily hypothesis, nodes that are close to each other are similar. This is particularly the case in social networks, where people are connected to friends and family.

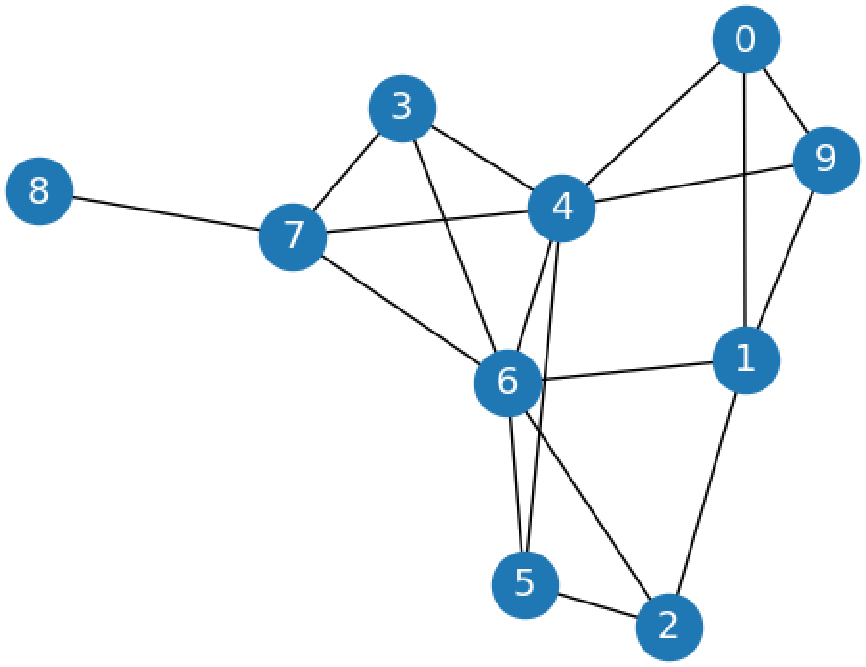
This idea is at the core of the DeepWalk algorithm: when nodes are close to each other, we want to obtain high similarity scores. On the contrary, we want low scores when they are farther apart.

Let's implement a random walk function using a networkx graph:

We generate a random graph thanks to the erdos_renyi_graph function with a fixed number of nodes (10) and a predefined probability of creating an edge between two nodes (0.3):

```
import networkx as nx
import matplotlib.pyplot as plt

# Create a graph
G = nx.erdos_renyi_graph(10, 0.3, seed=1, directed=False)

# Plot graph
plt.figure()
plt.axis('off')
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=0),
                 node_size=600,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white'
                 )
```

```
/Users/ikram.ali/miniconda3/envs/ml_notes/lib/python3.11/site-packages/networkx/drawing/
↪nx_pylab.py:433: UserWarning: No data for colormapping provided via 'c'. Parameters
↪'cmap' will be ignored
  node_collection = ax.scatter(
```



Let's implement random walks with a simple function. This function takes two parameters: the starting node (start) and the length of the walk (length). At every step, we randomly select a neighboring node (using np.random.choice) until the walk is complete:

```
def random_walk(start, length):
    walk = [str(start)]  # starting node
    for i in range(length):
        neighbors = [node for node in G.neighbors(start)]
        next_node = np.random.choice(neighbors, 1)[0]
        walk.append(str(next_node))
        start = next_node
    return walk
```

```
print(random_walk(0, 10))
```

```
['0', '1', '9', '1', '0', '4', '6', '7', '6', '5', '6']
```

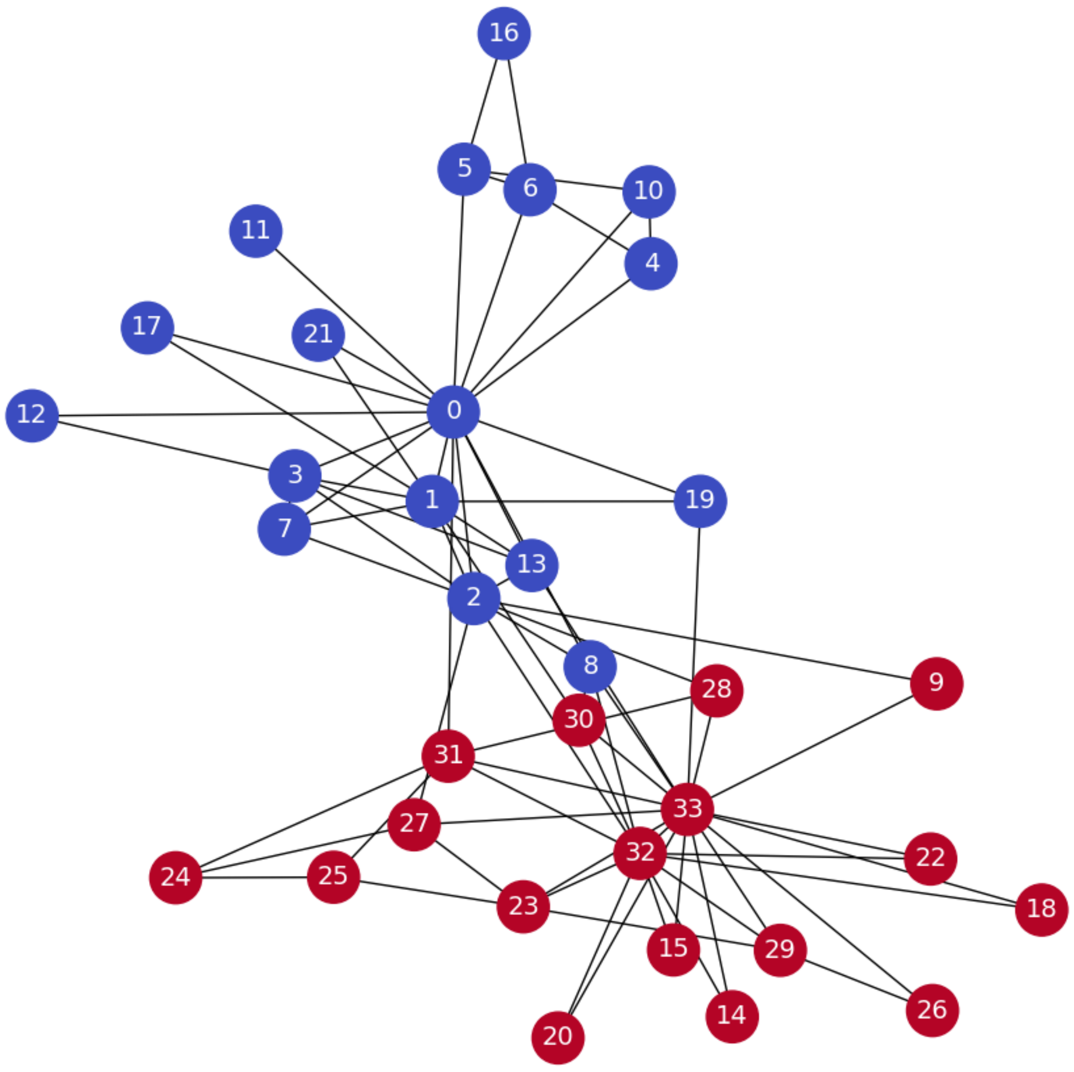
We can see that certain nodes, such as 0 and 9, are often found together. Considering that it is a homophilic graph, it means that they are similar. It is precisely the type of relationship we're trying to capture with DeepWalk.

Now that we have implemented Word2Vec and random walks separately, let's combine them to create DeepWalk.

Implementing DeepWalk

The dataset we will use is Zachary's Karate Club. It simply represents the relationships within a karate club studied by Wayne W. Zachary in the 1970s. It is a kind of social network where every node is a member, and members who interact outside the club are connected.

```python
G = nx.karate_club_graph()

# Process labels (Mr. Hi = 0, Officer = 1)
labels = []
for node in G.nodes:
    label = G.nodes[node]['club']
    labels.append(1 if label == 'Officer' else 0)

# Plot graph
plt.figure(figsize=(12,12))
plt.axis('off')
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=0),
                 node_color=labels,
                 node_size=800,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white'
                 )
```



The next step is to generate our dataset, the random walks. We want to be as exhaustive as possible, which is why we will create 80 random walks of a length of 10 for every node in the graph

```
walks = []
for node in G.nodes:
    for _ in range(80):
        walks.append(random_walk(node, 10))
```

```
walks[:10]
```

```
[['0', '10', '0', '17', '0', '2', '13', '0', '2', '9', '33'],
 ['0', '19', '0', '11', '0', '5', '16', '5', '16', '6', '5'],
 ['0', '31', '0', '3', '7', '0', '2', '7', '1', '7', '3'],
```

(continues on next page)

(continued from previous page)

```
 ['0', '21', '1', '0', '2', '32', '30', '33', '8', '30', '1'],
 ['0', '8', '32', '33', '13', '0', '1', '13', '0', '5', '6'],
 ['0', '31', '32', '30', '32', '31', '0', '2', '1', '21', '1'],
 ['0', '11', '0', '4', '10', '5', '10', '0', '21', '1', '17'],
 ['0', '17', '1', '13', '3', '12', '0', '5', '16', '6', '0'],
 ['0', '4', '0', '5', '0', '10', '5', '16', '5', '16', '5'],
 ['0', '8', '0', '2', '9', '33', '14', '32', '20', '32', '2']]
```

The final step consists of implementing Word2Vec. Here, we use the skip-gram model previously seen with H-Softmax. You can play with the other parameters to improve the quality of the embeddings:

```
model = Word2Vec(walks,
                 hs=1,   # Hierarchical softmax
                 sg=1,   # Skip-gram
                 vector_size=100,
                 window=10,
                 workers=1,
                 seed=1)

print(f'Shape of embedding matrix: {model.wv.vectors.shape}')

# Build vocabulary
model.build_vocab(walks)

# Train model
model.train(walks, total_examples=model.corpus_count, epochs=30, report_delay=1)
```

```
Shape of embedding matrix: (34, 100)
```

```
(186095, 897600)
```

```
# Most similar nodes
print('Nodes that are the most similar to node 0:')
for similarity in model.wv.most_similar(positive=['0']):
    print(f'   {similarity}')

# Similarity between two nodes
print(f"\nSimilarity between node 0 and 4: {model.wv.similarity('0', '4')}")
```

```
Nodes that are the most similar to node 0:
   ('7', 0.6418750882148743)
   ('11', 0.6362574696540833)
   ('10', 0.6352985501289368)
   ('4', 0.6283851265907288)
   ('1', 0.624032199382782)
   ('17', 0.6081531047821045)
   ('6', 0.5763437151908875)
   ('5', 0.5598757266998291)
   ('21', 0.557222843170166)
   ('16', 0.5503911972045898)
```

(continues on next page)

(continued from previous page)

```
Similarity between node 0 and 4: 0.628385066986084
```

```
# from sklearn.manifold import TSNE
```

```
# nodes_wv = np.array([model.wv.get_vector(str(i)) for i in range(len(model.wv))])
# labels = np.array(labels)
```

```
# tsne = TSNE(n_components=2,
#             learning_rate='auto',
#             init='pca',
#             random_state=0).fit_transform(nodes_wv)
```

```
# plt.figure(figsize=(6, 6), dpi=300)
# plt.scatter(tsne[:, 0], tsne[:, 1], s=100, c=labels, cmap="coolwarm")
# plt.show()
```

This plot is quite encouraging since we can see a clear line that separates the two classes. It should be possible for a simple ML algorithm to classify these nodes with enough examples (training data). Let's implement a classifier and train it on our node embeddings

Our model obtains an accuracy score of 95.45%, which is pretty good considering the unfavorable train/test split we gave it. There is still room for improvement, but this example showed two useful applications of DeepWalk:

- Discovering similarities between nodes using embeddings and cosine similarity (unsupervised learning)
- Using these embeddings as a dataset for a supervised task such as node classification

```
# from sklearn.ensemble import RandomForestClassifier
# from sklearn.metrics import accuracy_score

# # Create masks to train and test the model
# train_mask = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
# test_mask = [0, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 26, 27, 28, 29, 30, 31,␣
↪32, 33]

# # Train classifier
# clf = RandomForestClassifier(random_state=0)
# clf.fit(nodes_wv[train_mask], labels[train_mask])

# # Evaluate accuracy
# y_pred = clf.predict(nodes_wv[test_mask])
# acc = accuracy_score(y_pred, labels[test_mask])
# print(f'Accuracy = {acc*100:.2f}%')
```

Node2vec

Node2Vec is an architecture largely based on DeepWalk. DeepWalk the two main components of this architecture: random walks and Word2Vec. How can we improve the quality of our embeddings?

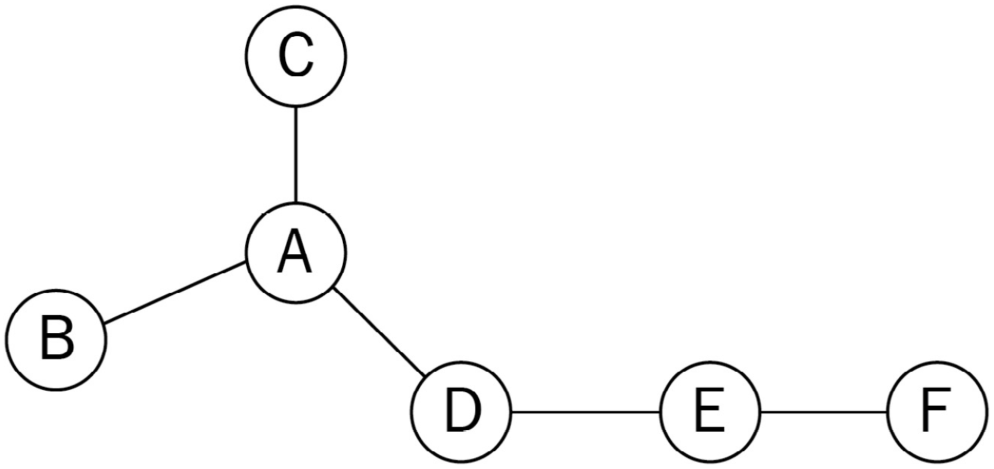
Node2Vec is an algorithmic framework for representational learning on graphs. Given any graph, it can learn continuous feature representations for the nodes, which can then be used for various downstream machine learning tasks. For example, we can use these embeddings as features for node classification, link prediction, clustering, and visualization.

Node2Vec was introduced in 2016 by Grover and Leskovec from Stanford University. It keeps the same two main components from DeepWalk: random walks and Word2Vec. The difference is that instead of obtaining sequences of nodes with a uniform distribution, the random walks are carefully biased in Node2Vec. We will see why these biased random walks perform better and how to implement them in the two following sections:

- Defining a neighborhood
- Introducing biases in random walks

Defining a neighborhood

How do you define the neighborhood of a node? what does "close" mean in the context of a graph?



We want to explore three nodes in the neighborhood of node A. This exploration process is also called a sampling strategy:

- A possible solution would be to consider the three closest nodes in terms of connections. In this case, the neighborhood of A, noted $N(A)$, would $N(A) = \{B, C, D\}$$
- Another possible sampling strategy consists of selecting nodes that are not adjacent to previous nodes first. In our example, the neighborhood of A would be $N(A) = \{D, \boldsymbol{F}, \boldsymbol{F}\}$

In other words, we want to implement a Breadth-First Search (BFS) in the first case and a Depth-First Search (DFS) in the second one. You can find more information about these algorithms and implementations in Chapter 2, Graph Theory for Graph Neural Networks.

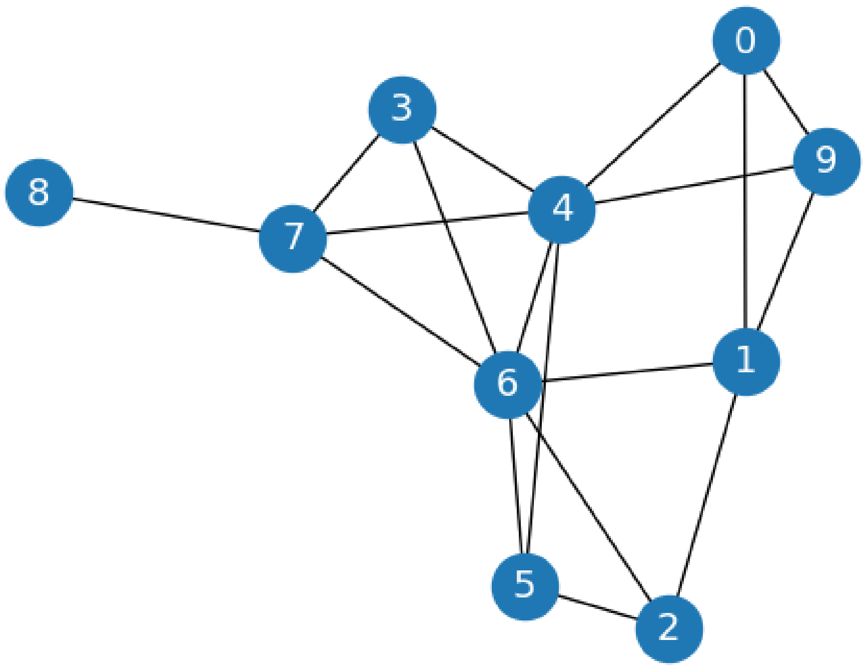
The best way to understand this is to actually implement this architecture and play with the parameters.

```python
import networkx as nx
import matplotlib.pyplot as plt
import random
random.seed(0)
```

(continues on next page)

(continued from previous page)

```python
import numpy as np
np.random.seed(0)

# Create graph
G = nx.erdos_renyi_graph(10, 0.3, seed=1, directed=False)

# Plot graph
plt.figure()
plt.axis('off')
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=0),
                 node_size=600,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white'
                 )
```



```python
def next_node(previous, current, p, q):
    alphas = []

    # Get the neighboring nodes
    neighbors = list(G.neighbors(current))

    # Calculate the appropriate alpha value for each neighbor
    for neighbor in neighbors:
        # Distance = 0: probability to return to the previous node
        if neighbor == previous:
            alpha = 1/p
```

(continues on next page)

(continued from previous page)

```
            # Distance = 1: probability of visiting a local node
            elif G.has_edge(neighbor, previous):
                alpha = 1
            # Distance = 2: probability to explore an unknown node
            else:
                alpha = 1/q
            alphas.append(alpha)

    # Normalize the alpha values to create transition probabilities
    probs = [alpha / sum(alphas) for alpha in alphas]

    # Randomly select the new node based on the transition probabilities
    next = np.random.choice(neighbors, size=1, p=probs)[0]
    return next
```

```
def random_walk(start, length, p, q):
    walk = [start]

    for i in range(length):
        current = walk[-1]
        previous = walk[-2] if len(walk) > 1 else None
        next = next_node(previous, current, p, q)
        walk.append(next)

    return walk
```

```
random_walk(0, 8, p=1, q=1)
```

```
[0, 4, 7, 6, 4, 5, 4, 5, 6]
```

```
random_walk(0, 8, p=1, q=10)
```

```
[0, 9, 1, 9, 1, 9, 1, 0, 1]
```

Implementing Node2Vec

Now that we have the functions to generate biased random walks, the implementation of Node2Vec is very similar to implementing DeepWalk.

```
from gensim.models.word2vec import Word2Vec
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load dataset
G = nx.karate_club_graph()

# Process labels (Mr. Hi = 0, Officer = 1)
labels = []
for node in G.nodes:
```

(continues on next page)

(continued from previous page)

```
    label = G.nodes[node]['club']
    labels.append(1 if label == 'Officer' else 0)

# Create a list of random walks
walks = []
for node in G.nodes:
    for _ in range(80):
        walks.append(random_walk(node, 10, 3, 2))

# Create and train Word2Vec for DeepWalk
node2vec = Word2Vec(walks,
                hs=1,   # Hierarchical softmax
                sg=1,   # Skip-gram
                vector_size=100,
                window=10,
                workers=2,
                min_count=1,
                seed=0)
node2vec.train(walks, total_examples=node2vec.corpus_count, epochs=30, report_delay=1)

# Create masks to train and test the model
train_mask = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
test_mask = [0, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 26, 27, 28, 29, 30, 31,␣
↪32, 33]
labels = np.array(labels)

# Train Node2Vec classifier
clf = RandomForestClassifier(random_state=0)
clf.fit(node2vec.wv[train_mask], labels[train_mask])

# Evaluate accuracy
y_pred = clf.predict(node2vec.wv[test_mask])
acc = accuracy_score(y_pred, labels[test_mask])
print(f'Node2Vec accuracy = {acc*100:.2f}%')
```

```
Node2Vec accuracy = 100.00%
```

Building a movie RecSys with Node2Vec

One of the most popular applications of GNNs is RecSys. If you think about the foundation of Word2Vec (and, thus, DeepWalk and Node2Vec), the goal is to produce vectors with the ability to measure their similarity. Encode movies instead of words, and you can suddenly ask for movies that are the most similar to a given input title. It sounds a lot like a RecSys, right?

Another approach is to look at user ratings. There are different techniques to build a graph based on ratings: bipartite graphs, edges based on pointwise mutual information, and so on. In this section, we'll implement a simple and intuitive approach: movies that are liked by the same users are connected. We'll then use this graph to learn movie embeddings using Node2Vec:

```
# from io import BytesIO
# from urllib.request import urlopen
```

(continues on next page)

(continued from previous page)

```
# from zipfile import ZipFile

# url = 'https://files.grouplens.org/datasets/movielens/ml-100k.zip'
# with urlopen(url) as zurl:
#     with ZipFile(BytesIO(zurl.read())) as zfile:
#         zfile.extractall('.')
```

```
# import pandas as pd

# ratings = pd.read_csv('ml-100k/u.data', sep='\t', names=['user_id', 'movie_id', 'rating',
↪'unix_timestamp'])
# ratings
```

```
# movies = pd.read_csv('ml-100k/u.item', sep='|', usecols=range(2), names=['movie_id', 'title
↪'], encoding='latin-1')
```

```
# Here, we want to see movies that have been liked by the same users.
# This means that ratings such as 1, 2, and 3 are not very relevant.
#  We can discard those and only keep scores of 4 and 5

# ratings = ratings[ratings.rating >= 4]
# ratings
```

We now have 48,580 ratings made by 610 users. The next step is to count every time that two movies are liked by the same user. We will repeat this process for every user in the dataset.

To simplify things, we will use a defaultdict data structure, which automatically creates missing entries instead of raising an error. We'll use this structure to count movies that are liked together:

```
# from collections import defaultdict
# pairs = defaultdict(int)
```

```
# # Loop through the entire list of users
# for group in ratings.groupby("user_id"):
#     # List of IDs of movies rated by the current user
#     user_movies = list(group[1]["movie_id"])

#     # Count every time two movies are seen together
#     for i in range(len(user_movies)):
#         for j in range(i+1, len(user_movies)):
#             pairs[(user_movies[i], user_movies[j])] += 1
```

```
# For each pair of movies in our pairs structure, we unpack the two movies and their␣
↪corresponding score:

# Create a networkx graph
# G = nx.Graph()

# # Try to create an edge between movies that are liked together
# for pair in pairs:
#     if not isinstance(pair, tuple):
```

(continues on next page)

(continued from previous page)

```
#         continue
#     movie1, movie2 = pair
#     score = pairs[pair]

#     # The edge is only created when the score is high enough
#     if score >= 20:
#         G.add_edge(movie1, movie2, weight=score)

# print("Total number of graph nodes:", G.number_of_nodes())
# print("Total number of graph edges:", G.number_of_edges())
```

```
# !pip install node2vec
```

```
# from node2vec import Node2Vec

# node2vec = Node2Vec(G, dimensions=64, walk_length=20, num_walks=200, p=2, q=1,␣
↪workers=1)

# model = node2vec.fit(window=10, min_count=1, batch_words=4)
```

```
# def recommend(movie):
#     movie_id = str(movies[movies.title == movie].movie_id.values[0])
#     for id in model.wv.most_similar(movie_id)[:5]:
#         title = movies[movies.movie_id == int(id[0])].title.values[0]
#         print(f'{title}: {id[1]:.2f}')

# recommend('Star Wars (1977)')
```

Vanilla Neural Networks

However, graph datasets tend to be richer than a mere set of connections: nodes and edges can also have features to represent scores, colors, words, and so on. Including this additional information in our input data is essential to produce the best embeddings possible.

The Cora dataset

Introduced by Sen et al. in 2008 [1], Cora (no license) is the most popular dataset for node classification in the scientific literature. It represents a network of 2,708 publications, where each connection is a reference. Each publication is described as a binary vector of 1,433 unique words, where 0 and 1 indicate the absence or presence of the corresponding word, respectively. This representation is also called a binary bag of words in natural language processing. Our goal is to classify each node into one of seven categories.

```
# !pip install torch_geometric
```

```
# from torch_geometric.datasets import Planetoid
```

```
# dataset = Planetoid(root=".", name="Cora")
# data = dataset[0]
```

```
# print(f'Dataset: {dataset}')
# print('---------------')
# print(f'Number of graphs: {len(dataset)}')
# print(f'Number of nodes: {data.x.shape[0]}')
# print(f'Number of features: {dataset.num_features}')
# print(f'Number of classes: {dataset.num_classes}')
```

```
# print(f'Graph:')
# print('------')
# print(f'Edges are directed: {data.is_directed()}')
# print(f'Graph has isolated nodes: {data.has_isolated_nodes()}')
# print(f'Graph has loops: {data.has_self_loops()}')
```

```
# from torch_geometric.datasets import FacebookPagePage
```

```
# dataset = FacebookPagePage(root=".")
```

```
# data = dataset[0]
```

```
# print(f'Dataset: {dataset}')
# print('-----------------------')
# print(f'Number of graphs: {len(dataset)}')
# print(f'Number of nodes: {data.x.shape[0]}')
# print(f'Number of features: {dataset.num_features}')
# print(f'Number of classes: {dataset.num_classes}')
```

```
# print(f'\nGraph:')
# print('------')
# print(f'Edges are directed: {data.is_directed()}')
# print(f'Graph has isolated nodes: {data.has_isolated_nodes()}')
# print(f'Graph has loops: {data.has_self_loops()}')
```

```
# data.train_mask = range(18000)
# data.val_mask = range(18001, 20000)
# data.test_mask = range(20001, 22470)
```

```
# import pandas as pd

# dataset = Planetoid(root=".", name="Cora")
# data = dataset[0]

# df_x = pd.DataFrame(data.x.numpy())
# df_x['label'] = pd.DataFrame(data.y)
# df_x
```

Classifying nodes with vanilla neural networks

Compared to Zachary's Karate Club, these two datasets include a new type of information: node features. They provide additional information about the nodes in a graph, such as a user's age, gender, or interests in a social network. In a vanilla neural network (also called multilayer perceptron), these embeddings are directly used in the model to perform downstream tasks such as node classification.

we will consider node features as a regular tabular dataset. We will train a simple neural network on this dataset to classify our nodes. Note that this architecture does not take into account the topology of the network. We will try to fix this issue in the next section and compare our results.

The tabular dataset of node features can be easily accessed through the data object we created.

If you're familiar with machine learning, you probably recognize a typical dataset with data and labels. We can develop a simple Multilayer Perceptron (MLP) and train it on data.x with the labels provided by data.y.

```
# import torch
# torch.manual_seed(0)
# from torch.nn import Linear
# import torch.nn.functional as F


# def accuracy(y_pred, y_true):
#     """Calculate accuracy."""
#     return torch.sum(y_pred == y_true) / len(y_true)


# class MLP(torch.nn.Module):
#     """Multilayer Perceptron"""
#     def __init__(self, dim_in, dim_h, dim_out):
#         super().__init__()
#         self.linear1 = Linear(dim_in, dim_h)
#         self.linear2 = Linear(dim_h, dim_out)

#     def forward(self, x):
#         x = self.linear1(x)
#         x = torch.relu(x)
#         x = self.linear2(x)
#         return F.log_softmax(x, dim=1)

#     def fit(self, data, epochs):
#         criterion = torch.nn.CrossEntropyLoss()
#         optimizer = torch.optim.Adam(self.parameters(),
#                                           lr=0.01,
#                                           weight_decay=5e-4)

#         self.train()
#         for epoch in range(epochs+1):
#             optimizer.zero_grad()
#             out = self(data.x)
#             loss = criterion(out[data.train_mask], data.y[data.train_mask])
#             acc = accuracy(out[data.train_mask].argmax(dim=1),
#                           data.y[data.train_mask])
#             loss.backward()
#             optimizer.step()

#             if(epoch % 20 == 0):
#                 val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
#                 val_acc = accuracy(out[data.val_mask].argmax(dim=1),
```

(continues on next page)

(continued from previous page)

```
#                                   data.y[data.val_mask])
#                 print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} | Train Acc:'
#                       f' {acc*100:>5.2f}% | Val Loss: {val_loss:.2f} | '
#                       f'Val Acc: {val_acc*100:.2f}%')

#     @torch.no_grad()
#     def test(self, data):
#         self.eval()
#         out = self(data.x)
#         acc = accuracy(out.argmax(dim=1)[data.test_mask], data.y[data.test_mask])
#         return acc

# # Create MLP model
# mlp = MLP(dataset.num_features, 16, dataset.num_classes)
# print(mlp)

# # Train
# mlp.fit(data, epochs=100)

# # Test
# acc = mlp.test(data)
# print(f'\nMLP test accuracy: {acc*100:.2f}%')
```

Vanilla graph neural networks

Instead of directly introducing well-known GNN architectures, let's try to build our own model to understand the thought process behind GNNs.

A basic neural network layer corresponds to a linear transformation $h_A = x_A W^T x_{A_{\text{is the input vector of node}}} A_{\text{and}}$ $\mathbf{V}$ is the weight matrix. In PyTorch, this equation can be implemented with the torch .mm () function, or with the nn. Linear class that adds other parameters such as biases.

With our graph datasets, the input vectors are node features. It means that nodes are completely separate from each other. This is not enough to capture a good understanding of the graph: like a pixel in an image, the context of a node is essential to understand it. If you look at a group of pixels instead of a single one, you can recognize edges, patterns, and so on. Likewise, to understand a node, you need to look at its neighborhood.

Lers call N_A the set of neighbors of node A. Our graph linear layer can be written as follows: \$$h_A = \sum_{i \in \mathcal{N}_A} x_i W^T$\$ You can imagine several variants of this equation. For instance, we could have a weight matrix the neighbors. Note that we cannot have a weight matrix per neighbor, as this number can change from node to node.

We're talking about neural networks, so we can't apply the previous equation to each node. Instead, we perform matrix multiplications that are much more efficient. For instance, the equation of the linear layer can be rewritten as \$$\boldsymbol{H} = \boldsymbol{YT}$, where $\boldsymbol{X}$ is the input matrix. $In our case, the adjacency matrix$\boldsymbol{A}$contains the connections between every node in$
$\tilde{A}^T X W^T$\$ Let's test this layer by implementing it in PyTorch Geometric. We'll then be able to use it as a regular layer to build a GNN:

```
# class VanillaGNNLayer(torch.nn.Module):
#     def __init__(self, dim_in, dim_out):
#         super().__init__()
#         self.linear = Linear(dim_in, dim_out, bias=False)

#     def forward(self, x, adjacency):
```

(continues on next page)

(continued from previous page)

```
#         x = self.linear(x)
#         # We perform two operations - the linear transformation,
#         #  and then the multiplication with the adjacency matrix
#         x = torch.sparse.mm(adjacency, x)
#         return x
```

Before we can create our vanilla GNN, we need to convert the edge index from our dataset (data.edge_index) in coordinate format to a dense adjacency matrix. We also need to include self loops; otherwise, the central nodes won't be taken into account in their own embeddings.

```
# from torch_geometric.utils import to_dense_adj
# adjacency = to_dense_adj(data.edge_index)[0]
# adjacency += torch.eye(len(adjacency))
# adjacency
```

```
# class VanillaGNN(torch.nn.Module):
#     def __init__(self, dim_in, dim_h, dim_out):
#         super().__init__()
#         self.gnn1 = VanillaGNNLayer(dim_in, dim_h)
#         self.gnn2 = VanillaGNNLayer(dim_h, dim_out)

#     # We perform the same operations with our new layers,
#     #  which take the adjacency matrix we previously calculated as an additional input:
#     def forward(self, x, adjacency):
#         h = self.gnn1(x, adjacency)
#         h = torch.relu(h)
#         h = self.gnn2(h, adjacency)
#         return F.log_softmax(h, dim=1)

#     def fit(self, data, epochs):
#         criterion = torch.nn.CrossEntropyLoss()
#         optimizer = torch.optim.Adam(self.parameters(),
#                                       lr=0.01,
#                                       weight_decay=5e-4)

#         self.train()
#         for epoch in range(epochs+1):
#             optimizer.zero_grad()
#             out = self(data.x, adjacency)
#             loss = criterion(out[data.train_mask], data.y[data.train_mask])
#             acc = accuracy(out[data.train_mask].argmax(dim=1),
#                           data.y[data.train_mask])
#             loss.backward()
#             optimizer.step()

#             if(epoch % 20 == 0):
#                 val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
#                 val_acc = accuracy(out[data.val_mask].argmax(dim=1),
#                                   data.y[data.val_mask])
#                 print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} | Train Acc:'
#                       f' {acc*100:>5.2f}% | Val Loss: {val_loss:.2f} | '
#                       f'Val Acc: {val_acc*100:.2f}%')
```

(continues on next page)

(continued from previous page)

```
#     torch.no_grad()
#     def test(self, data):
#         self.eval()
#         out = self(data.x, adjacency)
#         acc = accuracy(out.argmax(dim=1)[data.test_mask], data.y[data.test_mask])
#         return acc


# # Create the Vanilla GNN model
# gnn = VanillaGNN(dataset.num_features, 16, dataset.num_classes)
# print(gnn)
# # Train
# gnn.fit(data, epochs=100)
# # Test
# acc = gnn.test(data)
# print(f'\nGNN test accuracy: {acc*100:.2f}%')
```

1.33.2 Graph Convolutional Networks

The Graph Convolutional Network (GCN) architecture is the blueprint of what a GNN looks like.

In this chapter, we'll talk about the limitations of our previous vanilla GNN layer. This will help us to understand the motivation behind GCNs. We'll detail how the GCN layer works and why it performs better than our solution.

The last section is dedicated to a new task: node regression. This is not a very common task when it comes to GNNs, but it is particularly useful when you're working with tabular data.

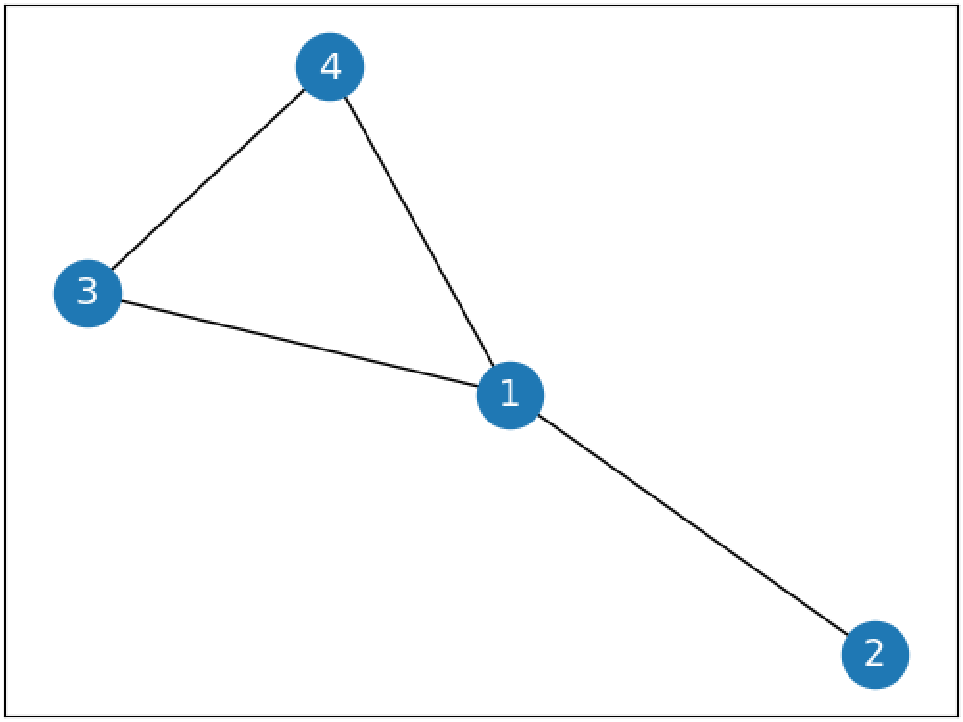
Designing the graph convolutional layer

Unlike tabular or image data, nodes do not always have the same number of neighbors. However, if we look at our GNN layer, we don't take into account this difference in the number of neighbors. Our layer consists of a simple sum without any normalization coefficient.

Here is how we calculated the embedding of a node i,

$$h_i = \sum_{j \in \mathcal{N}_i} x_j W^T$$

```
G = nx.Graph()
G.add_edges_from([(1, 2), (1, 3), (1, 4), (3,4)])

nx.draw_networkx(G, pos=nx.spring_layout(G, seed=0), node_size=600, cmap='coolwarm',␣
↪font_size=14, font_color='white')
```



Imagine that node 1 has 1,000 neighbors and node 2 only has 1: the embedding will have much larger values than . This is an issue because we want to compare these embeddings. How are we supposed to make meaningful comparisons when their values are so vastly different?

Fortunately, there is a simple solution: dividing the embedding by the number of neighbors.

Let's write the degree of node deg(A) . Here is the new formula for the GNN layer

$$h_i = \frac{1}{\deg(i)} \sum_{j \in \mathcal{N}_i} x_j W^T$$

But how do we translate it into a matrix multiplication? As a reminder, this was what we obtained for our vanilla GNN layer:

$$H = \tilde{A}^T X W^T$$

$$\text{Here, } \tilde{A} = A + I$$

The only thing that is missing from this formula is a matrix to give us the normalization coefficient, $\overline{\deg(i)}$. This is something that can be obtained thanks to the degree matrix $\boldsymbol{D}$, which counts the number of neighbors for each node. Here

Here is the same matrix in numpy

```
import numpy as np
D = np.array([
    [3, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 2, 0],
    [0, 0, 0, 2]
])

np.linalg.inv(D)
```

```
array([[0.33333333, 0.        , 0.        , 0.        ],
       [0.        , 1.        , 0.        , 0.        ],
       [0.        , 0.        , 0.5       , 0.        ],
       [0.        , 0.        , 0.        , 0.5       ]])
```

This is exactly what we were looking for. To be even more accurate, we added self-loops to the graph, A = A + 1

NumPy has a specific function, numpy.identity(n), to quickly create an identity matrix l of n dimensions

```
np.linalg.inv(D + np.identity(4))
```

```
array([[0.25      , 0.        , 0.        , 0.        ],
       [0.        , 0.5       , 0.        , 0.        ],
       [0.        , 0.        , 0.33333333, 0.        ],
       [0.        , 0.        , 0.        , 0.33333333]])
```

Comparing graph convolutional and graph linear layers

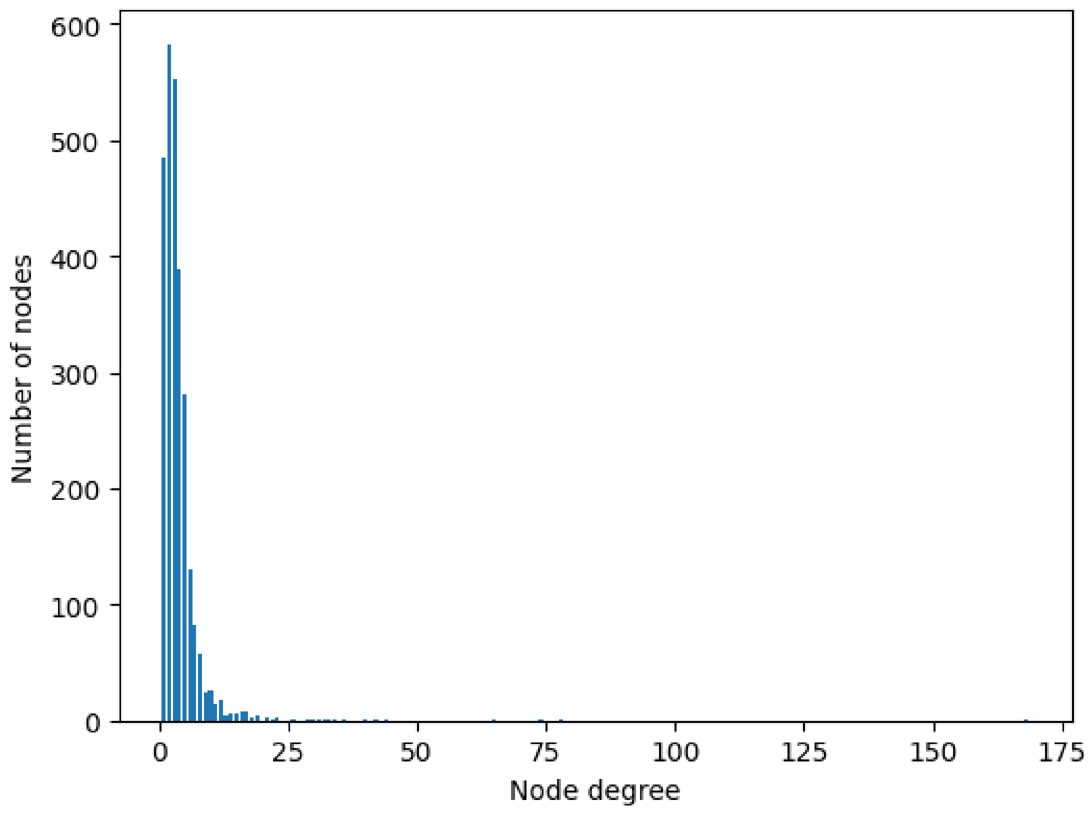
In the previous chapter, our vanilla GNN outperformed the Node2Vec model, but how does it compare to a GCN? In this section, we will compare their performance on the Cora and Facebook Page-Page datasets.

Compared to the vanilla GNN, the main feature of the GCN is that it considers node degrees to weigh its features. Before the real implementation, let's analyze the node degrees in both datasets. This information is relevant since it is directly linked to the performance of the GCN.

```
# from torch_geometric.datasets import Planetoid
# from torch_geometric.utils import degree
# from collections import Counter
# import matplotlib.pyplot as plt
```

```
# dataset = Planetoid(root=".", name="Cora")
# data = dataset[0]

# # Get list of degrees for each node
# degrees = degree(data.edge_index[0]).numpy()

# # Count the number of nodes for each degree
# numbers = Counter(degrees)

# # Bar plot
# fig, ax = plt.subplots()
# ax.set_xlabel('Node degree')
# ax.set_ylabel('Number of nodes')
# plt.bar(numbers.keys(), numbers.values())
```

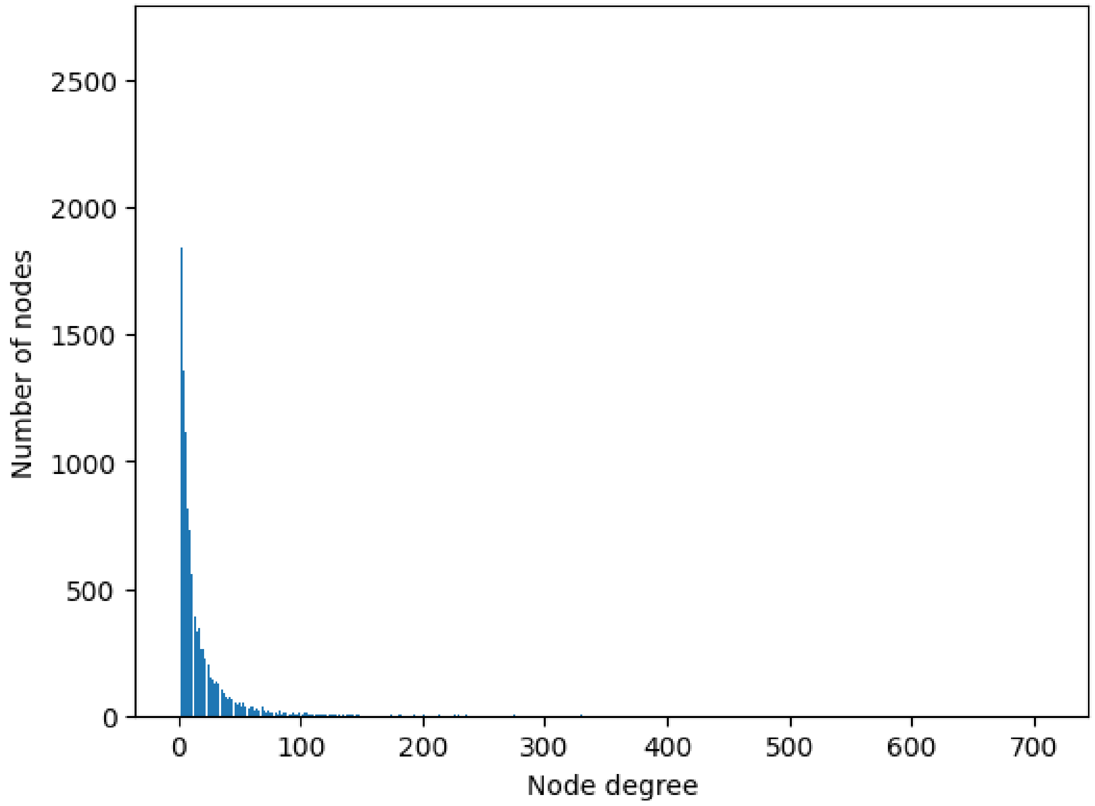
```
<BarContainer object of 37 artists>
```



```
# from torch_geometric.datasets import FacebookPagePage

# # Import dataset from PyTorch Geometric
# dataset = FacebookPagePage(root=".")
# data = dataset[0]

# # Create masks
# data.train_mask = range(18000)
# data.val_mask = range(18001, 20000)
# data.test_mask = range(20001, 22470)

# # Get list of degrees for each node
# degrees = degree(data.edge_index[0]).numpy()

# # Count the number of nodes for each degree
# numbers = Counter(degrees)

# # Bar plot
# fig, ax = plt.subplots()
# ax.set_xlabel('Node degree')
# ax.set_ylabel('Number of nodes')
# plt.bar(numbers.keys(), numbers.values())
```

```
Downloading https://graphmining.ai/datasets/ptg/facebook.npz
Processing...
Done!
```

```
<BarContainer object of 233 artists>
```



This distribution of node degrees looks even more skewed, with a number of neighbors that ranges from 1 to 709. For the same reason, the Facebook Page-Page dataset is also a good case in which to apply a GCN.

We could build our own graph layer but, conveniently enough, PyTorch Geometric already has a predefined GCN layer. Let's implement it on the Cora dataset first:

```
# import torch
# torch.manual_seed(1)
# import torch.nn.functional as F
# from torch_geometric.nn import GCNConv

# dataset = Planetoid(root=".", name="Cora")
# data = dataset[0]

# def accuracy(y_pred, y_true):
#     """Calculate accuracy."""
#     return torch.sum(y_pred == y_true) / len(y_true)


# class GCN(torch.nn.Module):
#     """Graph Convolutional Network"""
#     def __init__(self, dim_in, dim_h, dim_out):
#         super().__init__()
#         self.gcn1 = GCNConv(dim_in, dim_h)
#         self.gcn2 = GCNConv(dim_h, dim_out)
```

(continues on next page)

(continued from previous page)

```
#     def forward(self, x, edge_index):
#         h = self.gcn1(x, edge_index)
#         h = torch.relu(h)
#         h = self.gcn2(h, edge_index)
#         return F.log_softmax(h, dim=1)

#     def fit(self, data, epochs):
#         criterion = torch.nn.CrossEntropyLoss()
#         optimizer = torch.optim.Adam(self.parameters(),
#                                       lr=0.01,
#                                       weight_decay=5e-4)

#         self.train()
#         for epoch in range(epochs+1):
#             optimizer.zero_grad()
#             out = self(data.x, data.edge_index)
#             loss = criterion(out[data.train_mask], data.y[data.train_mask])
#             acc = accuracy(out[data.train_mask].argmax(dim=1),
#                           data.y[data.train_mask])
#             loss.backward()
#             optimizer.step()

#             if(epoch % 20 == 0):
#                 val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
#                 val_acc = accuracy(out[data.val_mask].argmax(dim=1),
#                                   data.y[data.val_mask])
#                 print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} | Train Acc:'
#                       f' {acc*100:>5.2f}% | Val Loss: {val_loss:.2f} | '
#                       f'Val Acc: {val_acc*100:.2f}%')

#     @torch.no_grad()
#     def test(self, data):
#         self.eval()
#         out = self(data.x, data.edge_index)
#         acc = accuracy(out.argmax(dim=1)[data.test_mask], data.y[data.test_mask])
#         return acc

# # Create the Vanilla GNN model
# gcn = GCN(dataset.num_features, 16, dataset.num_classes)
# print(gcn)

# # Train
# gcn.fit(data, epochs=100)

# # Test
# acc = gcn.test(data)
# print(f'\nGCN test accuracy: {acc*100:.2f}%\n')
```

```
GCN(
  (gcn1): GCNConv(1433, 16)
  (gcn2): GCNConv(16, 7)
```

(continues on next page)

(continued from previous page)

```
)
Epoch   0 | Train Loss: 1.932 | Train Acc: 15.71% | Val Loss: 1.94 | Val Acc: 15.20%
Epoch  20 | Train Loss: 0.099 | Train Acc: 100.00% | Val Loss: 0.75 | Val Acc: 77.80%
Epoch  40 | Train Loss: 0.014 | Train Acc: 100.00% | Val Loss: 0.72 | Val Acc: 77.20%
Epoch  60 | Train Loss: 0.015 | Train Acc: 100.00% | Val Loss: 0.71 | Val Acc: 77.80%
Epoch  80 | Train Loss: 0.017 | Train Acc: 100.00% | Val Loss: 0.71 | Val Acc: 77.00%
Epoch 100 | Train Loss: 0.016 | Train Acc: 100.00% | Val Loss: 0.71 | Val Acc: 76.40%

GCN test accuracy: 79.70%
```

If we repeat this experiment 100 times, we obtain an average accuracy score of 80.17% (± 0.61%), which is significantly higher than the 74.98% (± 1.50%) obtained by our vanilla GNN.

The exact same model is then applied to the Facebook Page-Page dataset, where it obtains an average accuracy of 91.54% (± 0.28%). Once again, it is significantly higher than the result obtained by the vanilla GNN, with only 84.85% (± 1.68%).We can attribute these high scores to the wide range of node degrees in these two datasets. By normalizing features and considering the number of neighbors of the central node and its own neighbors, the GCN gains a lot of flexibility and can work well with various types of graphs.

1.33.3 Graph Attention Networks

Graph Attention Networks (GATs) are a theoretical improvement over GCNs. Instead of static normalization coefficients, they propose weighting factors calculated by a process called self-attention. The same process is at the core of one of the most successful deep learning architectures.

we will learn how the graph attention layer works in four steps. This is actually the perfect example for understanding how self-attention works in general. This theoretical background will allow us to implement a graph attention layer from scratch in NumPy.

Introducing the graph attention layer

The main idea behind GATs is that some nodes are more important than others. In fact, this was already the case with the graph convolutional layer: nodes with few neighbors were more important than others, thanks to the normalization coefficient. This approach is limiting because it only takes into account node degrees. On the other hand, the goal of the graph attention layer is to produce weighting factors that also consider the importance of node features.

Let's call our weighting factors attention scores and note, , the attention score between the nodes and . We can define the graph attention operator as follows:

$$h_i = \sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} x_j$$

An important characteristic of GATs is that the attention scores are calculated implicitly by comparing inputs to each other (hence the name self-attention).

we will see how to calculate these attention scores in four steps and also how to make an improvement to the graph attention layer:

- Linear transformation
- Activation function
- Softmax normalization
- Multi-head attention

- Improved graph attention layer

```
# # Import dataset from PyTorch Geometric
# dataset = Planetoid(root=".", name="Cora")
# data = dataset[0]
```

```
# import torch
# torch.manual_seed(1)
# import torch.nn.functional as F
# from torch_geometric.nn import GATv2Conv, GCNConv
# from torch.nn import Linear, Dropout


# def accuracy(y_pred, y_true):
#     """Calculate accuracy."""
#     return torch.sum(y_pred == y_true) / len(y_true)


# class GAT(torch.nn.Module):
#     def __init__(self, dim_in, dim_h, dim_out, heads=8):
#         super().__init__()
#         self.gat1 = GATv2Conv(dim_in, dim_h, heads=heads)
#         self.gat2 = GATv2Conv(dim_h*heads, dim_out, heads=1)

#     def forward(self, x, edge_index):
#         h = F.dropout(x, p=0.6, training=self.training)
#         h = self.gat1(h, edge_index)
#         h = F.elu(h)
#         h = F.dropout(h, p=0.6, training=self.training)
#         h = self.gat2(h, edge_index)
#         return F.log_softmax(h, dim=1)

#     def fit(self, data, epochs):
#         criterion = torch.nn.CrossEntropyLoss()
#         optimizer = torch.optim.Adam(self.parameters(), lr=0.01, weight_decay=0.01)

#         self.train()
#         for epoch in range(epochs+1):
#             optimizer.zero_grad()
#             out = self(data.x, data.edge_index)
#             loss = criterion(out[data.train_mask], data.y[data.train_mask])
#             acc = accuracy(out[data.train_mask].argmax(dim=1), data.y[data.train_mask])
#             loss.backward()
#             optimizer.step()

#             if(epoch % 20 == 0):
#                 val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
#                 val_acc = accuracy(out[data.val_mask].argmax(dim=1), data.y[data.val_
↪mask])
#                 print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} | Train Acc:
↪{acc*100:>5.2f}% | Val Loss: {val_loss:.2f} | Val Acc: {val_acc*100:.2f}%')

#     @torch.no_grad()
```

(continues on next page)

(continued from previous page)

```
#     def test(self, data):
#         self.eval()
#         out = self(data.x, data.edge_index)
#         acc = accuracy(out.argmax(dim=1)[data.test_mask], data.y[data.test_mask])
#         return acc

# # Create the Vanilla GNN model
# gat = GAT(dataset.num_features, 32, dataset.num_classes)
# print(gat)

# # Train
# gat.fit(data, epochs=100)

# # Test
# acc = gat.test(data)
# print(f'GAT test accuracy: {acc*100:.2f}%')
```

```
GAT(
  (gat1): GATv2Conv(1433, 32, heads=8)
  (gat2): GATv2Conv(256, 7, heads=1)
)
Epoch   0 | Train Loss: 1.978 | Train Acc: 12.86% | Val Loss: 1.94 | Val Acc: 13.80%
Epoch  20 | Train Loss: 0.238 | Train Acc: 96.43% | Val Loss: 1.04 | Val Acc: 67.20%
Epoch  40 | Train Loss: 0.165 | Train Acc: 98.57% | Val Loss: 0.95 | Val Acc: 71.00%
Epoch  60 | Train Loss: 0.209 | Train Acc: 96.43% | Val Loss: 0.91 | Val Acc: 71.80%
Epoch  80 | Train Loss: 0.173 | Train Acc: 100.00% | Val Loss: 0.93 | Val Acc: 71.00%
Epoch 100 | Train Loss: 0.190 | Train Acc: 97.86% | Val Loss: 0.96 | Val Acc: 70.60%
GAT test accuracy: 81.00%
```

This accuracy score is slightly better than the average score we obtained with a GCN. We'll make a proper comparison after applying the GAT architecture to the second dataset.

1.33.4 GraphSAGE

GraphSAGE is a GNN architecture designed to handle large graphs. In the tech industry, scalability is a key driver for growth. As a result, systems are inherently designed to accommodate millions of users. This ability requires a fundamental shift in how the GNN model works compared to GCNs and GATs.

Its goal is to generate node embeddings for downstream tasks, such as node classification. In addition, it solves two issues with GCNs and GATs – scaling to large graphs and efficiently generalizing to unseen data. In this section, we will explain how to implement it by describing the two main components of GraphSAGE:

- Neighbor sampling
- Aggregation

Neighbor sampling

So far, we haven't discussed an essential concept in traditional neural networks – mini-batching. It consists of dividing our dataset into smaller fragments, called batches. They are used in gradient descent, the optimization algorithm that finds the best weights and biases during training. There are three types of gradient descent:

Batch gradient descent: Weights and biases are updated after a whole dataset has been processed (every epoch). This is the technique we have implemented so far. However, it is a slow process that requires the dataset to fit in memory.

Stochastic gradient descent: Weights and biases are updated for each training example in the dataset. This is a noisy process because the errors are not averaged. However, it can be used to perform online training.

Mini-batch gradient descent: Weights and biases are updated at the end of every mini-batch of training examples. This technique is faster (mini-batches can be processed in parallel using a GPU) and leads to more stable convergence. In addition, the dataset can exceed the available memory, which is essential for handling large graphs.

Dividing a tabular dataset is straightforward; it simply consists of selecting samples (rows). However, this is an issue regarding graph datasets – how do we choose nodes without breaking essential connections? If we're not careful, we could end up with a collection of isolated nodes where we cannot perform any aggregation.

We have to think about how GNNs use datasets. Every GNN layer computes node embeddings based on their neighbors. This means that computing an embedding only requires the direct neighbors of this node (1 hop). If our GNN has two GNN layers, we need these neighbors and their own neighbors (2 hops), and so on

Aggregation

Now that we've seen how to select the neighboring nodes, we still need to compute embeddings. This is performed by the aggregation operator (or aggregator). In GraphSAGE, the authors have proposed three solutions:

- A mean aggregator
- A long short-term memory (LSTM) aggregator
- A pooling aggregator

Classifying nodes on PubMed

In this section, we will implement a GraphSAGE architecture to perform node classification on the PubMed dataset. The PubMed dataset displays a similar but larger graph, with 19,717 nodes and 88,648 edges.

Node features are TF-IDF-weighted word vectors with 500 dimensions. The goal is to correctly classify nodes into three categories – diabetes mellitus experimental, diabetes mellitus type 1, and diabetes mellitus type 2. Let's implement it step by step using PyG:

```
# dataset = Planetoid(root='.', name="Pubmed")
# data = dataset[0]

# # Print information about the dataset
# print(f'Dataset: {dataset}')
# print('-------------------')
# print(f'Number of graphs: {len(dataset)}')
# print(f'Number of nodes: {data.x.shape[0]}')
# print(f'Number of features: {dataset.num_features}')
# print(f'Number of classes: {dataset.num_classes}')

# # Print information about the graph
```

(continues on next page)

(continued from previous page)

```
# print(f'\nGraph:')
# print('------')
# print(f'Training nodes: {sum(data.train_mask).item()}')
# print(f'Evaluation nodes: {sum(data.val_mask).item()}')
# print(f'Test nodes: {sum(data.test_mask).item()}')
# print(f'Edges are directed: {data.is_directed()}')
# print(f'Graph has isolated nodes: {data.has_isolated_nodes()}')
# print(f'Graph has loops: {data.has_self_loops()}')
```

```
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.pubmed.x
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.pubmed.tx
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.pubmed.allx
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.pubmed.y
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.pubmed.ty
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.pubmed.ally
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.pubmed.graph
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.pubmed.test.index
Processing...
```

```
Dataset: Pubmed()
-------------------
Number of graphs: 1
Number of nodes: 19717
Number of features: 500
Number of classes: 3

Graph:
------
Training nodes: 60
Evaluation nodes: 500
Test nodes: 1000
Edges are directed: False
Graph has isolated nodes: False
Graph has loops: False
```

```
Done!
```

```
# !pip install  torch_sparse
# !pip install  pyg_lib
```

```
Collecting torch_sparse
  Downloading torch_sparse-0.6.17.tar.gz (209 kB)
      209.2/209.2 kB 1.6 MB/s eta 0:00:00a 0:00:01
?25h  Preparing metadata (setup.py) ... ?25ldone
?25hRequirement already satisfied: scipy in /Users/ikram.ali/miniconda3/envs/ml_notes/
↪lib/python3.11/site-packages (from torch_sparse) (1.10.1)
Requirement already satisfied: numpy<1.27.0,>=1.19.5 in /Users/ikram.ali/miniconda3/envs/
↪ml_notes/lib/python3.11/site-packages (from scipy->torch_sparse) (1.24.2)
Building wheels for collected packages: torch_sparse
  Building wheel for torch_sparse (setup.py) ... ?25ldone
?25h  Created wheel for torch_sparse: filename=torch_sparse-0.6.17-cp311-cp311-macosx_11_
```

(continues on next page)

(continued from previous page)

```
↪0_arm64.whl size=460727␣
↪sha256=b43676345bd8a5d2d5d938834745dad52c789b7081b564a3769f64e0c2366511
  Stored in directory: /Users/ikram.ali/Library/Caches/pip/wheels/43/27/bc/
↪21943b121fafafd76c514e5f34c8ad8592766bee55f6771b43
Successfully built torch_sparse
Installing collected packages: torch_sparse
Successfully installed torch_sparse-0.6.17
```

```
# !pip install pyg_lib
```

```
ERROR: Could not find a version that satisfies the requirement pyg_lib (from versions:␣
↪none)
ERROR: No matching distribution found for pyg_lib
```

```
# from torch_geometric.loader import NeighborLoader
# from torch_geometric.utils import to_networkx

# # Create batches with neighbor sampling
# train_loader = NeighborLoader(
#     data,
#     num_neighbors=[5, 10],
#     batch_size=16,
#     input_nodes=data.train_mask,
# )

# # Print each subgraph
# for i, subgraph in enumerate(train_loader):
#     print(f'Subgraph {i}: {subgraph}')
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In[68], line 13
      5 train_loader = NeighborLoader(
      6     data,
      7     num_neighbors=[5, 10],
      8     batch_size=16,
      9     input_nodes=data.train_mask,
     10 )
     12 # Print each subgraph
---> 13 for i, subgraph in enumerate(train_loader):
     14     print(f'Subgraph {i}: {subgraph}')

File ~/miniconda3/envs/ml_notes/lib/python3.11/site-packages/torch_geometric/loader/base.
↪py:36, in DataLoaderIterator.__next__(self)
     35 def __next__(self) -> Any:
---> 36     return self.transform_fn(next(self.iterator))

File ~/miniconda3/envs/ml_notes/lib/python3.11/site-packages/torch/utils/data/dataloader.
↪py:634, in _BaseDataLoaderIter.__next__(self)
    631 if self._sampler_iter is None:
```

(continues on next page)

(continued from previous page)

```
    632     # TODO(https://github.com/pytorch/pytorch/issues/76750)
    633     self._reset()  # type: ignore[call-arg]
--> 634 data = self._next_data()
    635 self._num_yielded += 1
    636 if self._dataset_kind == _DatasetKind.Iterable and \
    637         self._IterableDataset_len_called is not None and \
    638         self._num_yielded > self._IterableDataset_len_called:

File ~/miniconda3/envs/ml_notes/lib/python3.11/site-packages/torch/utils/data/dataloader.
↪py:678, in _SingleProcessDataLoaderIter._next_data(self)
    676 def _next_data(self):
    677     index = self._next_index()  # may raise StopIteration
--> 678     data = self._dataset_fetcher.fetch(index)  # may raise StopIteration
    679     if self._pin_memory:
    680         data = _utils.pin_memory.pin_memory(data, self._pin_memory_device)

File ~/miniconda3/envs/ml_notes/lib/python3.11/site-packages/torch/utils/data/_utils/
↪fetch.py:54, in _MapDatasetFetcher.fetch(self, possibly_batched_index)
     52 else:
     53     data = self.dataset[possibly_batched_index]
---> 54 return self.collate_fn(data)

File ~/miniconda3/envs/ml_notes/lib/python3.11/site-packages/torch_geometric/loader/node_
↪loader.py:117, in NodeLoader.collate_fn(self, index)
    114 r"""Samples a subgraph from a batch of input nodes."""
    115 input_data: NodeSamplerInput = self.input_data[index]
--> 117 out = self.node_sampler.sample_from_nodes(input_data)
    119 if self.filter_per_worker:  # Execute `filter_fn` in the worker process
    120     out = self.filter_fn(out)

File ~/miniconda3/envs/ml_notes/lib/python3.11/site-packages/torch_geometric/sampler/
↪neighbor_sampler.py:174, in NeighborSampler.sample_from_nodes(self, inputs)
    170 def sample_from_nodes(
    171     self,
    172     inputs: NodeSamplerInput,
    173 ) -> Union[SamplerOutput, HeteroSamplerOutput]:
--> 174     return node_sample(inputs, self._sample)

File ~/miniconda3/envs/ml_notes/lib/python3.11/site-packages/torch_geometric/sampler/
↪neighbor_sampler.py:358, in node_sample(inputs, sample_fn)
    355     seed = inputs.node
    356     seed_time = inputs.time
--> 358 out = sample_fn(seed, seed_time)
    359 out.metadata = (inputs.input_id, inputs.time)
    361 return out

File ~/miniconda3/envs/ml_notes/lib/python3.11/site-packages/torch_geometric/sampler/
↪neighbor_sampler.py:325, in NeighborSampler._sample(self, seed, seed_time, **kwargs)
    322     num_sampled_nodes = num_sampled_edges = None
    324 else:
--> 325     raise ImportError(f"'{self.__class__.__name__}' requires "
    326                       f"either 'pyg-lib' or 'torch-sparse'")
```

(continues on next page)

(continued from previous page)

```
    328 return SamplerOutput(
    329     node=node,
    330     row=row,
   (...)
    335     num_sampled_edges=num_sampled_edges,
    336 )

ImportError: 'NeighborSampler' requires either 'pyg-lib' or 'torch-sparse'
```

```
# import numpy as np
# import networkx as nx
# import matplotlib.pyplot as plt

# # Plot each subgraph
# fig = plt.figure(figsize=(16,16))
# for idx, (subdata, pos) in enumerate(zip(train_loader, [221, 222, 223, 224])):
#     G = to_networkx(subdata, to_undirected=True)
#     ax = fig.add_subplot(pos)
#     ax.set_title(f'Subgraph {idx}', fontsize=24)
#     plt.axis('off')
#     nx.draw_networkx(G,
#                     pos=nx.spring_layout(G, seed=0),
#                     with_labels=False,
#                     node_color=subdata.y,
#                     )
# plt.show()
```

```
# import torch.nn.functional as F
# from torch_geometric.nn import SAGEConv


# def accuracy(pred_y, y):
#     """Calculate accuracy."""
#     return ((pred_y == y).sum() / len(y)).item()


# class GraphSAGE(torch.nn.Module):
#     """GraphSAGE"""
#     def __init__(self, dim_in, dim_h, dim_out):
#         super().__init__()
#         self.sage1 = SAGEConv(dim_in, dim_h)
#         self.sage2 = SAGEConv(dim_h, dim_out)

#     def forward(self, x, edge_index):
#         h = self.sage1(x, edge_index)
#         h = torch.relu(h)
#         h = F.dropout(h, p=0.5, training=self.training)
#         h = self.sage2(h, edge_index)
#         return F.log_softmax(h, dim=1)

#     def fit(self, data, epochs):
```

(continues on next page)

(continued from previous page)

```
#         criterion = torch.nn.CrossEntropyLoss()
#         optimizer = torch.optim.Adam(self.parameters(), lr=0.01)

#         self.train()
#         for epoch in range(epochs+1):
#             total_loss = 0
#             acc = 0
#             val_loss = 0
#             val_acc = 0

#             # Train on batches
#             for batch in train_loader:
#                 optimizer.zero_grad()
#                 out = self(batch.x, batch.edge_index)
#                 loss = criterion(out[batch.train_mask], batch.y[batch.train_mask])
#                 total_loss += loss.item()
#                 acc += accuracy(out[batch.train_mask].argmax(dim=1), batch.y[batch.
↪train_mask])
#                 loss.backward()
#                 optimizer.step()

#                 # Validation
#                 val_loss += criterion(out[batch.val_mask], batch.y[batch.val_mask])
#                 val_acc += accuracy(out[batch.val_mask].argmax(dim=1), batch.y[batch.
↪val_mask])

#             # Print metrics every 10 epochs
#             if epoch % 20 == 0:
#                 print(f'Epoch {epoch:>3} | Train Loss: {loss/len(train_loader):.3f} |␣
↪Train Acc: {acc/len(train_loader)*100:>6.2f}% | Val Loss: {val_loss/len(train_loader):.
↪2f} | Val Acc: {val_acc/len(train_loader)*100:.2f}%')

#     @torch.no_grad()
#     def test(self, data):
#         self.eval()
#         out = self(data.x, data.edge_index)
#         acc = accuracy(out.argmax(dim=1)[data.test_mask], data.y[data.test_mask])
#         return acc
```

1.34 Graph Equations

Explaining the graph neural networks equations

1.34.1 GCN layer

Semi-Supervised Classification with Graph Convolutional Networks https://arxiv.org/pdf/1609.02907.pdf

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}_i} \frac{1}{c_{ij}} h_j^{(l)} W^{(l)}\right)$$

where (+) denotes a trainable weight matrix of shape [num_output_features, num_input_features] and , refers to a fixed normalization coefficient for each edge.

PyG implements this layer via GCNConv

1.35 Pytorch Fundamental

PyTorch allows you to manipulate and process data and write machine learning algorithms using Python code.

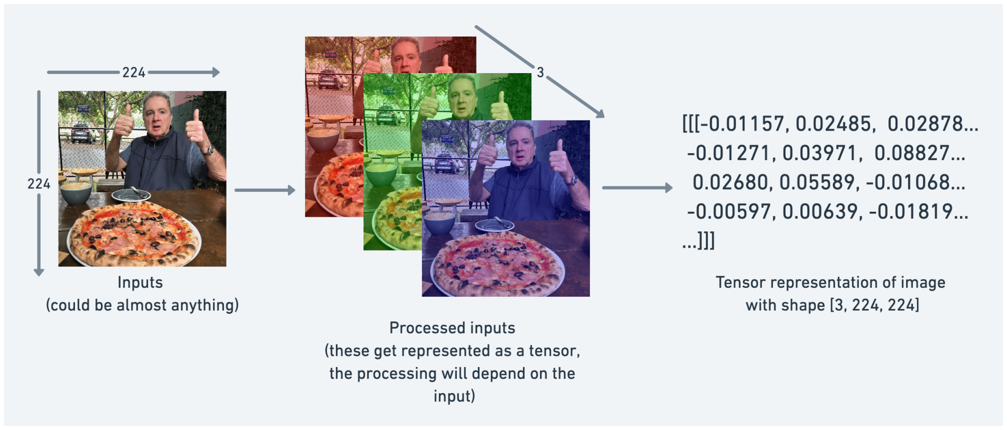
```python
import torch
import numpy as np

torch.__version__
```

```
'2.2.2+cu121'
```

1.35.1 Tensors

Tensors are the fundamental building block of machine learning.

For example, you could represent an image as a tensor with shape [3, 224, 224] which would mean [colour_channels, height, width], as in the image has 3 colour channels (red, green, blue), a height of 224 pixels and a width of 224 pixels.



Scalar

```
# Scalar
scalar = torch.tensor(7)
scalar
```

```
tensor(7)
```

```
scalar.shape
```

```
torch.Size([])
```

```
# We can check the dimensions of a tensor
scalar.ndim
```

```
0
```

```
# If we wan to retrieve the value of a scalar tensor, we can use the .item() method
scalar.item()
```

```
7
```

Vectors

```
vector = torch.tensor([7, 7])
vector
```

```
tensor([7, 7])
```

```
# Check the number of dimensions of vector

# You can tell the number of dimensions a tensor in PyTorch has by the number of square
# brackets on the outside ([) and you only need to count one side.
vector.ndim
```

```
1
```

```
#  The shape tells you how the elements inside them are arranged.
vector.shape
```

```
torch.Size([2])
```

Matrix

```
# Matrix
MATRIX = torch.tensor([[7, 8],
                       [9, 10]])
MATRIX
```

```
tensor([[ 7,  8],
        [ 9, 10]])
```

```
# Wow! More numbers! Matrices are as flexible as vectors, except they've got an extra␣
↪dimension.

MATRIX.ndim
```

```
2
```

```
MATRIX.shape
```

```
torch.Size([2, 2])
```

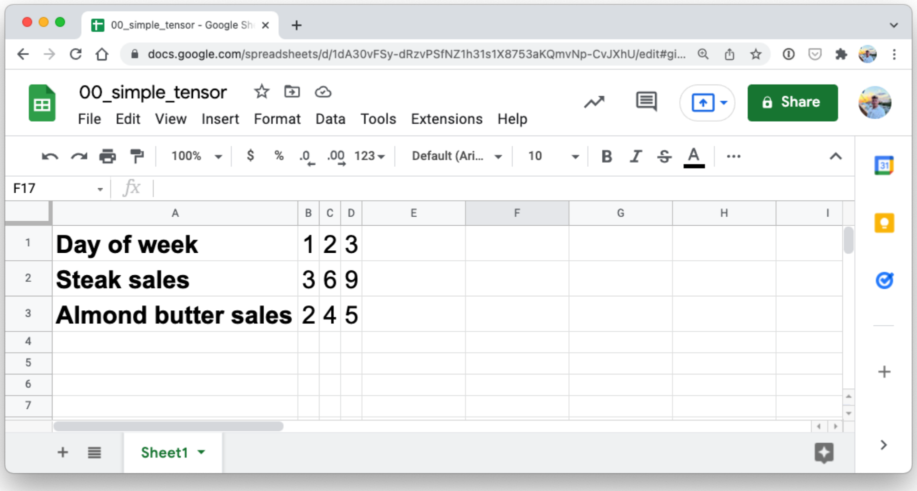
Tensor

I want to stress that tensors can represent almost anything.

```
TENSOR = torch.tensor([[[1, 2, 3],
                        [3, 6, 9],
                        [2, 4, 5]]])
TENSOR
```

```
tensor([[[1, 2, 3],
         [3, 6, 9],
         [2, 4, 5]]])
```

The one we just created could be the sales numbers for a steak and almond butter store (two of my favourite foods).


00_simple_tensor - Google Sh
docs.google.com/spreadsheets/d/1dA30vFSy-dRzvPSfNZ1h31s1X8753aKQmvNp-CvJXhU/edit#gi...
00_simple_tensor
File Edit View Insert Format Data Tools Extensions Help
Share
F17
Sheet1

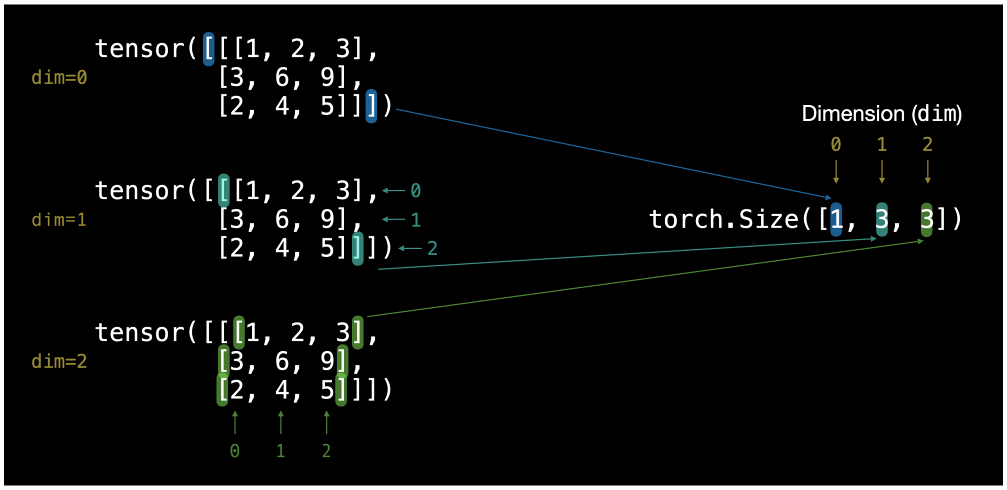
Day of week	1	2	3
Steak sales	3	6	9
Almond butter sales	2	4	5

```
TENSOR.ndim, TENSOR.shape
```

```
(3, torch.Size([1, 3, 3]))
```

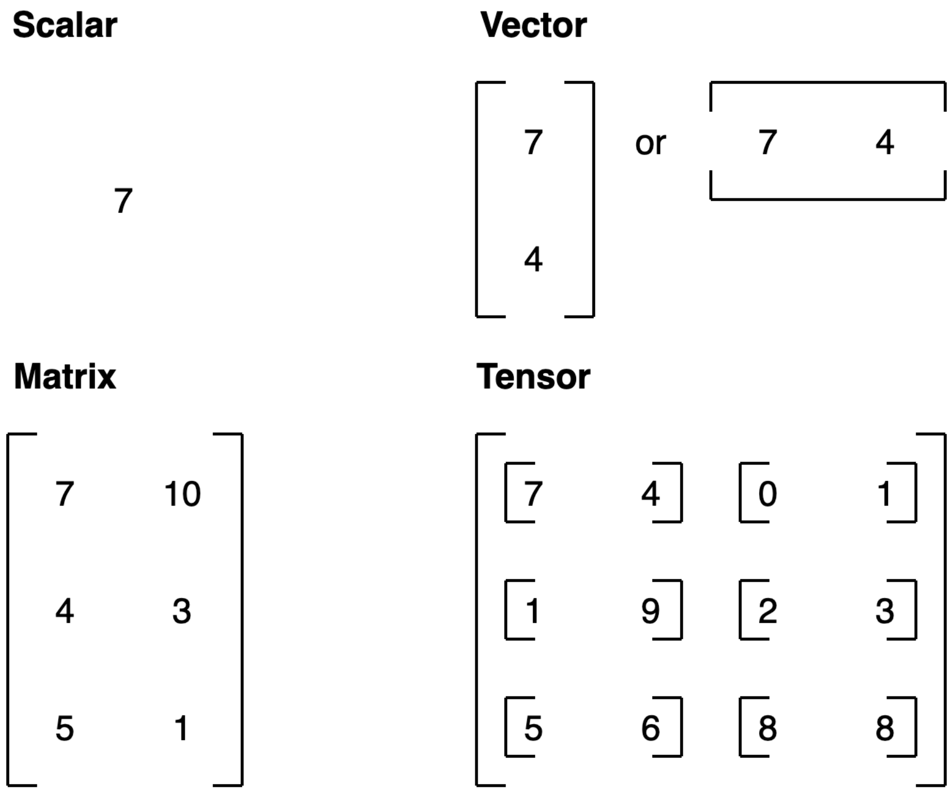
The dimensions go outer to inner.

That means there's 1 dimension of 3 by 3.


```
tensor([[[1, 2, 3],
         [3, 6, 9],
         [2, 4, 5]]])
```

Note: You might've noticed me using lowercase letters for scalar and vector and uppercase letters for MATRIX and TENSOR. This was on purpose. In practice, you'll often see scalars and vectors denoted as lowercase letters such as y or a. And matrices and tensors denoted as uppercase letters such as X or W.

You also might notice the names martrix and tensor used interchangably. This is common. Since in PyTorch you're often dealing with torch.Tensor's (hence the tensor name), however, the shape and dimensions of what's inside will dictate what it actually is.



1.35.2 Random tensors

But when building machine learning models with PyTorch, it's rare you'll create tensors by hand (like what we've being doing).

Instead, a machine learning model often starts out with large random tensors of numbers and adjusts these random numbers as it works through data to better represent it.

In essence:

Start with random numbers -> look at data -> update random numbers -> look at data -> update random numbers…

```
# Create a random tensor of size (3, 4)
random_tensor = torch.rand(size=(3, 4))
random_tensor, random_tensor.dtype, random_tensor.shape , random_tensor.ndim
```

```
(tensor([[0.9638, 0.9171, 0.8510, 0.3581],
         [0.7802, 0.7227, 0.5245, 0.0317],
         [0.0926, 0.1394, 0.3519, 0.5473]]),
 torch.float32,
 torch.Size([3, 4]),
 2)
```

The flexibility of torch.rand() is that we can adjust the size to be whatever we want.

For example, say you wanted a random tensor in the common image shape of [224, 224, 3] ([height, width, color_channels])

```
# Create a random tensor of size (224, 224, 3)
random_image_size_tensor = torch.rand(size=(224, 224, 3))
random_image_size_tensor.shape, random_image_size_tensor.ndim
```

```
(torch.Size([224, 224, 3]), 3)
```

```
random_image_size_tensor = torch.rand(size=(1, 3, 3))
random_image_size_tensor, random_image_size_tensor.ndim
```

```
(tensor([[[0.2242, 0.1729, 0.9013],
          [0.7966, 0.1042, 0.0178],
          [0.9993, 0.1032, 0.1653]]]),
 3)
```

Zeros and ones

Sometimes you'll just want to fill tensors with zeros or ones.

This happens a lot with masking (like masking some of the values in one tensor with zeros to let a model know not to learn them).

```
zeros = torch.zeros(size=(3, 4))
zeros, zeros.dtype
```

```
(tensor([[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]),
 torch.float32)
```

```
ones = torch.ones(size=(3, 4))
ones, ones.dtype
```

```
(tensor([[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]),
 torch.float32)
```

```
# Create a range of values 0 to 10
zero_to_ten = torch.arange(start=0, end=10, step=1)
zero_to_ten
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
# Can also create a tensor of zeros similar to another tensor
ten_zeros = torch.zeros_like(input=zero_to_ten) # will have same shape
ten_zeros
```

```
tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

1.35.3 Tensor datatypes

There are many different tensor datatypes available in PyTorch.

Some are specific for CPU and some are better for GPU.

Generally if you see torch.cuda anywhere, the tensor is being used for GPU (since Nvidia GPUs use a computing toolkit called CUDA).

The most common type (and generally the default) is torch.float32 or torch.float.

This is referred to as "32-bit floating point".

But there's also 16-bit floating point (torch.float16 or torch.half) and 64-bit floating point (torch.float64 or torch.double).

And to confuse things even more there's also 8-bit, 16-bit, 32-bit and 64-bit integers.

Plus more!

Note: An integer is a flat round number like 7 whereas a float has a decimal 7.0.

The reason for all of these is to do with precision in computing.

Precision is the amount of detail used to describe a number.

The higher the precision value (8, 16, 32), the more detail and hence data used to express a number.

This matters in deep learning and numerical computing because you're making so many operations, the more detail you have to calculate on, the more compute you have to use.

So lower precision datatypes are generally faster to compute on but sacrifice some performance on evaluation metrics like accuracy (faster to compute but less accurate).

```
# Default datatype for tensors is float32
float_32_tensor = torch.tensor([3.0, 6.0, 9.0],
                               dtype=None, # defaults to None, which is torch.float32 or␣
↪whatever datatype is passed
                               device=None, # defaults to None, which uses the default␣
↪tensor type
                               requires_grad=False) # if True, operations performed on␣
↪the tensor are recorded

float_32_tensor.shape, float_32_tensor.dtype, float_32_tensor.device
```

```
(torch.Size([3]), torch.float32, device(type='cpu'))
```

Aside from shape issues (tensor shapes don't match up), two of the other most common issues you'll come across in PyTorch are datatype and device issues.

For example, one of tensors is torch.float32 and the other is torch.float16 (PyTorch often likes tensors to be the same format).

Or one of your tensors is on the CPU and the other is on the GPU (PyTorch likes calculations between tensors to be on the same device).

We'll see more of this device talk later on.

For now let's create a tensor with dtype=torch.float16.

```
float_16_tensor = torch.tensor([3.0, 6.0, 9.0],
                               dtype=torch.float16) # torch.half would also work
```

(continues on next page)

(continued from previous page)

```
float_16_tensor.dtype
```

```
torch.float16
```

When you run into issues in PyTorch, it's very often one to do with one of the three attributes above. So when the error messages show up, sing yourself a little song called "what, what, where":

"what shape are my tensors? what datatype are they and where are they stored? what shape, what datatype, where where where"

1.35.4 Tensor Operations

In deep learning, data (images, text, video, audio, protein structures, etc) gets represented as tensors.

A model learns by investigating those tensors and performing a series of operations (could be 1,000,000s+) on tensors to create a representation of the patterns in the input data.

These operations are often a wonderful dance between:

Addition Substraction Multiplication (element-wise) Division Matrix multiplication And that's it. Sure there are a few more here and there but these are the basic building blocks of neural networks.

Addition & Multiply

```
# Create a tensor of values and add a number to it
tensor = torch.tensor([1, 2, 3])
tensor + 10
```

```
tensor([11, 12, 13])
```

```
# Multiply it by 10
tensor * 10
```

```
tensor([10, 20, 30])
```

```
# Tensors don't change unless reassigned
tensor
```

```
tensor([1, 2, 3])
```

```
# Subtract and reassign
tensor = tensor - 10
tensor
```

```
tensor([-9, -8, -7])
```

```
# Can also use torch functions
torch.multiply(tensor, 10)
```

```
tensor([-90, -80, -70])
```

```
# Element-wise multiplication (each element multiplies its equivalent, index 0->0, 1->1,␣
↪2->2)
print(tensor, "*", tensor)
print("Equals:", tensor * tensor)
```

```
tensor([-9, -8, -7]) * tensor([-9, -8, -7])
Equals: tensor([81, 64, 49])
```

```
# torch.tensor([1, 2, 3]) * torch.tensor([1, 2])

# This will create an error ""the size of tensor a (3) must match the size of tensor b␣
↪(2) at non-singleton dimension 0"
```

Matrix multiplication

Matrix multiplication (is all you need)

PyTorch implements matrix multiplication functionality in the torch.matmul() method.

The main two rules for matrix multiplication to remember are:

The inner dimensions must match: (3, 2) @ (3, 2) won't work (2, 3) @ (3, 2) will work (3, 2) @ (2, 3) will work The resulting matrix has the shape of the outer dimensions: (2, 3) @ (3, 2) -> (2, 2) (3, 2) @ (2, 3) -> (3, 3)

```
tensor = torch.tensor([1, 2, 3])
tensor.shape
```

```
torch.Size([3])
```

```
# Element-wise matrix multiplication
tensor * tensor
```

```
tensor([1, 4, 9])
```

```
# Matrix multiplication
torch.matmul(tensor, tensor)
# torch.mm(tensor, tensor)
```

```
tensor(14)
```

```
# Can also use the "@" symbol for matrix multiplication, though not recommended
tensor @ tensor
```

```
tensor(14)
```

```
%%time
torch.matmul(tensor, tensor)
```

```
CPU times: user 150 µs, sys: 28 µs, total: 178 µs
Wall time: 128 µs
```

```
tensor(14)
```

One of the most common errors in deep learning (shape errors)

Because much of deep learning is multiplying and performing operations on matrices and matrices have a strict rule about what shapes and sizes can be combined, one of the most common errors you'll run into in deep learning is shape mismatches.

```
# Shapes need to be in the right way
tensor_A = torch.tensor([[1, 2],
                         [3, 4],
                         [5, 6]], dtype=torch.float32)

tensor_B = torch.tensor([[7, 10],
                         [8, 11],
                         [9, 12]], dtype=torch.float32)

tensor_A.shape, tensor_B.shape
# torch.matmul(tensor_A, tensor_B) # (this will error)
```

```
(torch.Size([3, 2]), torch.Size([3, 2]))
```

```
# One of the ways to do this is with a transpose (switch the dimensions of a given␣
↪tensor).
print(f"Original shapes: tensor_A = {tensor_A.shape}, tensor_B = {tensor_B.shape}\n")
print(f"New shapes: tensor_A = {tensor_A.shape} (same as above), tensor_B.T = {tensor_B.
↪T.shape}\n")
print(f"Multiplying: {tensor_A.shape} * {tensor_B.T.shape} <- inner dimensions match\n")
print("Output:\n")
output = torch.matmul(tensor_A, tensor_B.T)
print(output)
print(f"\nOutput shape: {output.shape}")
```

```
Original shapes: tensor_A = torch.Size([3, 2]), tensor_B = torch.Size([3, 2])

New shapes: tensor_A = torch.Size([3, 2]) (same as above), tensor_B.T = torch.Size([2,␣
↪3])

Multiplying: torch.Size([3, 2]) * torch.Size([2, 3]) <- inner dimensions match

Output:

tensor([[ 27.,  30.,  33.],
        [ 61.,  68.,  75.],
        [ 95., 106., 117.]])

Output shape: torch.Size([3, 3])
```

```
# torch.mm is a shortcut for matmul
# A matrix multiplication like this is also referred to as the dot product of two␣
↪matrices.
torch.mm(tensor_A, tensor_B.T)
```

```
tensor([[ 27.,  30.,  33.],
        [ 61.,  68.,  75.],
        [ 95., 106., 117.]])
```

Neural networks are full of matrix multiplications and dot products.

The torch.nn.Linear() module (we'll see this in action later on), also known as a feed-forward layer or fully connected layer, implements a matrix multiplication between an input x and a weights matrix A.

y= xAT+b

Where:

x is the input to the layer (deep learning is a stack of layers like torch.nn.Linear() and others on top of each other). A is the weights matrix created by the layer, this starts out as random numbers that get adjusted as a neural network learns to better represent patterns in the data (notice the "T", that's because the weights matrix gets transposed). Note: You might also often see W or another letter like X used to showcase the weights matrix. b is the bias term used to slightly offset the weights and inputs. y is the output (a manipulation of the input in the hopes to discover patterns in it). This is a linear function (you may have seen something like $y = mx + b$ in high school or elsewhere), and can be used to draw a straight line!

Let's play around with a linear layer.

Try changing the values of in_features and out_features below and see what happens.

Do you notice anything to do with the shapes?

```
tensor_A
```

```
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
```

```
# Since the linear layer starts with a random weights matrix, let's make it reproducible␣
↪(more on this later)
torch.manual_seed(42)
# This uses matrix multiplication
linear = torch.nn.Linear(in_features=2, # in_features = matches inner dimension of input
                         out_features=6) # out_features = describes outer value
x = tensor_A
output = linear(x)
print(f"Input shape: {x.shape}\n")
print(f"Output:\n{output}\n\nOutput shape: {output.shape}")
```

```
Input shape: torch.Size([3, 2])

Output:
tensor([[2.2368, 1.2292, 0.4714, 0.3864, 0.1309, 0.9838],
        [4.4919, 2.1970, 0.4469, 0.5285, 0.3401, 2.4777],
        [6.7469, 3.1648, 0.4224, 0.6705, 0.5493, 3.9716]],
```

(continues on next page)

(continued from previous page)

```
       grad_fn=<AddmmBackward0>)

Output shape: torch.Size([3, 6])
```

```
# Finding the min, max, mean, sum
x = torch.arange(0, 100, 10)
x
```

```
tensor([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

```
print(f"Minimum: {x.min()}")
print(f"Maximum: {x.max()}")
# print(f"Mean: {x.mean()}") # this will error
print(f"Mean: {x.type(torch.float32).mean()}") # won't work without float datatype
print(f"Sum: {x.sum()}")
```

```
Minimum: 0
Maximum: 90
Mean: 45.0
Sum: 450
```

```
torch.max(x), torch.min(x), torch.mean(x.type(torch.float32)), torch.sum(x)
```

```
(tensor(90), tensor(0), tensor(45.), tensor(450))
```

Positional min/max

You can also find the index of a tensor where the max or minimum occurs with torch.argmax() and torch.argmin() respectively.

This is helpful incase you just want the position where the highest (or lowest) value is and not the actual value itself (we'll see this in a later section when using the softmax activation function).

```
# Create a tensor
tensor = torch.arange(10, 100, 10)
print(f"Tensor: {tensor}")

# Returns index of max and min values
print(f"Index where max value occurs: {tensor.argmax()}")
print(f"Index where min value occurs: {tensor.argmin()}")
```

```
Tensor: tensor([10, 20, 30, 40, 50, 60, 70, 80, 90])
Index where max value occurs: 8
Index where min value occurs: 0
```

Reshaping, stacking, squeezing and unsqueezing

Because deep learning models (neural networks) are all about manipulating tensors in some way. And because of the rules of matrix multiplication, if you've got shape mismatches, you'll run into errors. These methods help you make the right elements of your tensors are mixing with the right elements of other tensors.

```
x = torch.arange(1., 8.)
x, x.shape
```

```
(tensor([1., 2., 3., 4., 5., 6., 7.]), torch.Size([7]))
```

```
# Add an extra dimension
x_reshaped = x.reshape(1, 7)
x_reshaped, x_reshaped.shape
```

```
(tensor([[1., 2., 3., 4., 5., 6., 7.]]), torch.Size([1, 7]))
```

```
z = x.view(1, 7)
z, z.shape
```

```
(tensor([[1., 2., 3., 4., 5., 6., 7.]]), torch.Size([1, 7]))
```

Remember though, changing the view of a tensor with torch.view() really only creates a new view of the same tensor.

So changing the view changes the original tensor too.

```
# Changing z changes x
z[:, 0] = 5
z, x
```

```
(tensor([[5., 2., 3., 4., 5., 6., 7.]]), tensor([5., 2., 3., 4., 5., 6., 7.]))
```

```
# if we wanted to stack our new tensor on top of itself five times, we could do so with␣
↪torch.stack().

# Stack tensors on top of each other
x_stacked = torch.stack([x, x, x, x], dim=0) # try changing dim to dim=1 and see what␣
↪happens
x_stacked
```

```
tensor([[5., 2., 3., 4., 5., 6., 7.],
        [5., 2., 3., 4., 5., 6., 7.],
        [5., 2., 3., 4., 5., 6., 7.],
        [5., 2., 3., 4., 5., 6., 7.]])
```

How about removing all single dimensions from a tensor?

To do so you can use torch.squeeze() (I remember this as squeezing the tensor to only have dimensions over 1).

```
print(f"Previous tensor: {x_reshaped}")
print(f"Previous shape: {x_reshaped.shape}")
```

(continues on next page)

(continued from previous page)

```
# Remove extra dimension from x_reshaped
x_squeezed = x_reshaped.squeeze()
print(f"\nNew tensor: {x_squeezed}")
print(f"New shape: {x_squeezed.shape}")
```

```
Previous tensor: tensor([[5., 2., 3., 4., 5., 6., 7.]])
Previous shape: torch.Size([1, 7])

New tensor: tensor([5., 2., 3., 4., 5., 6., 7.])
New shape: torch.Size([7])
```

```
#  And to do the reverse of torch.squeeze() you can use torch.unsqueeze() to add a␣
↪dimension value of 1 at a specific index.

print(f"Previous tensor: {x_squeezed}")
print(f"Previous shape: {x_squeezed.shape}")

## Add an extra dimension with unsqueeze
x_unsqueezed = x_squeezed.unsqueeze(dim=0)
print(f"\nNew tensor: {x_unsqueezed}")
print(f"New shape: {x_unsqueezed.shape}")
```

```
Previous tensor: tensor([5., 2., 3., 4., 5., 6., 7.])
Previous shape: torch.Size([7])

New tensor: tensor([[5., 2., 3., 4., 5., 6., 7.]])
New shape: torch.Size([1, 7])
```

You can also rearrange the order of axes values with torch.permute(input, dims), where the input gets turned into a view with new dims.

```
# Create tensor with specific shape
x_original = torch.rand(size=(224, 224, 3))

# Permute the original tensor to rearrange the axis order
x_permuted = x_original.permute(2, 0, 1) # shifts axis 0->1, 1->2, 2->0

print(f"Previous shape: {x_original.shape}")
print(f"New shape: {x_permuted.shape}")
```

```
Previous shape: torch.Size([224, 224, 3])
New shape: torch.Size([3, 224, 224])
```

1.35.5 Indexing

Sometimes you'll want to select specific data from tensors (for example, only the first column or second row).

```
x = torch.arange(1, 10).reshape(1, 3, 3)
x, x.shape
```

```
(tensor([[[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]]),
 torch.Size([1, 3, 3]))
```

Indexing values goes outer dimension -> inner dimension (check out the square brackets).

```
# Let's index bracket by bracket
print(f"First square bracket:\n{x[0]}")
print(f"Second square bracket: {x[0][0]}")
print(f"Third square bracket: {x[0][0][0]}")
```

```
First square bracket:
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
Second square bracket: tensor([1, 2, 3])
Third square bracket: 1
```

You can also use : to specify "all values in this dimension" and then use a comma (,) to add another dimension.

```
x[:]
```

```
tensor([[[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]])
```

```
x[:,:,:,]
```

```
tensor([[[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]])
```

```
# Get all values of 0th dimension and the 0 index of 1st dimension
x[:, 0]
```

```
tensor([[1, 2, 3]])
```

```
# Get all values of 0th & 1st dimensions but only index 1 of 2nd dimension
x[:, :, 1]
```

```
tensor([[2, 5, 8]])
```

```
# Get all values of the 0 dimension but only the 1 index value of the 1st and 2nd␣
↪dimension
x[:, 1, 1]
```

```
tensor([5])
```

```
# Get index 0 of 0th and 1st dimension and all values of 2nd dimension
x[0, 0, :] # same as x[0][0]
```

```
tensor([1, 2, 3])
```

1.35.6 Pytorch Best Practise

You can also use torch.as_tensor() to convert a numpy array to a torch tensor, This will not create a new copy of the data

```
a = np.random.rand(3, 3)
# Bad way
t1 = torch. tensor(a)

# Good way
t2 = torch.as_tensor(a)
t3 = torch.from_numpy(a)
```

Avoid cpu, item() these will use functions to tranfer data between devices

```
t= torch.rand(2,2)
# bad way
t.cpu ()
t[0][0].item()
t. numpy ()

# good way
t.detach ()
```

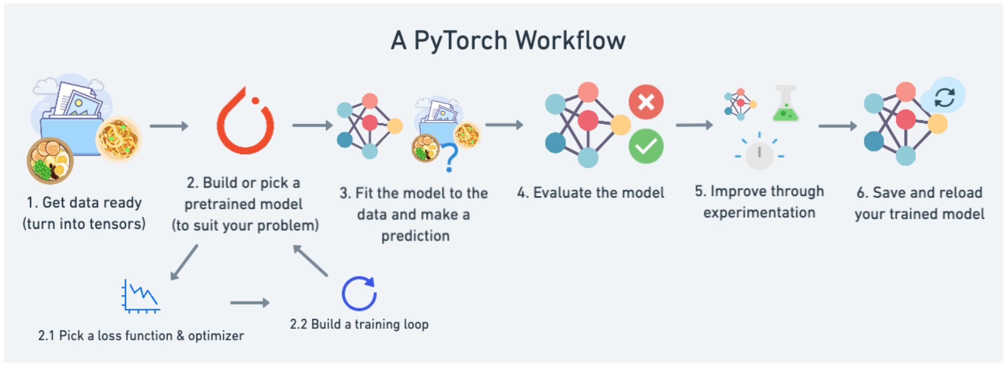
```
tensor([[0.8016, 0.3649],
        [0.6286, 0.9663]])
```

Create tensor direclty on GPU

```
# bad way
# t = torch.rand(2,2).cuda()

# good way
# t = torch. rand(2,2, device="cuda")
```

1.36 Pytorch Workflow

The essence of machine learning and deep learning is to take some data from the past, build an algorithm (like a neural network) to discover patterns in it and use the discoverd patterns to predict the future.

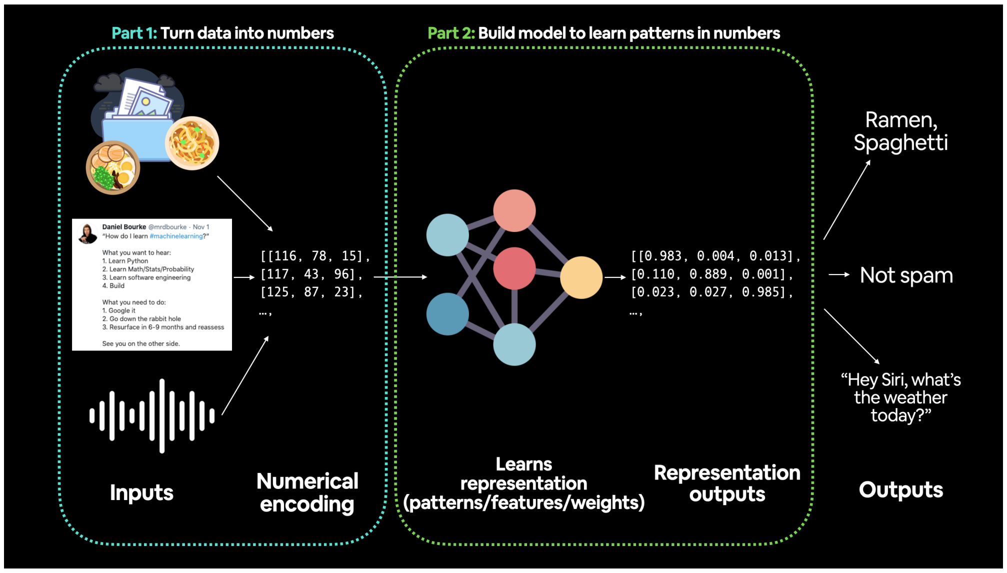

A PyTorch Workflow
1. Get data ready (turn into tensors)
2. Build or pick a pretrained model (to suit your problem)
3. Fit the model to the data and make a prediction
4. Evaluate the model
5. Improve through experimentation
6. Save and reload your trained model
2.1 Pick a loss function & optimizer
2.2 Build a training loop

```python
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

torch.__version__
```

```
'2.2.2+cu121'
```

1.36.1 Data preparing and loading

I want to stress that "data" in machine learning can be almost anything you can imagine


Part 1: Turn data into numbers
Part 2: Build model to learn patterns in numbers
Daniel Bourke @mrdbourke · Nov 1
"How do I learn #machinelearning?"
What you want to hear:
1. Learn Python
2. Learn Math/Stats/Probability
3. Learn software engineering
4. Build
What you need to do:
1. Google it
2. Go down the rabbit hole
3. Resurface in 6-9 months and reassess
See you on the other side.
[[116, 78, 15],
[117, 43, 96],
[125, 87, 23],
…,
[[0.983, 0.004, 0.013],
[0.110, 0.889, 0.001],
[0.023, 0.027, 0.985],
…,
Ramen, Spaghetti
Not spam
"Hey Siri, what's the weather today?"
Inputs
Numerical encoding
Learns representation (patterns/features/weights)
Representation outputs
Outputs

Machine learning is a game of two parts:

- Turn your data, whatever it is, into numbers (a representation).
- Pick or build a model to learn the representation as best as possible.

```
# Create *known* parameters
weight = 0.7
bias = 0.3

# Create data
start = 0
end = 1
step = 0.02
X = torch.arange(start, end, step).unsqueeze(dim=1)
y = weight * X + bias

X[:10], y[:10]
```

```
(tensor([[0.0000],
         [0.0200],
         [0.0400],
         [0.0600],
         [0.0800],
         [0.1000],
         [0.1200],
         [0.1400],
         [0.1600],
         [0.1800]]),
 tensor([[0.3000],
         [0.3140],
         [0.3280],
         [0.3420],
         [0.3560],
         [0.3700],
         [0.3840],
         [0.3980],
         [0.4120],
         [0.4260]]))
```

```
# Create train/test split
train_split = int(0.8 * len(X)) # 80% of data used for training set, 20% for testing
print(train_split)
X_train, y_train = X[:train_split], y[:train_split]
X_test, y_test = X[train_split:], y[train_split:]

len(X_train), len(y_train), len(X_test), len(y_test)
```

```
40
```

```
(40, 40, 10, 10)
```

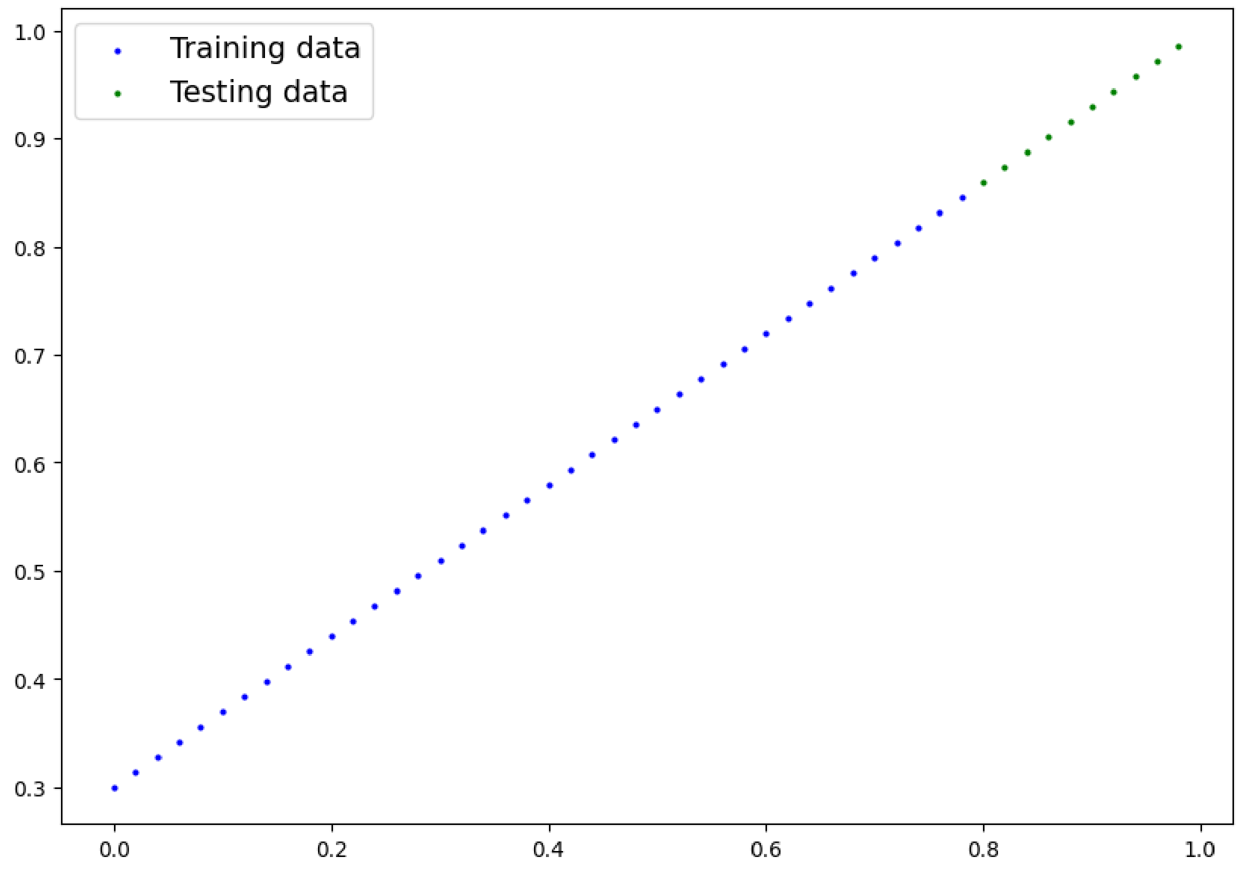
```
def plot_predictions(train_data=X_train,
                     train_labels=y_train,
                     test_data=X_test,
                     test_labels=y_test,
                     predictions=None):
```

(continues on next page)

(continued from previous page)

```
"""
Plots training data, test data and compares predictions.
"""
plt.figure(figsize=(10, 7))

# Plot training data in blue
plt.scatter(train_data, train_labels, c="b", s=4, label="Training data")

# Plot test data in green
plt.scatter(test_data, test_labels, c="g", s=4, label="Testing data")

if predictions is not None:
  # Plot the predictions in red (predictions were made on the test data)
  plt.scatter(test_data, predictions, c="r", s=4, label="Predictions")

# Show the legend
plt.legend(prop={"size": 14});
```

```
plot_predictions()
```



1.36.2 Build model

Now we've got some data, let's build a model to use the blue dots to predict the green dots.

Let's replicate a standard linear regression model using pure PyTorch.

```
# Create a Linear Regression model class
class LinearRegressionModel(nn.Module): # <- almost everything in PyTorch is a nn.Module␣
↪(think of this as neural network lego blocks)
    def __init__(self):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(1, # <- start with random weights (this␣
↪will get adjusted as the model learns)
                                                dtype=torch.float), # <- PyTorch loves␣
↪float32 by default
                                   requires_grad=True) # <- can we update this value␣
↪with gradient descent?)

        self.bias = nn.Parameter(torch.randn(1, # <- start with random bias (this will␣
↪get adjusted as the model learns)
                                            dtype=torch.float), # <- PyTorch loves␣
↪float32 by default
                                requires_grad=True) # <- can we update this value with␣
↪gradient descent?))

    # Forward defines the computation in the model
    def forward(self, x: torch.Tensor) -> torch.Tensor: # <- "x" is the input data (e.g.␣
↪training/testing features)
        return self.weights * x + self.bias # <- this is the linear regression formula␣
↪(y = m*x + b)
```

```
# Set manual seed since nn.Parameter are randomly initialzied
torch.manual_seed(42)

# Create an instance of the model (this is a subclass of nn.Module that contains nn.
↪Parameter(s))
model_0 = LinearRegressionModel()

# Check the nn.Parameter(s) within the nn.Module subclass we created
list(model_0.parameters())
```

```
[Parameter containing:
 tensor([0.3367], requires_grad=True),
 Parameter containing:
 tensor([0.1288], requires_grad=True)]
```

We can also get the state (what the model contains) of the model using .state_dict().

```
# List named parameters
model_0.state_dict()
```

```
OrderedDict([('weights', tensor([0.3367])), ('bias', tensor([0.1288]))])
```

Notice how the values for weights and bias from model_0.state_dict() come out as random float tensors?

This is becuase we initialized them above using torch.randn().

Essentially we want to start from random parameters and get the model to update them towards parameters that fit our data best (the hardcoded weight and bias values we set when creating our straight line data).

1.36.3 torch.inference_mode()

To check this we can pass it the test data X_test to see how closely it predicts y_test.

When we pass data to our model, it'll go through the model's forward() method and produce a result using the computation

```
# Make predictions with model
with torch.inference_mode():
    y_preds = model_0(X_test)

# Note: in older PyTorch code you might also see torch.no_grad()
# with torch.no_grad():
#   y_preds = model_0(X_test)
```

You probably noticed we used torch.inference_mode() as a context manager (that's what the with torch.inference_mode(): is) to make the predictions.

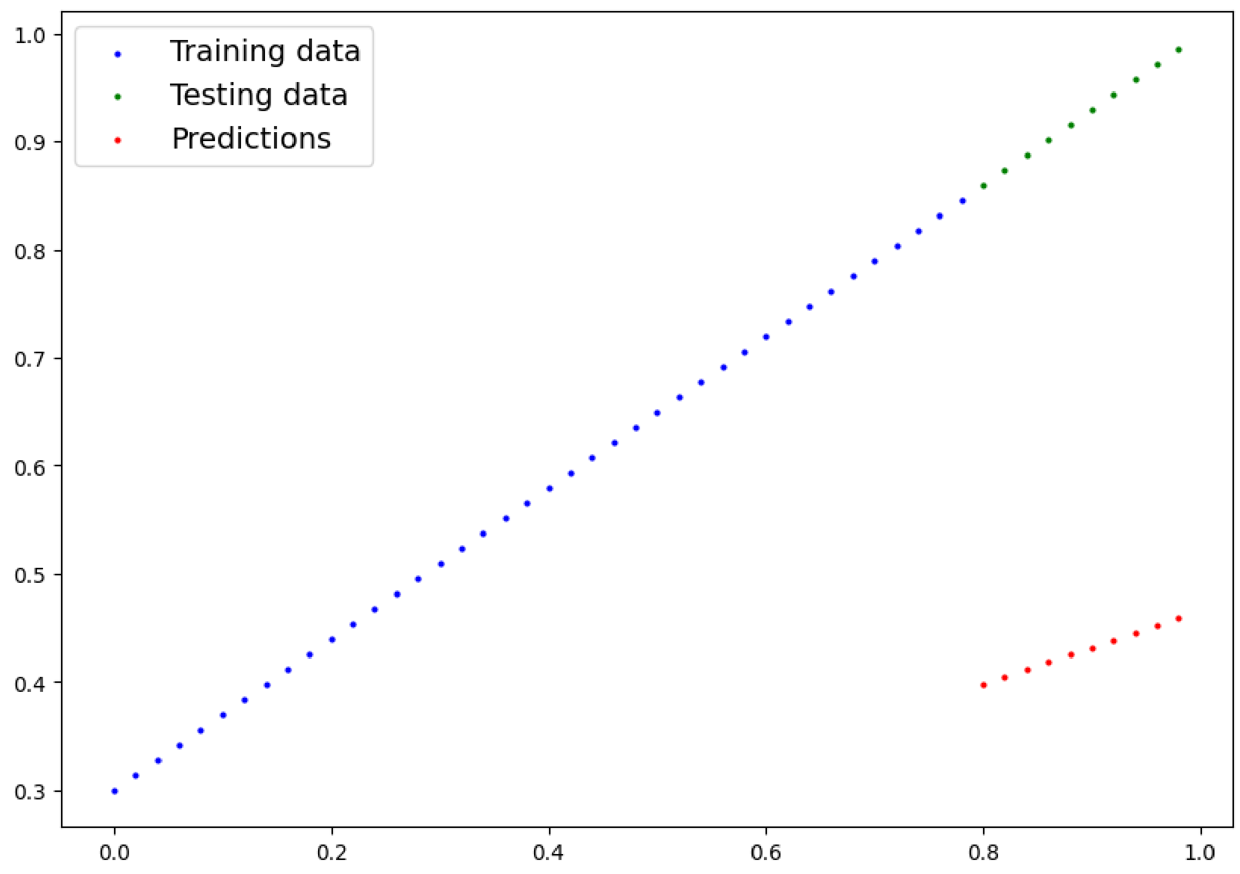
As the name suggests, torch.inference_mode() is used when using a model for inference (making predictions).

torch.inference_mode() turns off a bunch of things (like gradient tracking, which is necessary for training but not for inference) to make forward-passes (data going through the forward() method) faster.

```
# Check the predictions
print(f"Number of testing samples: {len(X_test)}")
print(f"Number of predictions made: {len(y_preds)}")
print(f"Predicted values:\n{y_preds}")
```

```
Number of testing samples: 10
Number of predictions made: 10
Predicted values:
tensor([[0.3982],
        [0.4049],
        [0.4116],
        [0.4184],
        [0.4251],
        [0.4318],
        [0.4386],
        [0.4453],
        [0.4520],
        [0.4588]])
```

```
plot_predictions(predictions=y_preds)
```



```
y_test - y_preds

# This make sense though when you remember our model is just using random parameter␣
↪values to make predictions.
```

```
tensor([[0.4618],
        [0.4691],
        [0.4764],
        [0.4836],
        [0.4909],
        [0.4982],
        [0.5054],
        [0.5127],
        [0.5200],
        [0.5272]])
```

1.36.4 loss function and optimizer in PyTorch

For our model to update its parameters on its own, we'll need to add a few more things to our recipe.

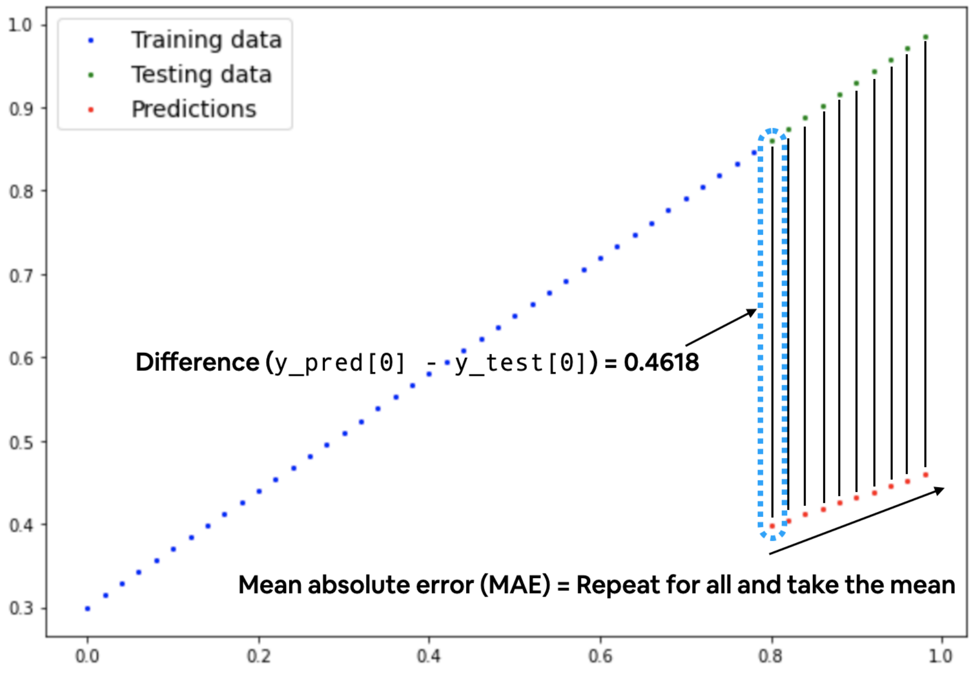
And that's a loss function as well as an optimizer.

Let's create a loss function and an optimizer we can use to help improve our model.

Depending on what kind of problem you're working on will depend on what loss function and what optimizer you use.

However, there are some common values, that are known to work well such as the SGD (stochastic gradient descent) or Adam optimizer. And the MAE (mean absolute error) loss function for regression problems (predicting a number) or binary cross entropy loss function for classification problems (predicting one thing or another).

For our problem, since we're predicting a number, let's use MAE (which is under torch.nn.L1Loss()) in PyTorch as our loss function.



Mean absolute error (MAE, in PyTorch: torch.nn.L1Loss) measures the absolute difference between two points (predictions and labels) and then takes the mean across all examples.

And we'll use SGD, torch.optim.SGD(params, lr) where:

params is the target model parameters you'd like to optimize (e.g. the weights and bias values we randomly set before).

lr is the learning rate you'd like the optimizer to update the parameters at, higher means the optimizer will try larger updates (these can sometimes be too large and the optimizer will fail to work), lower means the optimizer will try smaller updates (these can sometimes be too small and the optimizer will take too long to find the ideal values).

The learning rate is considered a hyperparameter (because it's set by a machine learning engineer). Common starting values for the learning rate are 0.01, 0.001, 0.0001, however, these can also be adjusted over time (this is called learning rate scheduling). Woah, that's a lot, let's see it in code.

```
# Create the loss function
loss_fn = nn.L1Loss() # MAE loss is same as L1Loss

# Create the optimizer
```

(continues on next page)

(continued from previous page)

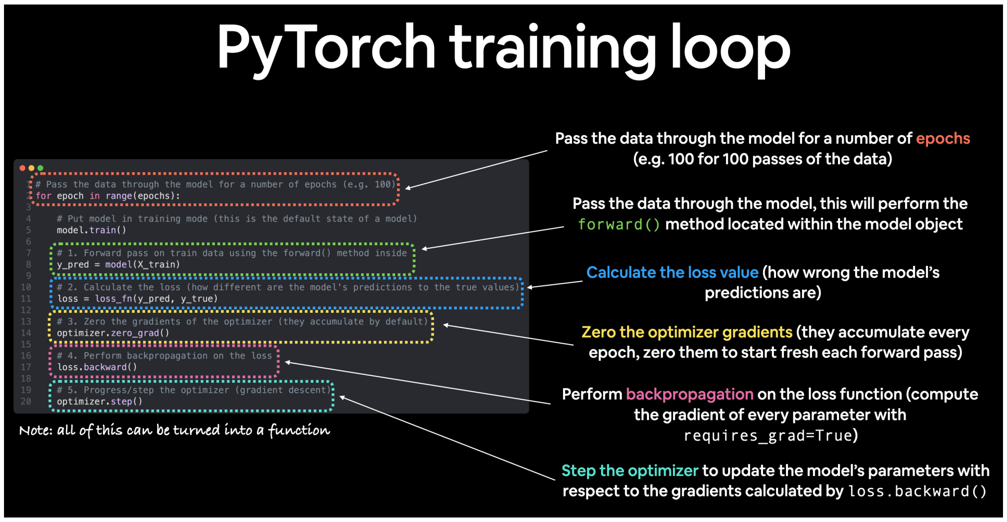
```
optimizer = torch.optim.SGD(params=model_0.parameters(), # parameters of target model to␣
↪optimize
                            lr=0.01) # learning rate (how much the optimizer should␣
↪change parameters at each step, higher=more (less stable), lower=less (might take a␣
↪long time))
```

Creating an optimization loop in PyTorch

The training loop involves the model going through the training data and learning the relationships between the features and labels.

The testing loop involves going through the testing data and evaluating how good the patterns are that the model learned on the training data (the model never see's the testing data during training).

Number	Step name	What does it do?	Code example
1	Forward pass	The model goes through all of the training data once, performing its `forward()` function calculations.	`model(x_train)`
2	Calculate the loss	The model's outputs (predictions) are compared to the ground truth and evaluated to see how wrong they are.	`loss = loss_fn(y_pred, y_train)`
3	Zero gradients	The optimizers gradients are set to zero (they are accumulated by default) so they can be recalculated for the specific training step.	`optimizer.zero_grad()`
4	Perform backpropagation on the loss	Computes the gradient of the loss with respect for every model parameter to be updated (each parameter with `requires_grad=True`). This is known as **backpropagation**, hence "backwards".	`loss.backward()`
5	Update the optimizer (**gradient descent**)	Update the parameters with `requires_grad=True` with respect to the loss gradients in order to improve them.	`optimizer.step()`


PyTorch training loop
Pass the data through the model for a number of epochs (e.g. 100 for 100 passes of the data)
Pass the data through the model, this will perform the forward() method located within the model object
Calculate the loss value (how wrong the model's predictions are)
Zero the optimizer gradients (they accumulate every epoch, zero them to start fresh each forward pass)
Perform backpropagation on the loss function (compute the gradient of every parameter with requires_grad=True)
Step the optimizer to update the model's parameters with respect to the gradients calculated by loss.backward()
Note: all of this can be turned into a function

```
# Pass the data through the model for a number of epochs (e.g. 100)
for epoch in range(epochs):

    # Put model in training mode (this is the default state of a model)
    model.train()

    # 1. Forward pass on train data using the forward() method inside
    y_pred = model(X_train)

    # 2. Calculate the loss (how different are the model's predictions to the true values)
    loss = loss_fn(y_pred, y_true)

    # 3. Zero the gradients of the optimizer (they accumulate by default)
    optimizer.zero_grad()

    # 4. Perform backpropagation on the loss
    loss.backward()

    # 5. Progress/step the optimizer (gradient descent)
    optimizer.step()
```

Note: The above is just one example of how the steps could be ordered or described. With experience you'll find making PyTorch training loops can be quite flexible.

And on the ordering of things, the above is a good default order but you may see slightly different orders. Some rules of thumb:

- Calculate the loss (`loss = ...`) *before* performing backpropagation on it (`loss.backward()`).
- Zero gradients (`optimizer.zero_grad()`) *before* stepping them (`optimizer.step()`).
- Step the optimizer (`optimizer.step()`) *after* performing backpropagation on the loss (`loss.backward()`).

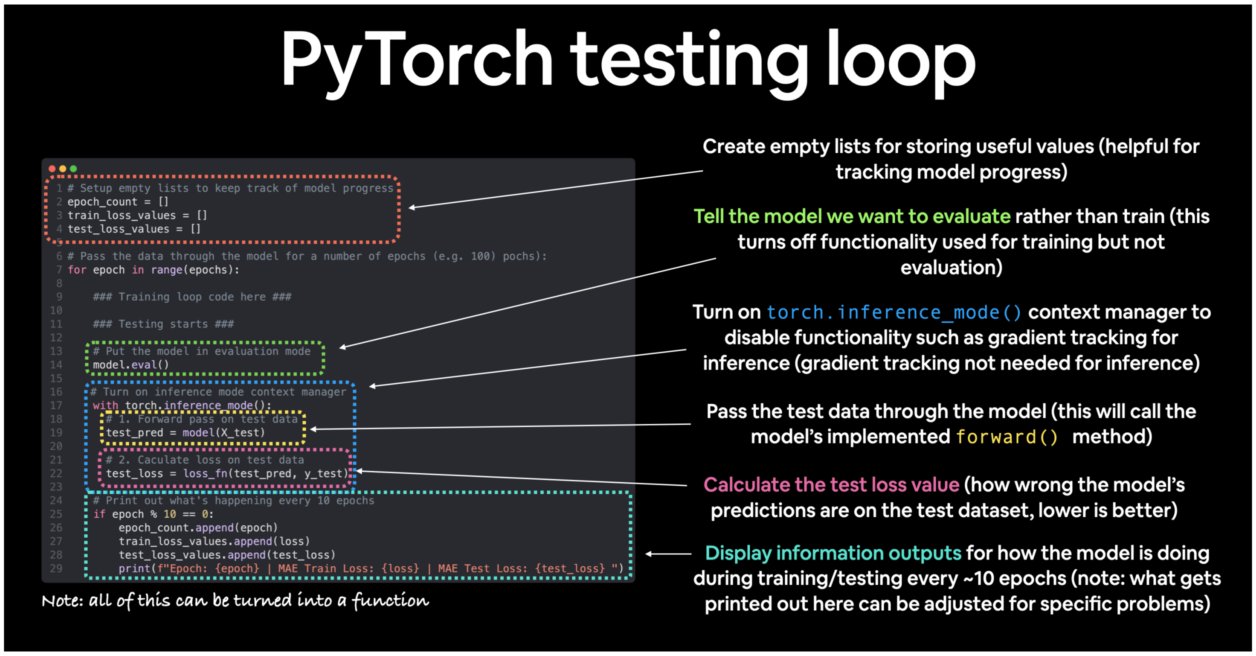
For resources to help understand what's happening behind the scenes with backpropagation and gradient descent, see the extra-curriculum section.

PyTorch testing loop

As for the testing loop (evaluating our model), the typical steps include:

Number	Step name	What does it do?	Code example
1	Forward pass	The model goes through all of the training data once, performing its `forward()` function calculations.	`model(x_test)`
2	Calculate the loss	The model's outputs (predictions) are compared to the ground truth and evaluated to see how wrong they are.	`loss = loss_fn(y_pred, y_test)`
3	Calulate evaluation metrics (optional)	Alongisde the loss value you may want to calculate other evaluation metrics such as accuracy on the test set.	Custom functions

Notice the testing loop doesn't contain performing backpropagation (`loss.backward()`) or stepping the optimizer (`optimizer.step()`), this is because no parameters in the model are being changed during testing, they've already been calculated. For testing, we're only interested in the output of the forward pass through the model.


PyTorch testing loop

```python
# Setup empty lists to keep track of model progress
epoch_count = []
train_loss_values = []
test_loss_values = []

# Pass the data through the model for a number of epochs (e.g. 100) pochs):
for epoch in range(epochs):

    ### Training loop code here ###

    ### Testing starts ###

    # Put the model in evaluation mode
    model.eval()

    # Turn on inference mode context manager
    with torch.inference_mode():
        # 1. Forward pass on test data
        test_pred = model(X_test)

        # 2. Caculate loss on test data
        test_loss = loss_fn(test_pred, y_test)

    # Print out what's happening every 10 epochs
    if epoch % 10 == 0:
        epoch_count.append(epoch)
        train_loss_values.append(loss)
        test_loss_values.append(test_loss)
        print(f"Epoch: {epoch} | MAE Train Loss: {loss} | MAE Test Loss: {test_loss} ")
```

Note: all of this can be turned into a function

Create empty lists for storing useful values (helpful for tracking model progress)

Tell the model we want to evaluate rather than train (this turns off functionality used for training but not evaluation)

Turn on `torch.inference_mode()` context manager to disable functionality such as gradient tracking for inference (gradient tracking not needed for inference)

Pass the test data through the model (this will call the model's implemented `forward()` method)

Calculate the test loss value (how wrong the model's predictions are on the test dataset, lower is better)

Display information outputs for how the model is doing during training/testing every ~10 epochs (note: what gets printed out here can be adjusted for specific problems)

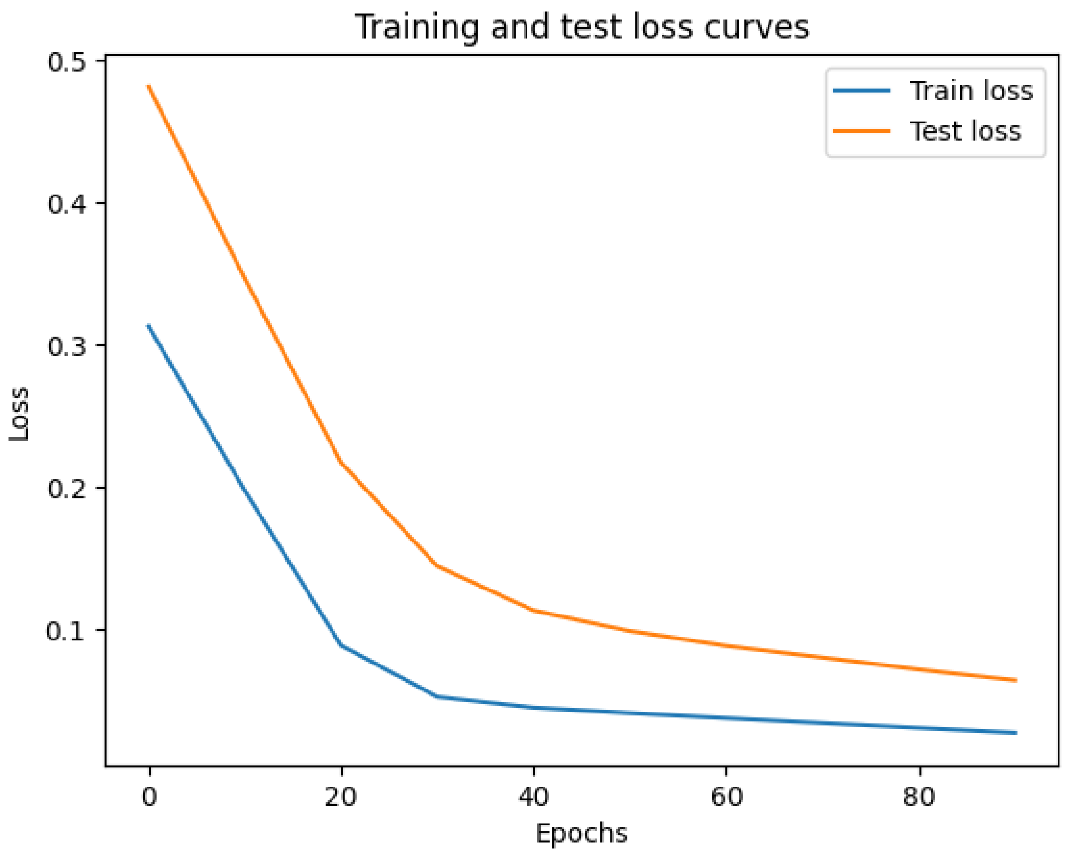
Let's put all of the above together and train our model for 100 **epochs** (forward passes through the data) and we'll evaluate it every 10 epochs.

```
torch.manual_seed(42)

# Set the number of epochs (how many times the model will pass over the training data)
epochs = 100

# Create empty loss lists to track values
train_loss_values = []
test_loss_values = []
epoch_count = []

for epoch in range(epochs):
    ### Training

    # Put model in training mode (this is the default state of a model)
    model_0.train()

    # 1. Forward pass on train data using the forward() method inside
    y_pred = model_0(X_train)
    # print(y_pred)

    # 2. Calculate the loss (how different are our models predictions to the ground␣
↪truth)
    loss = loss_fn(y_pred, y_train)

    # 3. Zero grad of the optimizer
    optimizer.zero_grad()

    # 4. Loss backwards
    loss.backward()

    # 5. Progress the optimizer
    optimizer.step()

    ### Testing

    # Put the model in evaluation mode
    model_0.eval()

    with torch.inference_mode():
      # 1. Forward pass on test data
      test_pred = model_0(X_test)

      # 2. Caculate loss on test data
      test_loss = loss_fn(test_pred, y_test.type(torch.float)) # predictions come in␣
↪torch.float datatype, so comparisons need to be done with tensors of the same type

      # Print out what's happening
      if epoch % 10 == 0:
            epoch_count.append(epoch)
            train_loss_values.append(loss.detach().numpy())
            test_loss_values.append(test_loss.detach().numpy())
            print(f"Epoch: {epoch} | MAE Train Loss: {loss} | MAE Test Loss: {test_loss}
↪")
```

```
Epoch: 0 | MAE Train Loss: 0.31288138031959534 | MAE Test Loss: 0.48106518387794495
Epoch: 10 | MAE Train Loss: 0.1976713240146637 | MAE Test Loss: 0.3463551998138428
Epoch: 20 | MAE Train Loss: 0.08908725529909134 | MAE Test Loss: 0.21729660034179688
Epoch: 30 | MAE Train Loss: 0.053148526698350906 | MAE Test Loss: 0.14464017748832703
Epoch: 40 | MAE Train Loss: 0.04543796554207802 | MAE Test Loss: 0.11360953003168106
Epoch: 50 | MAE Train Loss: 0.04167863354086876 | MAE Test Loss: 0.09919948130846024
Epoch: 60 | MAE Train Loss: 0.03818932920694351 | MAE Test Loss: 0.08886633068323135
Epoch: 70 | MAE Train Loss: 0.03476089984178543 | MAE Test Loss: 0.0805937647819519
Epoch: 80 | MAE Train Loss: 0.03132382780313492 | MAE Test Loss: 0.07232122868299484
Epoch: 90 | MAE Train Loss: 0.02788739837706089 | MAE Test Loss: 0.06473556160926819
```

```
# Plot the loss curves
plt.plot(epoch_count, train_loss_values, label="Train loss")
plt.plot(epoch_count, test_loss_values, label="Test loss")
plt.title("Training and test loss curves")
plt.ylabel("Loss")
plt.xlabel("Epochs")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fc2bd3b0590>
```



```
# Find our model's learned parameters
print("The model learned the following values for weights and bias:")
print(model_0.state_dict())
```

(continues on next page)

(continued from previous page)

```
print("\nAnd the original values for weights and bias are:")
print(f"weights: {weight}, bias: {bias}")
```

```
The model learned the following values for weights and bias:
OrderedDict([('weights', tensor([0.5784])), ('bias', tensor([0.3513]))])

And the original values for weights and bias are:
weights: 0.7, bias: 0.3
```

Wow! How cool is that?

Our model got very close to calculate the exact original values for weight and bias (and it would probably get even closer if we trained it for longer).

Exercise: Try changing the epochs value above to 200, what happens to the loss curves and the weights and bias parameter values of the model?

It'd likely never guess them perfectly (especially when using more complicated datasets) but that's okay, often you can do very cool things with a close approximation.

This is the whole idea of machine learning and deep learning, there are some ideal values that describe our data and rather than figuring them out by hand, we can train a model to figure them out programmatically.

1.36.5 Inference

There are three things to remember when making predictions (also called performing inference) with a PyTorch model:

1. Set the model in evaluation mode (model.eval()).
2. Make the predictions using the inference mode context manager (with torch.inference_mode(): …).
3. All predictions should be made with objects on the same device (e.g. data and model on GPU only or data and model on CPU only).

The first two items make sure all helpful calculations and settings PyTorch uses behind the scenes during training but aren't necessary for inference are turned off (this results in faster computation).

```
model_0.eval()

# 2. Setup the inference mode context manager
with torch.inference_mode():
  # 3. Make sure the calculations are done with the model and data on the same device
  # in our case, we haven't setup device-agnostic code yet so our data and model are
  # on the CPU by default.
  # model_0.to(device)
  # X_test = X_test.to(device)
  y_preds = model_0(X_test)
y_preds
```

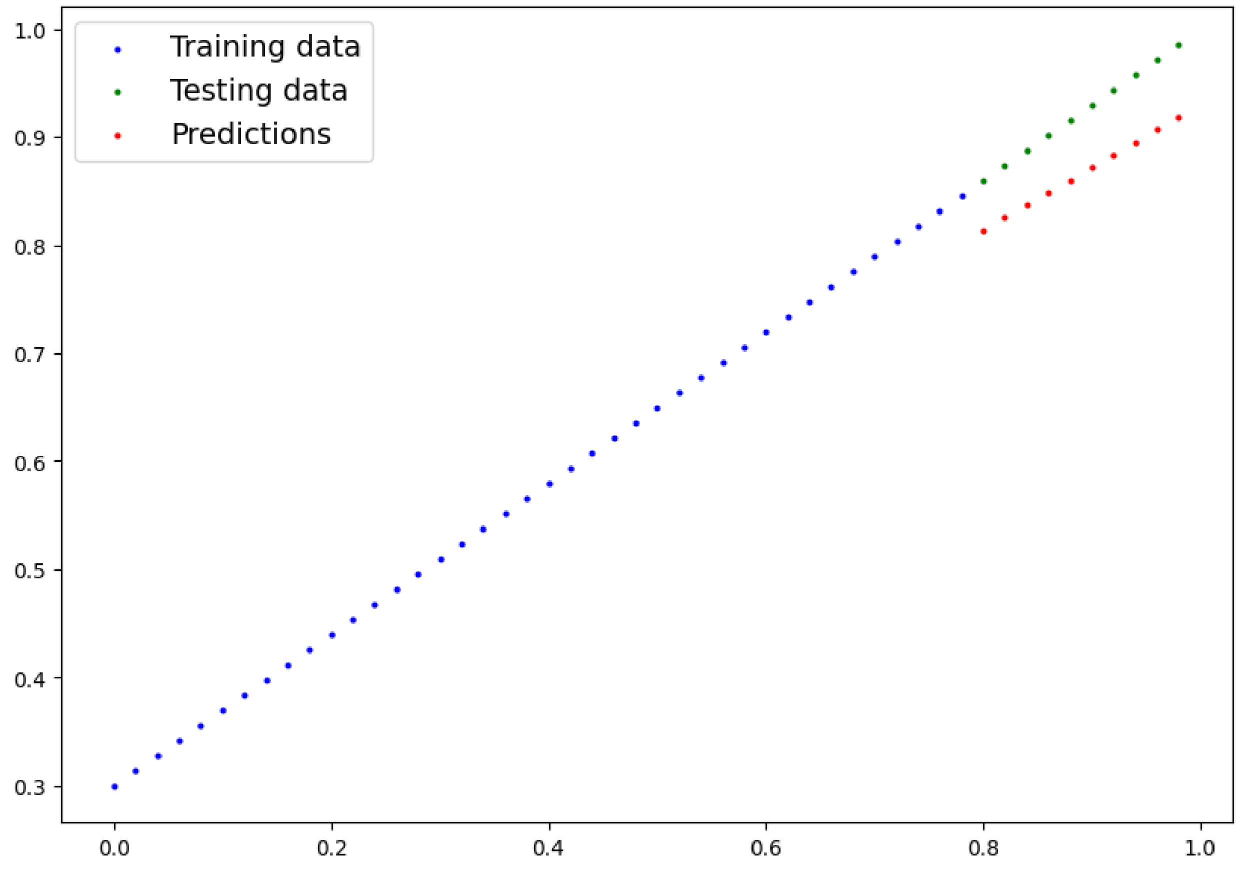
```
tensor([[0.8141],
        [0.8256],
        [0.8372],
        [0.8488],
        [0.8603],
        [0.8719],
```

(continues on next page)

(continued from previous page)

```
        [0.8835],
        [0.8950],
        [0.9066],
        [0.9182]])
```

```
plot_predictions(predictions=y_preds)
```



1.36.6 Saving and loading a PyTorch model

If you've trained a PyTorch model, chances are you'll want to save it and export it somewhere.

As in, you might train it on Google Colab or your local machine with a GPU but you'd like to now export it to some sort of application where others can use it.

Or maybe you'd like to save your progress on a model and come back and load it back later.

For saving and loading models in PyTorch, there are three main methods you should be aware of (all of below have been taken from the PyTorch saving and loading models guide):

PyTorch method	What does it do?
`torch.save`	Saves a serialzed object to disk using Python's `pickle` utility. Models, tensors and various other Python objects like dictionaries can be saved using `torch.save`.
`torch.load`	Uses `pickle`'s unpickling features to deserialize and load pickled Python object files (like models, tensors or dictionaries) into memory. You can also set which device to load the object to (CPU, GPU etc).
`torch.nn.Module.load_state_dict`	Loads a model's parameter dictionary (`model.state_dict()`) using a saved `state_dict()` object.

Note: As stated in Python's `pickle` documentation, the `pickle` module **is not secure**. That means you should only ever unpickle (load) data you trust. That goes for loading PyTorch models as well. Only ever use saved PyTorch models from sources you trust.

Saving a PyTorch model's state_dict()

The recommended way for saving and loading a model for inference (making predictions) is by saving and loading a model's state_dict().

Let's see how we can do that in a few steps:

We'll create a directory for saving models to called models using Python's pathlib module. We'll create a file path to save the model to. We'll call torch.save(obj, f) where obj is the target model's state_dict() and f is the filename of where to save the model.

```
# from pathlib import Path

# # 1. Create models directory
# MODEL_PATH = Path("models")
# MODEL_PATH.mkdir(parents=True, exist_ok=True)

# # 2. Create model save path
# MODEL_NAME = "01_pytorch_workflow_model_0.pt"
# MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME

# # 3. Save the model state dict
# print(f"Saving model to: {MODEL_SAVE_PATH}")
# torch.save(obj=model_0.state_dict(), # only saving the state_dict() only saves the␣
↪models learned parameters
#            f=MODEL_SAVE_PATH)
```

```
# Check the saved file path
# !ls -l models/
```

Loading a saved PyTorch model's state_dict()

Since we've now got a saved model state_dict() at models/01_pytorch_workflow_model_0.pt we can now load it in using torch.nn.Module.load_state_dict(torch.load(f)) where f is the filepath of our saved model state_dict().

Why call torch.load() inside torch.nn.Module.load_state_dict()?

Because we only saved the model's state_dict() which is a dictionary of learned parameters and not the entire model, we first have to load the state_dict() with torch.load() and then pass that state_dict() to a new instance of our model (which is a subclass of nn.Module).

Why not save the entire model?

Saving the entire model rather than just the state_dict() is more intuitive, however, to quote the PyTorch documentation:

The disadvantage of this approach (saving the whole model) is that the serialized data is bound to the specific classes and the exact directory structure used when the model is saved…

Because of this, your code can break in various ways when used in other projects or after refactors.

So instead, we're using the flexible method of saving and loading just the state_dict(), which again is basically a dictionary of model parameters.

Let's test it out by created another instance of LinearRegressionModel(), which is a subclass of torch.nn.Module and will hence have the in-built method load_state_dit().

```
# # Instantiate a new instance of our model (this will be instantiated with random␣
↪weights)
# loaded_model_0 = LinearRegressionModel()

# # Load the state_dict of our saved model (this will update the new instance of our␣
↪model with trained weights)
# loaded_model_0.load_state_dict(torch.load(f=MODEL_SAVE_PATH))
```

```
# 1. Put the loaded model into evaluation mode
# loaded_model_0.eval()

# # 2. Use the inference mode context manager to make predictions
# with torch.inference_mode():
#     loaded_model_preds = loaded_model_0(X_test)
```

```
# Compare previous model predictions with loaded model predictions (these should be the␣
↪same)
# y_preds == loaded_model_preds
```

1.37 PyTorch Neural Network Classification

For example, you might want to:

Problem type	What is it?	Example
Binary classification	Target can be one of two options, e.g. yes or no	Predict whether or not someone has heart disease based on their health parameters.
Multi-class classification	Target can be one of more than two options	Decide whether a photo of is of food, a person or a dog.
Multi-label classification	Target can be assigned more than one option	Predict what categories should be assigned to a Wikipedia article (e.g. mathematics, science & philosohpy).

In this notebook, we're going to work through a couple of different classification problems with PyTorch.

In other words, taking a set of inputs and predicting what class those set of inputs belong to.

1.37.1 Architecture

Before we get into writing code, let's look at the general architecture of a classification neural network.

Hyperparameter	Binary Classification	Multiclass classification
Input layer shape (`in_features`)	Same as number of features (e.g. 5 for age, sex, height, weight, smoking status in heart disease prediction)	Same as binary classification
Hidden layer(s)	Problem specific, minimum = 1, maximum = unlimited	Same as binary classification
Neurons per hidden layer	Problem specific, generally 10 to 512	Same as binary classification
Output layer shape (`out_features`)	1 (one class or the other)	1 per class (e.g. 3 for food, person or dog photo)
Hidden layer activation	Usually ReLU (rectified linear unit) but can be many others	Same as binary classification
Output activation	Sigmoid (`torch.sigmoid` in PyTorch)	Softmax (`torch.softmax` in PyTorch)
Loss function	Binary crossentropy (`torch.nn.BCELoss` in PyTorch)	Cross entropy (`torch.nn.CrossEntropyLoss` in PyTorch)
Optimizer	SGD (stochastic gradient descent), Adam (see `torch.optim` for more options)	Same as binary classification

Of course, this ingredient list of classification neural network components will vary depending on the problem you're working on.

But it's more than enough to get started.

We're going to gets hands-on with this setup throughout this notebook.

1.37.2 Make classification data

We'll use the make_circles() method from Scikit-Learn to generate two circles with different coloured dots.

```
import torch

torch.__version__
```

```
'2.2.2+cu121'
```

```
from sklearn.datasets import make_circles


# Make 1000 samples
n_samples = 1000

# Create circles
X, y = make_circles(n_samples,
                    noise=0.03, # a little bit of noise to the dots
                    random_state=42) # keep random state so we get the same values
```

```
print(f"First 5 X features:\n{X[:5]}")
print(f"\nFirst 5 y labels:\n{y[:5]}")
```

```
First 5 X features:
[[ 0.75424625  0.23148074]
 [-0.75615888  0.15325888]
 [-0.81539193  0.17328203]
 [-0.39373073  0.69288277]
 [ 0.44220765 -0.89672343]]

First 5 y labels:
[1 1 1 1 0]
```

```
# Make DataFrame of circle data
import pandas as pd
circles = pd.DataFrame({"X1": X[:, 0],
    "X2": X[:, 1],
    "label": y
})
circles.head(10)
```

```
         X1        X2  label
0  0.754246  0.231481      1
1 -0.756159  0.153259      1
2 -0.815392  0.173282      1
3 -0.393731  0.692883      1
4  0.442208 -0.896723      0
5 -0.479646  0.676435      1
6 -0.013648  0.803349      1
7  0.771513  0.147760      1
```

(continues on next page)

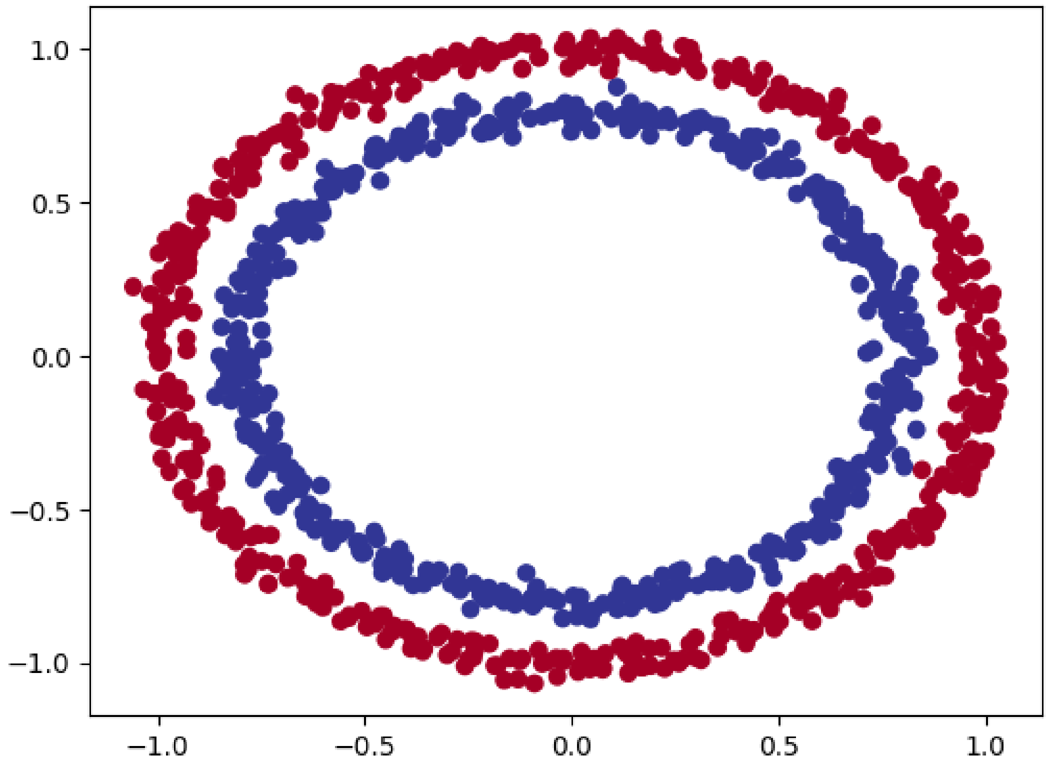
(continued from previous page)

```
8 -0.169322 -0.793456      1
9 -0.121486  1.021509      0
```

```
# Check different labels
circles.label.value_counts()
```

```
label
1    500
0    500
Name: count, dtype: int64
```

```
# Visualize with a plot
import matplotlib.pyplot as plt
plt.scatter(x=X[:, 0],
            y=X[:, 1],
            c=y,
            cmap=plt.cm.RdYlBu);
```



1.37.3 Input and output shapes

One of the most common errors in deep learning is shape errors.

Mismatching the shapes of tensors and tensor operations with result in errors in your models.

We're going to see plenty of these throughout the course.

And there's no surefire way to making sure they won't happen, they will.

What you can do instead is continually familiarize yourself with the shape of the data you're working with.

I like referring to it as input and output shapes.

Ask yourself:

"What shapes are my inputs and what shapes are my outputs?"

```
# Check the shapes of our features and labels
X.shape, y.shape
```

```
((1000, 2), (1000,))
```

```
# View the first example of features and labels
X_sample = X[0]
y_sample = y[0]
print(f"Values for one sample of X: {X_sample} and the same for y: {y_sample}")
print(f"Shapes for one sample of X: {X_sample.shape} and the same for y: {y_sample.shape}
↪")
```

```
Values for one sample of X: [0.75424625 0.23148074] and the same for y: 1
Shapes for one sample of X: (2,) and the same for y: ()
```

```
X = torch.from_numpy(X).type(torch.float)
y = torch.from_numpy(y).type(torch.float)

# View the first five samples
X[:5], y[:5]
```

```
(tensor([[ 0.7542,  0.2315],
         [-0.7562,  0.1533],
         [-0.8154,  0.1733],
         [-0.3937,  0.6929],
         [ 0.4422, -0.8967]]),
 tensor([1., 1., 1., 1., 0.]))
```

```
# Split data into train and test sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2, # 20% test, 80% train
                                                    random_state=42) # make the random␣
↪split reproducible

len(X_train), len(X_test), len(y_train), len(y_test)
```

```
(800, 200, 800, 200)
```

1.37.4 Building a model

We'll break it down into a few parts.

Setting up device agnostic code (so our model can run on CPU or GPU if it's available). Constructing a model by subclassing nn.Module. Defining a loss function and optimizer. Creating a training loop (this'll be in the next section).

```
# Standard PyTorch imports
from torch import nn

# Make device agnostic code
device = "cuda" if torch.cuda.is_available() else "cpu"
device
```

```
'cpu'
```

How about we create a model?

We'll want a model capable of handling our X data as inputs and producing something in the shape of our y data as ouputs.

In other words, given X (features) we want our model to predict y (label).

This setup where you have features and labels is referred to as supervised learning. Because your data is telling your model what the outputs should be given a certain input.

To create such a model it'll need to handle the input and output shapes of X and y.

Remember how I said input and output shapes are important? Here we'll see why.

Let's create a model class that:

Subclasses nn.Module (almost all PyTorch models are subclasses of nn.Module). Creates 2 nn.Linear layers in the constructor capable of handling the input and output shapes of X and y. Defines a forward() method containing the forward pass computation of the model. Instantiates the model class and sends it to the target device.

```
# 1. Construct a model class that subclasses nn.Module
class CircleModelV0(nn.Module):
    def __init__(self):
        super().__init__()
        # 2. Create 2 nn.Linear layers capable of handling X and y input and output␣
↪shapes
        self.layer_1 = nn.Linear(in_features=2, out_features=5) # takes in 2 features␣
↪(X), produces 5 features
        self.layer_2 = nn.Linear(in_features=5, out_features=1) # takes in 5 features,␣
↪produces 1 feature (y)

    # 3. Define a forward method containing the forward pass computation
    def forward(self, x):
        # Return the output of layer_2, a single feature, the same shape as y
        return self.layer_2(self.layer_1(x)) # computation goes through layer_1 first␣
↪then the output of layer_1 goes through layer_2
```

(continues on next page)

(continued from previous page)

```
# 4. Create an instance of the model and send it to target device
model_0 = CircleModelV0().to(device)
model_0
```

```
CircleModelV0(
  (layer_1): Linear(in_features=2, out_features=5, bias=True)
  (layer_2): Linear(in_features=5, out_features=1, bias=True)
)
```

self.layer_1 takes 2 input features in_features=2 and produces 5 output features out_features=5.

This is known as having 5 hidden units or neurons.

This layer turns the input data from having 2 features to 5 features.

Why do this?

This allows the model to learn patterns from 5 numbers rather than just 2 numbers, potentially leading to better outputs.

I say potentially because sometimes it doesn't work.

The number of hidden units you can use in neural network layers is a hyperparameter (a value you can set yourself) and there's no set in stone value you have to use.

Generally more is better but there's also such a thing as too much. The amount you choose will depend on your model type and dataset you're working with.

Since our dataset is small and simple, we'll keep it small.

The only rule with hidden units is that the next layer, in our case, self.layer_2 has to take the same in_features as the previous layer out_features.

That's why self.layer_2 has in_features=5, it takes the out_features=5 from self.layer_1 and performs a linear computation on them, turning them into out_features=1 (the same shape as y).

You can also do the same as above using nn.Sequential.

nn.Sequential performs a forward pass computation of the input data through the layers in the order they appear.

```
# Replicate CircleModelV0 with nn.Sequential
model_0 = nn.Sequential(
    nn.Linear(in_features=2, out_features=5),
    nn.Linear(in_features=5, out_features=1)
).to(device)

model_0
```

```
Sequential(
  (0): Linear(in_features=2, out_features=5, bias=True)
  (1): Linear(in_features=5, out_features=1, bias=True)
)
```

```
# Make predictions with the model
untrained_preds = model_0(X_test.to(device))
print(f"Length of predictions: {len(untrained_preds)}, Shape: {untrained_preds.shape}")
print(f"Length of test samples: {len(y_test)}, Shape: {y_test.shape}")
print(f"\nFirst 10 predictions:\n{untrained_preds[:10]}")
print(f"\nFirst 10 test labels:\n{y_test[:10]}")
```

```
Length of predictions: 200, Shape: torch.Size([200, 1])
Length of test samples: 200, Shape: torch.Size([200])

First 10 predictions:
tensor([[0.7694],
        [0.6636],
        [0.7209],
        [0.7846],
        [0.2033],
        [0.1427],
        [0.2548],
        [0.1315],
        [0.7428],
        [0.6508]], grad_fn=<SliceBackward0>)

First 10 test labels:
tensor([1., 0., 1., 0., 1., 1., 0., 0., 1., 0.])
```

1.37.5 Setup loss function and optimizer

But different problem types require different loss functions.

For example, for a regression problem (predicting a number) you might used mean absolute error (MAE) loss.

And for a binary classification problem (like ours), you'll often use binary cross entropy as the loss function.

However, the same optimizer function can often be used across different problem spaces.

For example, the stochastic gradient descent optimizer (SGD, `torch.optim.SGD()`) can be used for a range of problems, so can too the Adam optimizer (`torch.optim.Adam()`).

Loss function/Optimizer	Problem type	PyTorch Code
Stochastic Gradient Descent (SGD) optimizer	Classification, regression, many others.	`torch.optim.SGD()`
Adam Optimizer	Classification, regression, many others.	`torch.optim.Adam()`
Binary cross entropy loss	Binary classification	`torch.nn.BCELossWithLogits` or `torch.nn.BCELoss`
Cross entropy loss	Mutli-class classification	`torch.nn.CrossEntropyLoss`
Mean absolute error (MAE) or L1 Loss	Regression	`torch.nn.L1Loss`
Mean squared error (MSE) or L2 Loss	Regression	`torch.nn.MSELoss`

Table of various loss functions and optimizers, there are more but these some common ones you'll see.

Since we're working with a binary classification problem, let's use a binary cross entropy loss function.

> **Note:** Recall a **loss function** is what measures how *wrong* your model predictions are, the higher the loss, the worse your model.
>
> Also, PyTorch documentation often refers to loss functions as "loss criterion" or "criterion", these are all different ways of describing the same thing.

PyTorch has two binary cross entropy implementations:

1. torch.nn.BCELoss() - Creates a loss function that measures the binary cross entropy between the target (label) and input (features).
2. torch.nn.BCEWithLogitsLoss() - This is the same as above except it has a sigmoid layer (nn.Sigmoid) built-in (we'll see what this means soon).

Which one should you use?

The documentation for torch.nn.BCEWithLogitsLoss() states that it's more numerically stable than using torch.nn.BCELoss() after a nn.Sigmoid layer.

So generally, implementation 2 is a better option. However for advanced usage, you may want to separate the combination of nn.Sigmoid and torch.nn.BCELoss() but that is beyond the scope of this notebook.

Knowing this, let's create a loss function and an optimizer.

For the optimizer we'll use torch.optim.SGD() to optimize the model parameters with learning rate 0.1.

> **Note:** There's a discussion on the PyTorch forums about the use of nn.BCELoss vs. nn.BCEWithLogitsLoss. It can be confusing at first but as with many things, it becomes easier with practice.

```
# Create a loss function
# loss_fn = nn.BCELoss() # BCELoss = no sigmoid built-in
loss_fn = nn.BCEWithLogitsLoss() # BCEWithLogitsLoss = sigmoid built-in

# Create an optimizer
optimizer = torch.optim.SGD(params=model_0.parameters(),
                            lr=0.1)
```

```
# Now let's also create an evaluation metric.
# Calculate accuracy (a classification metric)
def accuracy_fn(y_true, y_pred):
    correct = torch.eq(y_true, y_pred).sum().item() # torch.eq() calculates where two␣
↪tensors are equal
    acc = (correct / len(y_pred)) * 100
    return acc
```

1.37.6 Train model

Okay, now we've got a loss function and optimizer ready to go, let's train a model.

Going from raw model outputs to predicted labels (logits -> prediction probabilities -> prediction labels) Before we the training loop steps, let's see what comes out of our model during the forward pass (the forward pass is defined by the forward() method).

To do so, let's pass the model some data.

```
# View the frist 5 outputs of the forward pass on the test data
y_logits = model_0(X_test.to(device))[:5]
y_logits
```

```
tensor([[0.7694],
        [0.6636],
        [0.7209],
        [0.7846],
        [0.2033]], grad_fn=<SliceBackward0>)
```

Since our model hasn't been trained, these outputs are basically random.

But *what* are they?

They're the output of our `forward()` method.

Which implements two layers of `nn.Linear()` which internally calls the following equation:

$$\mathbf{y} = x \cdot \mathbf{Weights}^T + \mathbf{bias}$$

The *raw outputs* (unmodified) of this equation ($\mathbf{y}$) and in turn, the raw outputs of our model are often referred to as **logits**.

That's what our model is outputing above when it takes in the input data (x in the equation or `X_test` in the code), logits.

However, these numbers are hard to interpret.

We'd like some numbers that are comparable to our truth labels.

To get our model's raw outputs (logits) into such a form, we can use the sigmoid activation function.

Let's try it out.

```
# Use sigmoid on model logits
y_pred_probs = torch.sigmoid(y_logits)
y_pred_probs
```

```
tensor([[0.6834],
        [0.6601],
        [0.6728],
        [0.6867],
        [0.5507]], grad_fn=<SigmoidBackward0>)
```

They're now in the form of prediction probabilities (I usually refer to these as y_pred_probs), in other words, the values are now how much the model thinks the data point belongs to one class or another.

In our case, since we're dealing with binary classification, our ideal outputs are 0 or 1.

So these values can be viewed as a decision boundary.

The closer to 0, the more the model thinks the sample belongs to class 0, the closer to 1, the more the model thinks the sample belongs to class 1.

```
# Find the predicted labels (round the prediction probabilities)
y_preds = torch.round(y_pred_probs)

# In full
y_pred_labels = torch.round(torch.sigmoid(model_0(X_test.to(device))[:5]))

# Check for equality
print(torch.eq(y_preds.squeeze(), y_pred_labels.squeeze()))

# Get rid of extra dimension
y_preds,y_preds.squeeze()
```

```
tensor([True, True, True, True, True])
```

```
(tensor([[1.],
         [1.],
         [1.],
         [1.],
         [1.]], grad_fn=<RoundBackward0>),
 tensor([1., 1., 1., 1., 1.], grad_fn=<SqueezeBackward0>))
```

```
# Excellent! Now it looks like our model's predictions are in the same form as our truth␣
↪labels (y_test)

y_test[:5]
```

```
tensor([1., 0., 1., 0., 1.])
```

Building a training and testing loop

Let's start by training for 100 epochs and outputing the model's progress every 10 epochs.

```
torch.manual_seed(42)

# Set the number of epochs
epochs = 100

# Put data to target device
X_train, y_train = X_train.to(device), y_train.to(device)
X_test, y_test = X_test.to(device), y_test.to(device)

# Build training and evaluation loop
for epoch in range(epochs):
    ### Training
    model_0.train()

    # 1. Forward pass (model outputs raw logits)
    y_logits = model_0(X_train).squeeze() # squeeze to remove extra `1` dimensions, this␣
↪won't work unless model and data are on same device
    y_pred = torch.round(torch.sigmoid(y_logits)) # turn logits -> pred probs -> pred␣
↪labls

    # 2. Calculate loss/accuracy
    # loss = loss_fn(torch.sigmoid(y_logits), # Using nn.BCELoss you need torch.sigmoid()
    #                y_train)
    loss = loss_fn(y_logits, # Using nn.BCEWithLogitsLoss works with raw logits
                   y_train)
    acc = accuracy_fn(y_true=y_train,
                      y_pred=y_pred)

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backwards
    loss.backward()
```

(continues on next page)

(continued from previous page)

```
    # 5. Optimizer step
    optimizer.step()

    ### Testing
    model_0.eval()
    with torch.inference_mode():
        # 1. Forward pass
        test_logits = model_0(X_test).squeeze()
        test_pred = torch.round(torch.sigmoid(test_logits))
        # 2. Caculate loss/accuracy
        test_loss = loss_fn(test_logits,
                            y_test)
        test_acc = accuracy_fn(y_true=y_test,
                               y_pred=test_pred)

    # Print out what's happening every 10 epochs
    if epoch % 10 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc:.2f}% | Test loss:
↪{test_loss:.5f}, Test acc: {test_acc:.2f}%")
```

```
Epoch: 0 | Loss: 0.73277, Accuracy: 50.00% | Test loss: 0.72974, Test acc: 50.00%
Epoch: 10 | Loss: 0.71041, Accuracy: 56.25% | Test loss: 0.71072, Test acc: 56.00%
Epoch: 20 | Loss: 0.70210, Accuracy: 51.88% | Test loss: 0.70381, Test acc: 54.00%
Epoch: 30 | Loss: 0.69874, Accuracy: 51.12% | Test loss: 0.70108, Test acc: 52.00%
Epoch: 40 | Loss: 0.69717, Accuracy: 51.00% | Test loss: 0.69984, Test acc: 51.50%
Epoch: 50 | Loss: 0.69628, Accuracy: 51.00% | Test loss: 0.69915, Test acc: 50.50%
Epoch: 60 | Loss: 0.69570, Accuracy: 50.38% | Test loss: 0.69867, Test acc: 50.50%
Epoch: 70 | Loss: 0.69527, Accuracy: 50.75% | Test loss: 0.69831, Test acc: 50.00%
Epoch: 80 | Loss: 0.69493, Accuracy: 50.88% | Test loss: 0.69801, Test acc: 49.50%
Epoch: 90 | Loss: 0.69465, Accuracy: 50.12% | Test loss: 0.69775, Test acc: 50.50%
```

The accuracy barely moves above 50% on each data split.

And because we're working with a balanced binary classification problem, it means our model is performing as good as random guessing (with 500 samples of class 0 and class 1 a model predicting class 1 every single time would achieve 50% accuracy).

1.37.7 Evaluate the model

From the metrics it looks like our model is random guessing.

How could we investigate this further?

I've got an idea.

The data explorer's motto!

"Visualize, visualize, visualize!"

Let's make a plot of our model's predictions, the data it's trying to predict on and the decision boundary it's creating for whether something is class 0 or class 1.

To do so, we'll write some code to download and import the helper_functions.py script from the Learn PyTorch for Deep Learning repo.

It contains a helpful function called plot_decision_boundary() which creates a NumPy meshgrid to visually plot the different points where our model is predicting certain classes.

We'll also import plot_predictions() which we wrote in notebook 01 to use later.

In machine learning terms, our model is underfitting, meaning it's not learning predictive patterns from the data.

How could we improve this?

1.37.8 Improving a model

Let's try to fix our model's underfitting problem.

Focusing specifically on the model (not the data), there are a few ways we could do this.

Model improvement technique*	What does it do?
Add more layers	Each layer *potentially* increases the learning capabilities of the model with each layer being able to learn some kind of new pattern in the data, more layers is often referred to as making your neural network *deeper*.
Add more hidden units	Similar to the above, more hidden units per layer means a *potential* increase in learning capabilities of the model, more hidden units is often referred to as making your neural network *wider*.
Fitting for longer (more epochs)	Your model might learn more if it had more opportunities to look at the data.
Changing the activation functions	Some data just can't be fit with only straight lines (like what we've seen), using non-linear activation functions can help with this (hint, hint).
Change the learning rate	Less model specific, but still related, the learning rate of the optimizer decides how much a model should change its parameters each step, too much and the model overcorrects, too little and it doesn't learn enough.
Change the loss function	Again, less model specific but still important, different problems require different loss functions. For example, a binary cross entropy loss function won't work with a multi-class classification problem.
Use transfer learning	Take a pretrained model from a problem domain similar to yours and adjust it to your own problem. We cover transfer learning in notebook 06.

Note: *because you can adjust all of these by hand, they're referred to as **hyperparameters**.

And this is also where machine learning's half art half science comes in, there's no real way to know here what the best combination of values is for your project, best to follow the data scientist's motto of "experiment, experiment, experiment".

Let's see what happens if we add an extra layer to our model, fit for longer (`epochs=1000` instead of `epochs=100`) and increase the number of hidden units from `5` to `10`.

We'll follow the same steps we did above but with a few changed hyperparameters.

```python
class CircleModelV1(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(in_features=2, out_features=10)
        self.layer_2 = nn.Linear(in_features=10, out_features=10) # extra layer
        self.layer_3 = nn.Linear(in_features=10, out_features=1)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x): # note: always make sure forward is spelt correctly!
        # Creating a model like this is the same as below, though below
        # generally benefits from speedups where possible.
        # z = self.layer_1(x)
        # z = self.layer_2(z)
        # z = self.layer_3(z)
        # return z
        return self.layer_3(self.layer_2(self.layer_1(x)))

model_1 = CircleModelV1().to(device)
model_1
```

```
CircleModelV1(
  (layer_1): Linear(in_features=2, out_features=10, bias=True)
  (layer_2): Linear(in_features=10, out_features=10, bias=True)
  (layer_3): Linear(in_features=10, out_features=1, bias=True)
)
```

```
# loss_fn = nn.BCELoss() # Requires sigmoid on input
loss_fn = nn.BCEWithLogitsLoss() # Does not require sigmoid on input
optimizer = torch.optim.SGD(model_1.parameters(), lr=0.1)
```

```
# This time we'll train for longer (epochs=1000 vs epochs=100) and see if it improves our␣
↪model.

torch.manual_seed(42)

epochs = 1000 # Train for longer

# Put data to target device
X_train, y_train = X_train.to(device), y_train.to(device)
X_test, y_test = X_test.to(device), y_test.to(device)

for epoch in range(epochs):
    ### Training
    # 1. Forward pass
    y_logits = model_1(X_train).squeeze()
    y_pred = torch.round(torch.sigmoid(y_logits)) # logits -> predicition probabilities -
↪> prediction labels

    # 2. Calculate loss/accuracy
    loss = loss_fn(y_logits, y_train)
    acc = accuracy_fn(y_true=y_train,
                      y_pred=y_pred)

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backwards
    loss.backward()
```

(continues on next page)

(continued from previous page)

```
    # 5. Optimizer step
    optimizer.step()

    ### Testing
    model_1.eval()
    with torch.inference_mode():
        # 1. Forward pass
        test_logits = model_1(X_test).squeeze()
        test_pred = torch.round(torch.sigmoid(test_logits))
        # 2. Caculate loss/accuracy
        test_loss = loss_fn(test_logits,
                            y_test)
        test_acc = accuracy_fn(y_true=y_test,
                               y_pred=test_pred)

    # Print out what's happening every 10 epochs
    if epoch % 100 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc:.2f}% | Test loss:
↪{test_loss:.5f}, Test acc: {test_acc:.2f}%")
```

```
Epoch: 0 | Loss: 0.69396, Accuracy: 50.88% | Test loss: 0.69261, Test acc: 51.00%
```

```
Epoch: 100 | Loss: 0.69305, Accuracy: 50.38% | Test loss: 0.69379, Test acc: 48.00%
```

```
Epoch: 200 | Loss: 0.69299, Accuracy: 51.12% | Test loss: 0.69437, Test acc: 46.00%
```

```
Epoch: 300 | Loss: 0.69298, Accuracy: 51.62% | Test loss: 0.69458, Test acc: 45.00%
```

```
Epoch: 400 | Loss: 0.69298, Accuracy: 51.12% | Test loss: 0.69465, Test acc: 46.00%
```

```
Epoch: 500 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69467, Test acc: 46.00%
```

```
Epoch: 600 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%
```

```
Epoch: 700 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%
```

```
Epoch: 800 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%
```

```
Epoch: 900 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%
```

What? Our model trained for longer and with an extra layer but it still looks like it didn't learn any patterns better than random guessing.

Our model is still drawing a straight line between the red and blue dots.

The missing piece: non-linearity

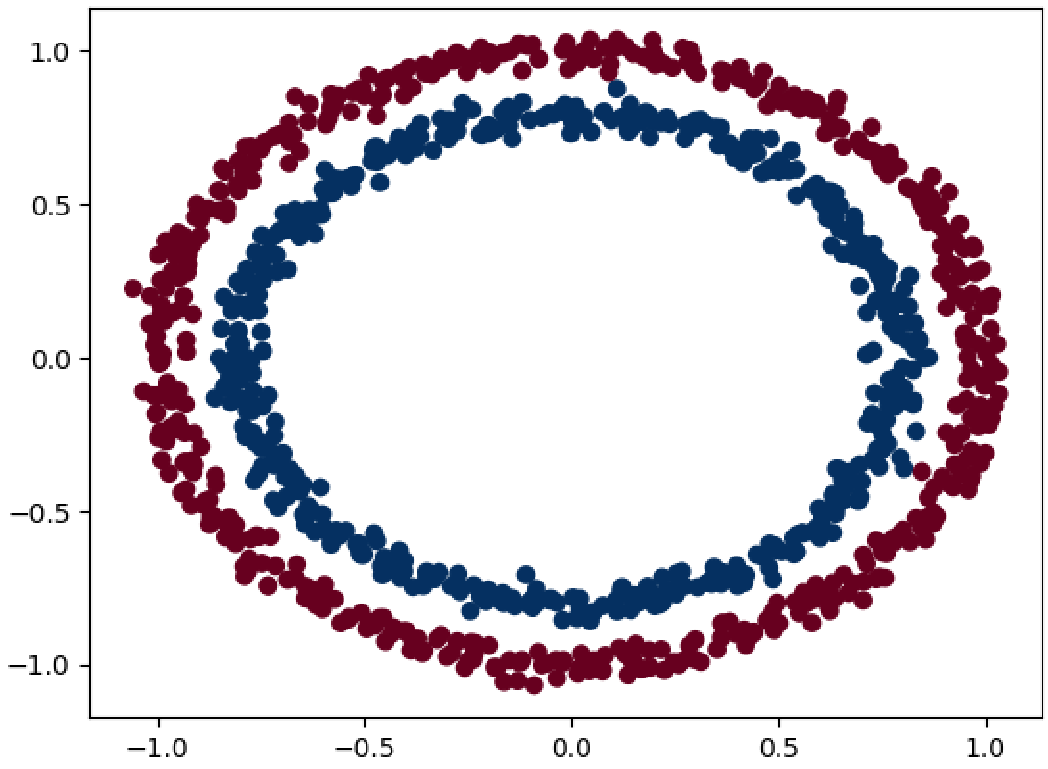
We’ve seen our model can draw straight (linear) lines, thanks to its linear layers.

But how about we give it the capacity to draw non-straight (non-linear) lines?

How?

Let’s find out.

```python
# Make and plot data
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles

n_samples = 1000

X, y = make_circles(n_samples=1000,
    noise=0.03,
    random_state=42,
)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdBu);
```



```python
# Convert to tensors and split into train and test sets
import torch
from sklearn.model_selection import train_test_split

# Turn data into tensors
X = torch.from_numpy(X).type(torch.float)
```

(continues on next page)

(continued from previous page)

```
y = torch.from_numpy(y).type(torch.float)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42
)

X_train[:5], y_train[:5]
```

```
(tensor([[ 0.6579, -0.4651],
         [ 0.6319, -0.7347],
         [-1.0086, -0.1240],
         [-0.9666, -0.2256],
         [-0.1666,  0.7994]]),
 tensor([1., 0., 0., 0., 1.]))
```

1.37.9 Building a model with non-linearity

Now here comes the fun part.

What kind of pattern do you think you could draw with unlimited straight (linear) and non-straight (non-linear) lines?

I bet you could get pretty creative.

So far our neural networks have only been using linear (straight) line functions.

But the data we've been working with is non-linear (circles).

What do you think will happen when we introduce the capability for our model to use non-linear actviation functions?

Well let's see.

PyTorch has a bunch of ready-made non-linear activation functions that do similiar but different things.

One of the most common and best performing is [ReLU](https://en.wikipedia.org/wiki/Rectifier_(neural_networks) (rectified linear-unit, torch.nn.ReLU()).

Rather than talk about it, let's put it in our neural network between the hidden layers in the forward pass and see what happens.

```
# Build model with non-linear activation function
from torch import nn
class CircleModelV2(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(in_features=2, out_features=10)
        self.layer_2 = nn.Linear(in_features=10, out_features=10)
        self.layer_3 = nn.Linear(in_features=10, out_features=1)
        self.relu = nn.ReLU() # <- add in ReLU activation function
        # Can also put sigmoid in the model
        # This would mean you don't need to use it on the predictions
        # self.sigmoid = nn.Sigmoid()

```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
      # Intersperse the ReLU activation function between layers
       return self.layer_3(self.relu(self.layer_2(self.relu(self.layer_1(x)))))

model_3 = CircleModelV2().to(device)
print(model_3)
```

```
CircleModelV2(
  (layer_1): Linear(in_features=2, out_features=10, bias=True)
  (layer_2): Linear(in_features=10, out_features=10, bias=True)
  (layer_3): Linear(in_features=10, out_features=1, bias=True)
  (relu): ReLU()
)
```

```
# Setup loss and optimizer
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.SGD(model_3.parameters(), lr=0.1)
```

```
# Fit the model
torch.manual_seed(42)
epochs = 1000

# Put all data on target device
X_train, y_train = X_train.to(device), y_train.to(device)
X_test, y_test = X_test.to(device), y_test.to(device)

for epoch in range(epochs):
    # 1. Forward pass
    y_logits = model_3(X_train).squeeze()
    y_pred = torch.round(torch.sigmoid(y_logits)) # logits -> prediction probabilities ->
↪ prediction labels

    # 2. Calculate loss and accuracy
    loss = loss_fn(y_logits, y_train) # BCEWithLogitsLoss calculates loss using logits
    acc = accuracy_fn(y_true=y_train,
                      y_pred=y_pred)

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backward
    loss.backward()

    # 5. Optimizer step
    optimizer.step()

    ### Testing
    model_3.eval()
    with torch.inference_mode():
      # 1. Forward pass
      test_logits = model_3(X_test).squeeze()
```

(continues on next page)

(continued from previous page)

```
      test_pred = torch.round(torch.sigmoid(test_logits)) # logits -> prediction␣
↪probabilities -> prediction labels
      # 2. Calcuate loss and accuracy
      test_loss = loss_fn(test_logits, y_test)
      test_acc = accuracy_fn(y_true=y_test,
                             y_pred=test_pred)

    # Print out what's happening
    if epoch % 100 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc:.2f}% | Test Loss:
↪{test_loss:.5f}, Test Accuracy: {test_acc:.2f}%")
```

```
Epoch: 0 | Loss: 0.69295, Accuracy: 50.00% | Test Loss: 0.69319, Test Accuracy: 50.00%
```

```
Epoch: 100 | Loss: 0.69115, Accuracy: 52.88% | Test Loss: 0.69102, Test Accuracy: 52.50%
```

```
Epoch: 200 | Loss: 0.68977, Accuracy: 53.37% | Test Loss: 0.68940, Test Accuracy: 55.00%
```

```
Epoch: 300 | Loss: 0.68795, Accuracy: 53.00% | Test Loss: 0.68723, Test Accuracy: 56.00%
```

```
Epoch: 400 | Loss: 0.68517, Accuracy: 52.75% | Test Loss: 0.68411, Test Accuracy: 56.50%
```

```
Epoch: 500 | Loss: 0.68102, Accuracy: 52.75% | Test Loss: 0.67941, Test Accuracy: 56.50%
```

```
Epoch: 600 | Loss: 0.67515, Accuracy: 54.50% | Test Loss: 0.67285, Test Accuracy: 56.00%
```

```
Epoch: 700 | Loss: 0.66659, Accuracy: 58.38% | Test Loss: 0.66322, Test Accuracy: 59.00%
```

```
Epoch: 800 | Loss: 0.65160, Accuracy: 64.00% | Test Loss: 0.64757, Test Accuracy: 67.50%
```

```
Epoch: 900 | Loss: 0.62362, Accuracy: 74.00% | Test Loss: 0.62145, Test Accuracy: 79.00%
```

Ho ho! That's looking far better!

Evaluating a model trained with non-linear activation functions

Remember how our circle data is non-linear? Well, let's see how our models predictions look now the model's been trained with non-linear activation functions.

```
# Make predictions
model_3.eval()
with torch.inference_mode():
    y_preds = torch.round(torch.sigmoid(model_3(X_test))).squeeze()
y_preds[:10], y[:10] # want preds in same format as truth labels
```

```
(tensor([1., 0., 1., 0., 0., 1., 0., 0., 1., 0.]),
 tensor([1., 1., 1., 1., 0., 1., 1., 1., 1., 0.]))
```

1.37.10 Multi-class PyTorch model

We've covered a fair bit.

But now let's put it all together using a multi-class classification problem.

Recall a binary classification problem deals with classifying something as one of two options (e.g. a photo as a cat photo or a dog photo) where as a multi-class classification problem deals with classifying something from a list of more than two options (e.g. classifying a photo as a cat a dog or a chicken).

Creating multi-class classification data

To begin a multi-class classification problem, let's create some multi-class data.

To do so, we can leverage Scikit-Learn's make_blobs() method.

This method will create however many classes (using the centers parameter) we want.

Specifically, let's do the following:

Create some multi-class data with make_blobs(). Turn the data into tensors (the default of make_blobs() is to use NumPy arrays). Split the data into training and test sets using train_test_split(). Visualize the data.

```
# Import dependencies
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split

# Set the hyperparameters for data creation
NUM_CLASSES = 4
NUM_FEATURES = 2
RANDOM_SEED = 42

# 1. Create multi-class data
X_blob, y_blob = make_blobs(n_samples=1000,
    n_features=NUM_FEATURES, # X features
    centers=NUM_CLASSES, # y labels
    cluster_std=1.5, # give the clusters a little shake up (try changing this to 1.0,␣
↪the default)
    random_state=RANDOM_SEED
)

# 2. Turn data into tensors
X_blob = torch.from_numpy(X_blob).type(torch.float)
y_blob = torch.from_numpy(y_blob).type(torch.LongTensor)
print(X_blob[:5], y_blob[:5])

# 3. Split into train and test sets
X_blob_train, X_blob_test, y_blob_train, y_blob_test = train_test_split(X_blob,
    y_blob,
    test_size=0.2,
    random_state=RANDOM_SEED
)

# 4. Plot data
```

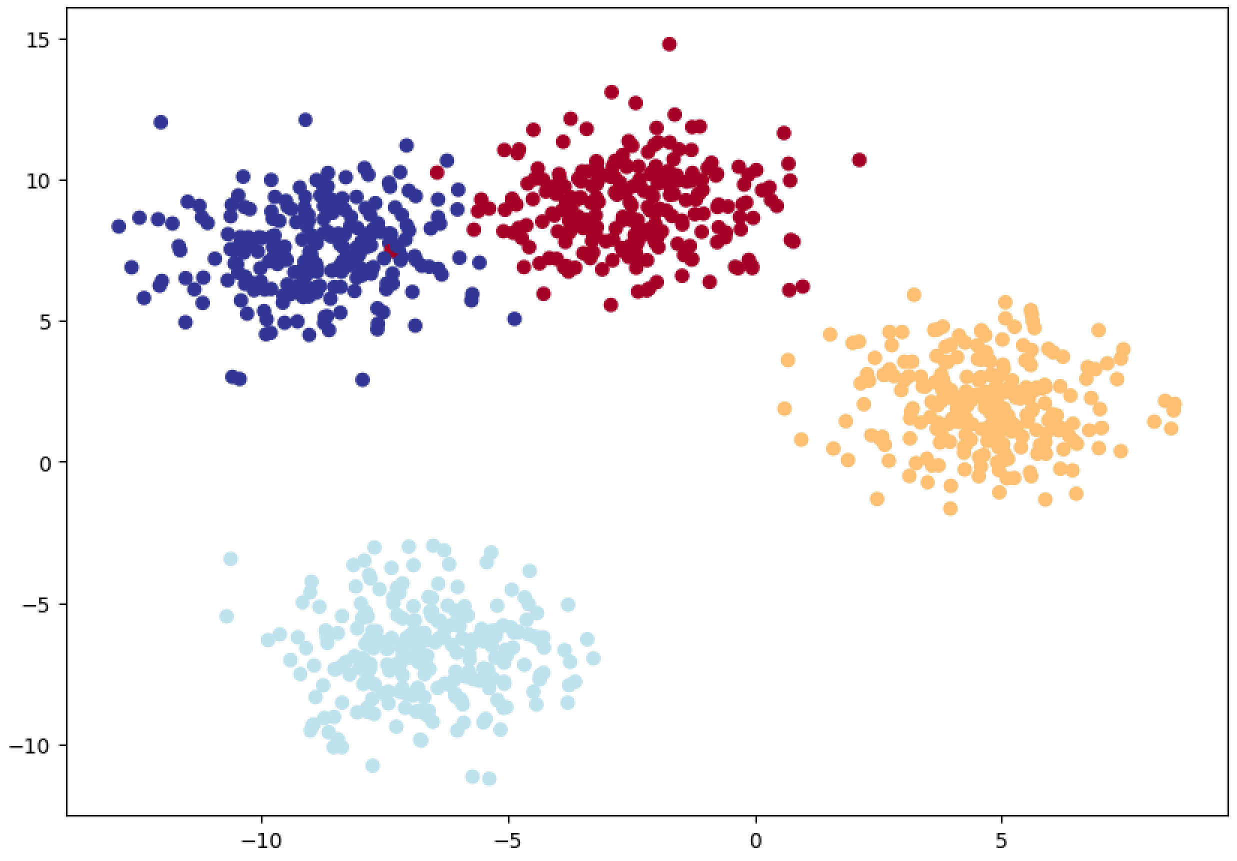
(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(10, 7))
plt.scatter(X_blob[:, 0], X_blob[:, 1], c=y_blob, cmap=plt.cm.RdYlBu)
```

```
tensor([[-8.4134,  6.9352],
        [-5.7665, -6.4312],
        [-6.0421, -6.7661],
        [ 3.9508,  0.6984],
        [ 4.2505, -0.2815]]) tensor([3, 2, 2, 1, 1])
```

```
<matplotlib.collections.PathCollection at 0x7f0be7b037d0>
```



Multi-class classification model

We've created a few models in PyTorch so far.

You might also be starting to get an idea of how flexible neural networks are.

How about we build one similar to model_3 but this still capable of handling multi-class data?

To do so, let's create a subclass of nn.Module that takes in three hyperparameters:

- input_features - the number of X features coming into the model.
- output_features - the ideal numbers of output features we'd like (this will be equivalent to NUM_CLASSES or the number of classes in your multi-class classification problem).

- hidden_units - the number of hidden neurons we'd like each hidden layer to use.
- Since we're putting things together, let's setup some device agnostic code (we don't have to do this again in the same notebook, it's only a reminder).

Then we'll create the model class using the hyperparameters above.

```
# Build model
class BlobModel(nn.Module):
    def __init__(self, input_features, output_features, hidden_units=8):
        """Initializes all required hyperparameters for a multi-class classification␣
↪model.

        Args:
            input_features (int): Number of input features to the model.
            out_features (int): Number of output features of the model
              (how many classes there are).
            hidden_units (int): Number of hidden units between layers, default 8.
        """
        super().__init__()
        self.linear_layer_stack = nn.Sequential(
            nn.Linear(in_features=input_features, out_features=hidden_units),
            # nn.ReLU(), # <- does our dataset require non-linear layers? (try␣
↪uncommenting and see if the results change)
            nn.Linear(in_features=hidden_units, out_features=hidden_units),
            # nn.ReLU(), # <- does our dataset require non-linear layers? (try␣
↪uncommenting and see if the results change)
            nn.Linear(in_features=hidden_units, out_features=output_features), # how␣
↪many classes are there?
        )

    def forward(self, x):
        return self.linear_layer_stack(x)

# Create an instance of BlobModel and send it to the target device
model_4 = BlobModel(input_features=NUM_FEATURES,
                    output_features=NUM_CLASSES,
                    hidden_units=8).to(device)
model_4
```

```
BlobModel(
  (linear_layer_stack): Sequential(
    (0): Linear(in_features=2, out_features=8, bias=True)
    (1): Linear(in_features=8, out_features=8, bias=True)
    (2): Linear(in_features=8, out_features=4, bias=True)
  )
)
```

loss function and optimizer

Since we're working on a multi-class classification problem, we'll use the nn.CrossEntropyLoss() method as our loss function.

And we'll stick with using SGD with a learning rate of 0.1 for optimizing our model_4 parameters.

```
# Create loss and optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_4.parameters(),
                            lr=0.1) # exercise: try changing the learning rate here and␣
↪seeing what happens to the model's performance
```

Getting prediction probabilities

Alright, we've got a loss function and optimizer ready, and we're ready to train our model but before we do let's do a single forward pass with our model to see if it works.

```
# Testing the input features calculation

w_ = torch.rand(3, 2)
b_ = torch.rand(3)

d = nn.Linear(in_features=2, out_features=3)(X_blob_train[:5])
formula = torch.matmul(X_blob_train[:5], w_.T) + b_
formula1 = X_blob_train[:5] @ w_.T + b_

d , w_ , b_ , formula, formula1, X_blob_train[:5], w_.T , X_blob_train[:5].shape, w_.T.
↪shape
```

```
(tensor([[-0.6177, -2.3760,  1.3912],
         [ 7.0950,  0.8435,  2.5291],
         [ 0.7714,  6.1724, -5.1197],
         [ 1.7382,  4.3139, -3.0706],
         [ 6.2979,  1.4964,  1.5808]], grad_fn=<AddmmBackward0>),
 tensor([[0.0766, 0.8460],
         [0.3624, 0.3083],
         [0.0850, 0.0029]]),
 tensor([0.6431, 0.3908, 0.6947]),
 tensor([[ 3.8275,  3.2374,  1.1326],
         [ 8.5012,  2.3767,  0.4995],
         [-7.6581, -5.4860, -0.0560],
         [-3.5728, -3.1615,  0.1697],
         [ 6.4933,  1.4058,  0.4320]]),
 tensor([[ 3.8275,  3.2374,  1.1326],
         [ 8.5012,  2.3767,  0.4995],
         [-7.6581, -5.4860, -0.0560],
         [-3.5728, -3.1615,  0.1697],
         [ 6.4933,  1.4058,  0.4320]]),
 tensor([[ 5.0405,  3.3076],
         [-2.6249,  9.5260],
         [-8.5240, -9.0402],
         [-6.0262, -4.4375],
```

(continues on next page)

(continued from previous page)

```
        [-3.3397,  7.2175]]),
 tensor([[0.0766, 0.3624, 0.0850],
        [0.8460, 0.3083, 0.0029]]),
 torch.Size([5, 2]),
 torch.Size([2, 3]))
```

```
# Perform a single forward pass on the data (we'll need to put it to the target device␣
↪for it to work)
model_4(X_blob_train.to(device))[:5]
```

```
tensor([[-1.2711, -0.6494, -1.4740, -0.7044],
        [ 0.2210, -1.5439,  0.0420,  1.1531],
        [ 2.8698,  0.9143,  3.3169,  1.4027],
        [ 1.9576,  0.3125,  2.2244,  1.1324],
        [ 0.5458, -1.2381,  0.4441,  1.1804]], grad_fn=<SliceBackward0>)
```

```
# How many elements in a single prediction sample?
model_4(X_blob_train.to(device))[0].shape, NUM_CLASSES
```

```
(torch.Size([4]), 4)
```

Wonderful, our model is predicting one value for each class that we have.

Do you remember what the raw outputs of our model are called?

Hint: it rhymes with "frog splits" (no animals were harmed in the creation of these materials).

If you guessed logits, you'd be correct.

So right now our model is outputing logits but what if we wanted to figure out exactly which label is was giving the sample?

As in, how do we go from logits -> prediction probabilities -> prediction labels just like we did with the binary classification problem?

That's where the softmax activation function comes into play.

The softmax function calculates the probability of each prediction class being the actual predicted class compared to all other possible classes.

If this doesn't make sense, let's see in code.

```
# Make prediction logits with model
y_logits = model_4(X_blob_test.to(device))

# Perform softmax calculation on logits across dimension 1 to get prediction␣
↪probabilities
y_pred_probs = torch.softmax(y_logits, dim=1)
print(y_logits[:5])
print(y_pred_probs[:5])
```

```
tensor([[-1.2549, -0.8112, -1.4795, -0.5696],
        [ 1.7168, -1.2270,  1.7367,  2.1010],
        [ 2.2400,  0.7714,  2.6020,  1.0107],
        [-0.7993, -0.3723, -0.9138, -0.5388],
```

(continues on next page)

(continued from previous page)

```
        [-0.4332, -1.6117, -0.6891,  0.6852]], grad_fn=<SliceBackward0>)
tensor([[0.1872, 0.2918, 0.1495, 0.3715],
        [0.2824, 0.0149, 0.2881, 0.4147],
        [0.3380, 0.0778, 0.4854, 0.0989],
        [0.2118, 0.3246, 0.1889, 0.2748],
        [0.1945, 0.0598, 0.1506, 0.5951]], grad_fn=<SliceBackward0>)
```

Hmm, what's happened here?

It may still look like the outputs of the softmax function are jumbled numbers (and they are, since our model hasn't been trained and is predicting using random patterns) but there's a very specific thing different about each sample.

After passing the logits through the softmax function, each individual sample now adds to 1 (or very close to).

Let's check.

```
# Sum the first sample output of the softmax activation function
torch.sum(y_pred_probs[0])
```

```
tensor(1., grad_fn=<SumBackward0>)
```

These prediction probablities are essentially saying how much the model thinks the target X sample (the input) maps to each class.

Since there's one value for each class in y_pred_probs, the index of the highest value is the class the model thinks the specific data sample most belongs to.

We can check which index has the highest value using torch.argmax().

```
# Which class does the model think is *most* likely at the index 0 sample?
print(y_pred_probs[0])
print(torch.argmax(y_pred_probs[0]))
```

```
tensor([0.1872, 0.2918, 0.1495, 0.3715], grad_fn=<SelectBackward0>)
tensor(3)
```

You can see the output of torch.argmax() returns 3, so for the features (X) of the sample at index 0, the model is predicting that the most likely class value (y) is 3.

Of course, right now this is just random guessing so it's got a 25% chance of being right (since there's four classes). But we can improve those chances by training the model.

Note: To summarize the above, a model's raw output is referred to as logits.

For a multi-class classification problem, to turn the logits into prediction probabilities, you use the softmax activation function (torch.softmax).

The index of the value with the highest prediction probability is the class number the model thinks is most likely given the input features for that sample (although this is a prediction, it doesn't mean it will be correct).

Creating a training and testing loop

Alright, now we've got all of the preparation steps out of the way, let's write a training and testing loop to improve and evaluation our model.

We've done many of these steps before so much of this will be practice.

The only difference is that we'll be adjusting the steps to turn the model outputs (logits) to prediction probabilities (using the softmax activation function) and then to prediction labels (by taking the argmax of the output of the softmax activation function).

Let's train the model for epochs=100 and evaluate it every 10 epochs.

```
# Fit the model
torch.manual_seed(42)

# Set number of epochs
epochs = 100

# Put data to target device
X_blob_train, y_blob_train = X_blob_train.to(device), y_blob_train.to(device)
X_blob_test, y_blob_test = X_blob_test.to(device), y_blob_test.to(device)

for epoch in range(epochs):
    ### Training
    model_4.train()

    # 1. Forward pass
    y_logits = model_4(X_blob_train) # model outputs raw logits
    y_pred = torch.softmax(y_logits, dim=1).argmax(dim=1) # go from logits -> prediction␣
↪probabilities -> prediction labels
    # print(y_logits)
    # 2. Calculate loss and accuracy
    loss = loss_fn(y_logits, y_blob_train)
    acc = accuracy_fn(y_true=y_blob_train,
                      y_pred=y_pred)

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backwards
    loss.backward()

    # 5. Optimizer step
    optimizer.step()

    ### Testing
    model_4.eval()
    with torch.inference_mode():
      # 1. Forward pass
      test_logits = model_4(X_blob_test)
      test_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
      # 2. Calculate test loss and accuracy
      test_loss = loss_fn(test_logits, y_blob_test)
      test_acc = accuracy_fn(y_true=y_blob_test,
```

(continues on next page)

(continued from previous page)

```
                                y_pred=test_pred)

    # Print out what's happening
    if epoch % 10 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Acc: {acc:.2f}% | Test Loss: {test_
↪loss:.5f}, Test Acc: {test_acc:.2f}%")
```

```
Epoch: 0 | Loss: 1.04324, Acc: 65.50% | Test Loss: 0.57861, Test Acc: 95.50%
Epoch: 10 | Loss: 0.14398, Acc: 99.12% | Test Loss: 0.13037, Test Acc: 99.00%
Epoch: 20 | Loss: 0.08062, Acc: 99.12% | Test Loss: 0.07216, Test Acc: 99.50%
Epoch: 30 | Loss: 0.05924, Acc: 99.12% | Test Loss: 0.05133, Test Acc: 99.50%
Epoch: 40 | Loss: 0.04892, Acc: 99.00% | Test Loss: 0.04098, Test Acc: 99.50%
Epoch: 50 | Loss: 0.04295, Acc: 99.00% | Test Loss: 0.03486, Test Acc: 99.50%
Epoch: 60 | Loss: 0.03910, Acc: 99.00% | Test Loss: 0.03083, Test Acc: 99.50%
Epoch: 70 | Loss: 0.03643, Acc: 99.00% | Test Loss: 0.02799, Test Acc: 99.50%
Epoch: 80 | Loss: 0.03448, Acc: 99.00% | Test Loss: 0.02587, Test Acc: 99.50%
Epoch: 90 | Loss: 0.03300, Acc: 99.12% | Test Loss: 0.02423, Test Acc: 99.50%
```

evaluating predictions

It looks like our trained model is performaning pretty well.

But to make sure of this, let’s make some predictions and visualize them.

```
# Make predictions
model_4.eval()
with torch.inference_mode():
    y_logits = model_4(X_blob_test)

# View the first 10 predictions
y_logits[:10]
```

```
tensor([[  4.3377,  10.3539, -14.8948,  -9.7642],
        [  5.0142, -12.0371,   3.3860,  10.6699],
        [ -5.5885, -13.3448,  20.9894,  12.7711],
        [  1.8400,   7.5599,  -8.6016,  -6.9942],
        [  8.0727,   3.2906, -14.5998,  -3.6186],
        [  5.5844, -14.9521,   5.0168,  13.2891],
        [ -5.9739, -10.1913,  18.8655,   9.9179],
        [  7.0755,  -0.7601,  -9.5531,   0.1736],
        [ -5.5919, -18.5990,  25.5310,  17.5799],
        [  7.3142,   0.7197, -11.2017,  -1.2011]])
```

```
# Turn predicted logits in prediction probabilities
y_pred_probs = torch.softmax(y_logits, dim=1)

# Turn prediction probabilities into prediction labels
y_preds = y_pred_probs.argmax(dim=1)

# Compare first 10 model preds and test labels
```

(continues on next page)

(continued from previous page)

```
print(f"Predictions: {y_preds[:10]}\nLabels: {y_blob_test[:10]}")
print(f"Test accuracy: {accuracy_fn(y_true=y_blob_test, y_pred=y_preds)}%")
```

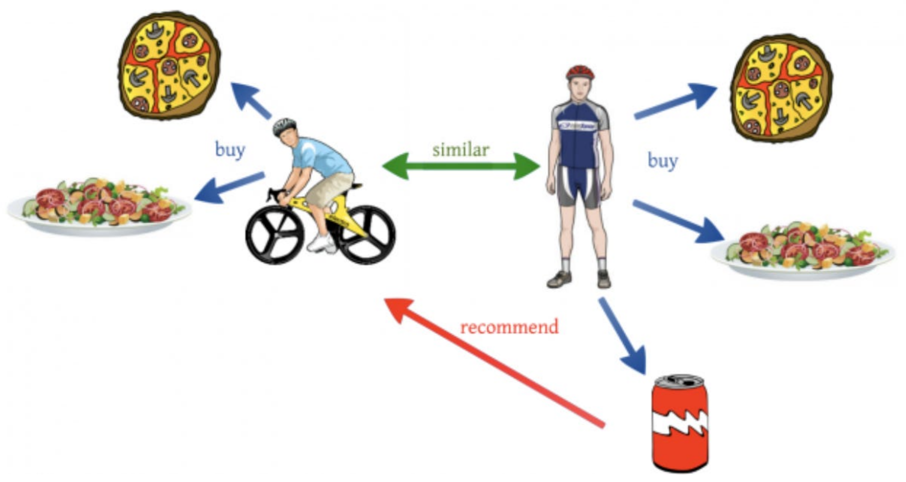
```
Predictions: tensor([1, 3, 2, 1, 0, 3, 2, 0, 2, 0])
Labels: tensor([1, 3, 2, 1, 0, 3, 2, 0, 2, 0])
Test accuracy: 99.5%
```

1.38 Recommendation Systems

There are many ways to recommend items to users. There are two primary types of recommendation systems, each with different sub-types. The two primary types are **content-based** and **collaborative filtering**.

1.38.1 Collaborative Filtering

It primarily makes recommendations based on inputs or actions from other people.



- Ignore User and Item Attributes
- Focus on User-Item Interactions
- Pure Behavior-Based Recommendation

Variations on this type of recommendation system include:

Key Concepts

- Nearest Neighbor Collaborative Filtering
- User-User CF Algorithm
  - Neighborhoods and Tuning Parameters
  - Alternatives to Historic Agreement (social, trust)
- Item-Item CF Algorithm
  - Dealing with Unary Data

- Hybrids and Extensions
- Practical Implications

User-User Collaborative Filtering

This strategy involves creating user groups by comparing users' activities and providing recommendations that are popular among other members of the group. It is useful on sites with a strong but versatile audience to quickly provide recommendations for a user on which little information is available.

Find users similar to you and recommend what they like.

Excerise: Movie Recommendations

This is a 25 user x 100 movie matrix of ratings selected from the class data set. Rows are movies ratings, columns are users, and cells are ratings from 1 to 5.

```
import pandas as pd
import torch
import seaborn
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
```

```
df = pd.read_csv('https://github.com/akkefa/ml-notes/releases/download/v0.1.0/
↪recommendation_systems_movies_ratings_data.csv')
```

```
df.head()
```

```
                                         Unnamed: 0  1648  5136  918  2824  \
0      11: Star Wars: Episode IV - A New Hope (1977)   NaN   4.5  5.0   4.5
1                            12: Finding Nemo (2003)   NaN   5.0  5.0   NaN
2                            13: Forrest Gump (1994)   NaN   5.0  4.5   5.0
3                         14: American Beauty (1999)   NaN   4.0  NaN   NaN
4  22: Pirates of the Caribbean: The Curse of the...   4.0   5.0  3.0   4.5

   3867  860  3712  2968  3525  ...  3556  5261  2492  5062  2486  4942  2267  \
0   4.0  4.0   NaN   5.0   4.0  ...   4.0   NaN   4.5   4.0   3.5   NaN   NaN
1   4.0  4.0   4.5   4.5   4.0  ...   4.0   NaN   3.5   4.0   2.0   3.5   NaN
2   4.5  4.5   NaN   5.0   4.5  ...   4.0   5.0   3.5   4.5   4.5   4.0   3.5
3   NaN  NaN   4.5   2.0   3.5  ...   4.0   NaN   3.5   4.5   3.5   4.0   NaN
4   4.0  2.5   NaN   5.0   3.0  ...   3.0   1.5   4.0   4.0   2.5   3.5   NaN

   4809  3853  2288
0   NaN   NaN   NaN
1   NaN   NaN   3.5
2   4.5   3.5   3.5
3   3.5   NaN   NaN
4   5.0   NaN   3.5

[5 rows x 26 columns]
```

```
tmp_df = df.copy()
```

```
# Drop the first column (movie title)
tmp_df.drop(columns=tmp_df.columns[0], axis=1, inplace=True)
```

```
tmp_df.head()
```

```
   1648  5136  918  2824  3867  860  3712  2968  3525  4323  ...  3556  5261  \
0   NaN   4.5  5.0   4.5   4.0  4.0   NaN   5.0   4.0   5.0  ...   4.0   NaN
1   NaN   5.0  5.0   NaN   4.0  4.0   4.5   4.5   4.0   5.0  ...   4.0   NaN
2   NaN   5.0  4.5   5.0   4.5  4.5   NaN   5.0   4.5   5.0  ...   4.0   5.0
3   NaN   4.0  NaN   NaN   NaN  NaN   4.5   2.0   3.5   5.0  ...   4.0   NaN
4   4.0   5.0  3.0   4.5   4.0  2.5   NaN   5.0   3.0   4.0  ...   3.0   1.5

   2492  5062  2486  4942  2267  4809  3853  2288
0   4.5   4.0   3.5   NaN   NaN   NaN   NaN   NaN
1   3.5   4.0   2.0   3.5   NaN   NaN   NaN   3.5
2   3.5   4.5   4.5   4.0   3.5   4.5   3.5   3.5
3   3.5   4.5   3.5   4.0   NaN   3.5   NaN   NaN
4   4.0   4.0   2.5   3.5   NaN   5.0   NaN   3.5

[5 rows x 25 columns]
```

Given a set of items I, and a set of users U, and a sparse matrix of ratings R, We compute the prediction $s(\mathrm{u}, \mathrm{i})$ as follows:

- For all users $v \neq u$, compute w_{uv}
- similarity metric (e.g., Pearson correlation)
- Select a neighborhood of users $V \subset U$ with highest w_{uv}
- may limit neighborhood to top-k neighbors
- may limit neighborhood to sim > sim_threshold
- may use sim or |sim| (risks of negative correlations)
- may limit neighborhood to people who rated i (if single-use)

$$s(u, i) = \bar{r}_u + \frac{\sum_{v \in V} (r_{vi} - \bar{r}_v) * w_{uv}}{\sum_{v \in V} w_{uv}}$$

Computing the person correlation coefficient between each pair of users. Pearson correlation coefficient formula:

$$r_{xy} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^{n} (y_i - \bar{y})^2}}$$

where $\bar{x}$ and $\bar{y}$ are the means of x and y respectively.

```
corr_df = tmp_df.corr()
corr_df
```

```
          1648      5136       918      2824      3867       860      3712  \
1648  1.000000  0.402980 -0.142206  0.517620  0.300200  0.480537 -0.312412
5136  0.402980  1.000000  0.118979  0.057916  0.341734  0.241377  0.131398
```

(continues on next page)

(continued from previous page)

```
918  -0.142206  0.118979  1.000000 -0.317063  0.294558  0.468333  0.092037
2824  0.517620  0.057916 -0.317063  1.000000 -0.060913 -0.008066  0.462910
3867  0.300200  0.341734  0.294558 -0.060913  1.000000  0.282497  0.400275
860   0.480537  0.241377  0.468333 -0.008066  0.282497  1.000000  0.171151
3712 -0.312412  0.131398  0.092037  0.462910  0.400275  0.171151  1.000000
2968  0.383348  0.206695 -0.045854  0.214760  0.264249  0.072927  0.065015
3525  0.092775  0.360056  0.367568  0.169907  0.125193  0.387133  0.095623
4323  0.098191  0.033642 -0.035394  0.119350 -0.333602  0.146158 -0.292501
3617 -0.041734  0.138548  0.011316  0.282756 -0.066576  0.219929 -0.038900
4360  0.264425  0.152948 -0.231660 -0.005326 -0.093801 -0.005316 -0.364324
2756  0.261268  0.148882  0.148431 -0.087747  0.310104  0.323499  0.126899
89    0.464610  0.562449  0.267029  0.241567 -0.003878  0.539066 -0.051320
442   0.022308  0.414438  0.304139  0.116532  0.113581  0.181276  0.227130
3556 -0.191988  0.488607  0.373226 -0.201275  0.174085  0.347470  0.016406
5261  0.493008  0.328120  0.470972  0.228341  0.297977  0.399436 -0.240764
2492  0.360644  0.422236  0.069956  0.238700  0.476683  0.207314 -0.115254
5062  0.551089  0.226635 -0.054762  0.259660  0.293868  0.311363  0.247693
2486  0.002544  0.305803  0.133812  0.247097  0.438992  0.276306  0.166913
4942  0.116653  0.037769  0.015169  0.149247 -0.162818  0.079698  0.146011
2267 -0.429183  0.240728 -0.273096 -0.361466 -0.295966  0.212991  0.009685
4809  0.394371  0.411676  0.082528  0.474974  0.054518  0.165608 -0.451625
3853 -0.304422  0.189234  0.667168 -0.262073  0.464110  0.162314  0.193660
2288  0.245048  0.390067  0.119162  0.166999  0.379856  0.279677  0.113266

          2968      3525      4323  ...      3556      5261      2492  \
1648  0.383348  0.092775  0.098191  ... -0.191988  0.493008  0.360644
5136  0.206695  0.360056  0.033642  ...  0.488607  0.328120  0.422236
918  -0.045854  0.367568 -0.035394  ...  0.373226  0.470972  0.069956
2824  0.214760  0.169907  0.119350  ... -0.201275  0.228341  0.238700
3867  0.264249  0.125193 -0.333602  ...  0.174085  0.297977  0.476683
860   0.072927  0.387133  0.146158  ...  0.347470  0.399436  0.207314
3712  0.065015  0.095623 -0.292501  ...  0.016406 -0.240764 -0.115254
2968  1.000000  0.028529 -0.073252  ...  0.049132 -0.009041  0.203613
3525  0.028529  1.000000  0.210879  ...  0.475711  0.306957  0.136343
4323 -0.073252  0.210879  1.000000  ... -0.040606  0.155045 -0.204164
3617  0.312573  0.243283  0.022907  ...  0.079571 -0.165628  0.053306
4360  0.053024 -0.086061  0.252529  ...  0.072993  0.161882 -0.000311
2756  0.143347  0.058365 -0.221789  ...  0.101784 -0.140953  0.150476
89   -0.118085  0.475495  0.258866  ...  0.326774  0.291476  0.372676
442   0.100841  0.201734 -0.024337  ...  0.251660  0.046822  0.218575
3556  0.049132  0.475711 -0.040606  ...  1.000000  0.086665  0.158739
5261 -0.009041  0.306957  0.155045  ...  0.086665  1.000000  0.149165
2492  0.203613  0.136343 -0.204164  ...  0.158739  0.149165  1.000000
5062  0.033301  0.301750  0.263654  ... -0.016164  0.372177  0.276883
2486  0.137982  0.143414  0.167198  ...  0.256537  0.198086  0.158002
4942  0.070602  0.056100 -0.084592  ... -0.055137  0.270928  0.035825
2267  0.109452  0.179908  0.315712  ...  0.503247 -0.393376 -0.345495
4809 -0.083562  0.284648  0.085673  ...  0.100277  0.455274  0.449025
3853 -0.089317  0.170757 -0.109892  ...  0.423225  0.039050  0.289410
2288  0.229219  0.193131 -0.279385  ...  0.222458  0.374264  0.169239

          5062      2486      4942      2267      4809      3853      2288
```

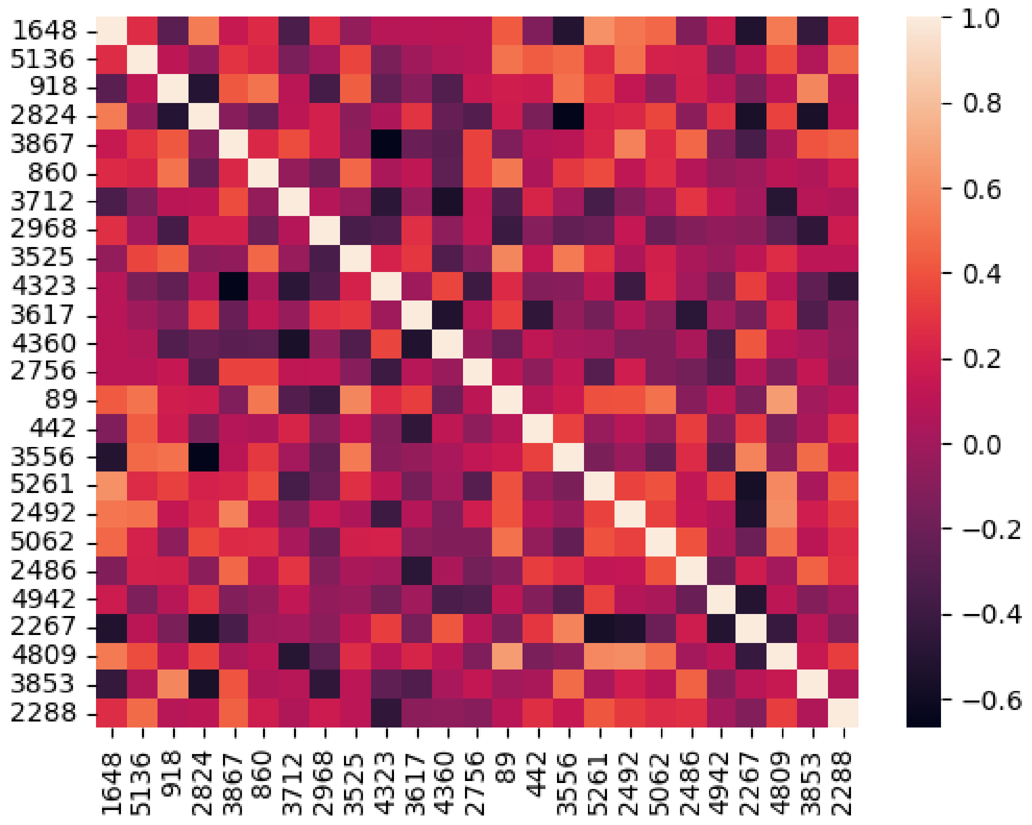
(continues on next page)

(continued from previous page)

```
1648  0.551089  0.002544  0.116653 -0.429183  0.394371 -0.304422  0.245048
5136  0.226635  0.305803  0.037769  0.240728  0.411676  0.189234  0.390067
918  -0.054762  0.133812  0.015169 -0.273096  0.082528  0.667168  0.119162
2824  0.259660  0.247097  0.149247 -0.361466  0.474974 -0.262073  0.166999
3867  0.293868  0.438992 -0.162818 -0.295966  0.054518  0.464110  0.379856
860   0.311363  0.276306  0.079698  0.212991  0.165608  0.162314  0.279677
3712  0.247693  0.166913  0.146011  0.009685 -0.451625  0.193660  0.113266
2968  0.033301  0.137982  0.070602  0.109452 -0.083562 -0.089317  0.229219
3525  0.301750  0.143414  0.056100  0.179908  0.284648  0.170757  0.193131
4323  0.263654  0.167198 -0.084592  0.315712  0.085673 -0.109892 -0.279385
3617  0.007810 -0.244637 -0.030709 -0.070660  0.268595 -0.143503  0.013284
4360 -0.077598  0.039389 -0.156091  0.408592  0.179652  0.280402  0.040328
2756  0.024572 -0.031130 -0.133768  0.142067  0.015140  0.181210 -0.005935
89    0.525990  0.123380  0.178088  0.088600  0.668516  0.179680  0.155869
442   0.150431  0.280392  0.038378  0.262520  0.064179 -0.023439  0.257864
3556 -0.016164  0.256537 -0.055137  0.503247  0.100277  0.423225  0.222458
5261  0.372177  0.198086  0.270928 -0.393376  0.455274  0.039050  0.374264
2492  0.276883  0.158002  0.035825 -0.345495  0.449025  0.289410  0.169239
5062  1.000000  0.403809  0.028521  0.107821  0.428055  0.407044  0.278868
2486  0.403809  1.000000 -0.068421  0.173797  0.105761  0.472361  0.257462
4942  0.028521 -0.068421  1.000000 -0.346386 -0.004638  0.143672  0.074476
2267  0.107821  0.173797 -0.346386  1.000000 -0.339845  0.165960  0.156341
4809  0.428055  0.105761 -0.004638 -0.339845  1.000000  0.542192  0.435520
3853  0.407044  0.472361  0.143672  0.165960  0.542192  1.000000  0.080403
2288  0.278868  0.257462  0.074476  0.156341  0.435520  0.080403  1.000000

[25 rows x 25 columns]
```

```
seaborn.heatmap(corr_df.corr())
```

```
<Axes: >
```



```
corr_df["3867"].sort_values(ascending=False)
```

```
3867    1.000000
2492    0.476683
3853    0.464110
2486    0.438992
3712    0.400275
2288    0.379856
5136    0.341734
2756    0.310104
1648    0.300200
5261    0.297977
918     0.294558
5062    0.293868
860     0.282497
2968    0.264249
3556    0.174085
3525    0.125193
442     0.113581
4809    0.054518
89     -0.003878
2824   -0.060913
3617   -0.066576
4360   -0.093801
4942   -0.162818
```

(continues on next page)

(continued from previous page)

```
2267   -0.295966
4323   -0.333602
Name: 3867, dtype: float64
```

```
corr_df["89"].sort_values(ascending=False)
```

```
89      1.000000
4809    0.668516
5136    0.562449
860     0.539066
5062    0.525990
3525    0.475495
1648    0.464610
2492    0.372676
3556    0.326774
442     0.296826
5261    0.291476
2756    0.290591
3617    0.278335
918     0.267029
4323    0.258866
2824    0.241567
3853    0.179680
4942    0.178088
2288    0.155869
2486    0.123380
2267    0.088600
3867   -0.003878
3712   -0.051320
4360   -0.115492
2968   -0.118085
Name: 89, dtype: float64
```

Compute the predictions for each movie for users 3867 and 89 by taking the correlation-weighted average of the ratings of the top-five neighbors (for each target user) for each movie. The formal formula for correlation-weighted average is

$$\hat{x}_{u,i} = \frac{\sum_{v \in N} r_{u,v} x_{v,i}}{\sum_{v \in N} |r_{u,v}|}$$

where N is the set of the top-five neighbors of user u and $x_{v,i}$ is the rating of user v for movie i.

```
def get_top_users(df_corr,target,n=5):
    target_cor = df_corr.loc[target]
    top_neighbors = target_cor.nlargest(n+1).iloc[1:]
    return top_neighbors

def get_user_movie_score(movie,user):
    neighbors = get_top_users(corr_df,str(user))
    rating_sum = 0
    weight_sum = 0
    for user,w in zip(neighbors.index,neighbors.values):
        if np.isnan(movie[user]):
```

(continues on next page)

(continued from previous page)

```
            continue
        rating_sum += movie[user] * w
        weight_sum += w
    if weight_sum == 0:
        return 0
    else:
        return rating_sum/weight_sum
```

```
get_top_users(corr_df,"3867")
```

```
2492    0.476683
3853    0.464110
2486    0.438992
3712    0.400275
2288    0.379856
Name: 3867, dtype: float64
```

```
get_top_users(corr_df,"3712")
```

```
2824    0.462910
3867    0.400275
5062    0.247693
442     0.227130
3853    0.193660
Name: 3712, dtype: float64
```

```
pred_3867 = df.apply(get_user_movie_score,axis=1,args=(3867,))
pred_89 = df.apply(get_user_movie_score,axis=1,args=(89,))
```

```
pred_3867.sort_values(ascending=False)[:3]
```

```
77    4.760291
21    4.551454
16    4.507637
dtype: float64
```

```
for i in pred_3867.sort_values(ascending=False)[:5].index:
    print(df.loc[i][0])
```

```
1891: Star Wars: Episode V - The Empire Strikes Back (1980)
155: The Dark Knight (2008)
122: The Lord of the Rings: The Return of the King (2003)
77: Memento (2000)
121: The Lord of the Rings: The Two Towers (2002)
```

```
/tmp/ipykernel_1076/879984262.py:2: FutureWarning: Series.__getitem__ treating keys as␣
↪positions is deprecated. In a future version, integer keys will always be treated as␣
↪labels (consistent with DataFrame behavior). To access a value by position, use `ser.
↪iloc[pos]`
  print(df.loc[i][0])
```

```
for i in pred_89.sort_values(ascending=False)[:5].index:
    print(df.loc[i][0])
```

```
238: The Godfather (1972)
278: The Shawshank Redemption (1994)
807: Seven (a.k.a. Se7en) (1995)
275: Fargo (1996)
424: Schindler's List (1993)
```

```
/tmp/ipykernel_1076/3855090423.py:2: FutureWarning: Series.__getitem__ treating keys as␣
↪positions is deprecated. In a future version, integer keys will always be treated as␣
↪labels (consistent with DataFrame behavior). To access a value by position, use `ser.
↪iloc[pos]`
  print(df.loc[i][0])
```

Normalization

```
def get_norm_user_movie_score(movie,user):
    user = str(user)
    neighbors = get_top_users(corr_df,str(user))
    rating_sum = 0
    weight_sum = 0
    user_rating_mean = df.loc[:,user].mean()
    for user,w in zip(neighbors.index,neighbors.values):
        if np.isnan(movie[user]):
            continue
        movie_user_mean = df.loc[:,user].mean()
        rating_sum += (movie[user]-movie_user_mean) * w
        weight_sum += w
    if weight_sum == 0:
        return 0
    else:
        return user_rating_mean + rating_sum/weight_sum
```

```
norm_pred_3867 = df.apply(get_norm_user_movie_score,axis=1,args=(3867,))
norm_pred_89 = df.apply(get_norm_user_movie_score,axis=1,args=(89,))
```

```
for i in norm_pred_3867.sort_values(ascending=False)[:5].index:
    print(df.loc[i][0])
```

```
1891: Star Wars: Episode V - The Empire Strikes Back (1980)
155: The Dark Knight (2008)
77: Memento (2000)
275: Fargo (1996)
807: Seven (a.k.a. Se7en) (1995)
```

```
/tmp/ipykernel_1076/3753397190.py:2: FutureWarning: Series.__getitem__ treating keys as␣
↪positions is deprecated. In a future version, integer keys will always be treated as␣
```

(continues on next page)

(continued from previous page)

```
↪labels (consistent with DataFrame behavior). To access a value by position, use `ser.
↪iloc[pos]`
  print(df.loc[i][0])
```

Problem in User - User Collaborative Filtering:

Issues of Sparsity – With large item sets, small numbers of ratings, too often there are points where no recommendation can be made (for a user, for an item to a set of users, etc.) – Many solutions proposed here, including "filterbots", item-item, and dimensionality reduction

Computational performance – With millions of users (or more), computing all- pairs correlations is expensive – Even incremental approaches were expensive – And user profiles could change quickly – needed to compute in real time to keep users happy

Item-Item Collaborative Filtering

Item-Item similarity is fairly stable.

- This is dependent on having many more usersthan items
  - Average item has many more ratings than an average user
  - Intuitively, items don't generally change rapidly – at least not in ratings space (special case for time-bound items)
- Item similarity is a route to computing a prediction of a user's item preference

https://github.com/shenweichen/Coursera/blob/master/Specialization_Recommender_System_University_of_Minnesota/Course2_Neare

```
data = pd.read_excel("https://github.com/akkefa/ml-notes/releases/download/v0.1.0/item_
↪item_cb.xls", sheet_name=0)
```

```
data = data.fillna(0)
```

```
data.head()
```

```
   User  1: Toy Story (1995)  \
0   755                 2.0
1  5277                 1.0
2  1577                 0.0
3  4388                 2.0
4  1202                 0.0

   1210: Star Wars: Episode VI - Return of the Jedi (1983)  \
0                                                5.0
1                                                0.0
2                                                0.0
3                                                3.0
4                                                3.0

   356: Forrest Gump (1994)  318: Shawshank Redemption, The (1994)  \
0                       2.0                                    0.0
1                       0.0                                    2.0
2                       0.0                                    5.0
```

(continues on next page)

(continued from previous page)

```
3                      0.0                                     0.0
4                      4.0                                     1.0

   593: Silence of the Lambs, The (1991)  3578: Gladiator (2000)  \
0                                   4.0                     4.0
1                                   4.0                     2.0
2                                   2.0                     0.0
3                                   0.0                     1.0
4                                   4.0                     1.0

   260: Star Wars: Episode IV - A New Hope (1977)  \
0                                            1.0
1                                            5.0
2                                            0.0
3                                            0.0
4                                            4.0

   2028: Saving Private Ryan (1998)  296: Pulp Fiction (1994)  ...  \
0                               2.0                       0.0  ...
1                               0.0                       0.0  ...
2                               0.0                       0.0  ...
3                               3.0                       4.0  ...
4                               4.0                       0.0  ...

   2916: Total Recall (1990)  780: Independence Day (ID4) (1996)  \
0                        0.0                                 5.0
1                        2.0                                 2.0
2                        1.0                                 4.0
3                        4.0                                 0.0
4                        1.0                                 0.0

   541: Blade Runner (1982)  1265: Groundhog Day (1993)  \
0                       2.0                         5.0
1                       0.0                         2.0
2                       4.0                         1.0
3                       3.0                         5.0
4                       4.0                         0.0

   2571: Matrix, The (1999)  527: Schindler's List (1993)  \
0                       4.0                           2.0
1                       0.0                           5.0
2                       1.0                           2.0
3                       0.0                           5.0
4                       3.0                           5.0

   2762: Sixth Sense, The (1999)  1198: Raiders of the Lost Ark (1981)  \
0                            5.0                                   0.0
1                            1.0                                   3.0
2                            3.0                                   1.0
3                            1.0                                   1.0
4                            5.0                                   0.0
```

(continues on next page)

(continued from previous page)

```
   34: Babe (1995)      Mean
0              0.0  3.200000
1              0.0  2.769231
2              3.0  2.333333
3              2.0  2.833333
4              0.0  3.214286

[5 rows x 22 columns]
```

```
matrix = pd.read_excel('https://github.com/akkefa/ml-notes/releases/download/v0.1.0/item_
↪item_cb.xls',sheet_name=2)
```

```
matrix.head()
```

```
                                        Unnamed: 0  1: Toy Story (1995)  \
0                              1: Toy Story (1995)                  NaN
1  1210: Star Wars: Episode VI - Return of the Je...                  NaN
2                         356: Forrest Gump (1994)                  NaN
3             318: Shawshank Redemption, The (1994)                  NaN
4             593: Silence of the Lambs, The (1991)                  NaN

   1210: Star Wars: Episode VI - Return of the Jedi (1983)  \
0                                                NaN
1                                                NaN
2                                                NaN
3                                                NaN
4                                                NaN

   356: Forrest Gump (1994)  318: Shawshank Redemption, The (1994)  \
0                      NaN                                    NaN
1                      NaN                                    NaN
2                      NaN                                    NaN
3                      NaN                                    NaN
4                      NaN                                    NaN

   593: Silence of the Lambs, The (1991)  3578: Gladiator (2000)  \
0                                    NaN                     NaN
1                                    NaN                     NaN
2                                    NaN                     NaN
3                                    NaN                     NaN
4                                    NaN                     NaN

   260: Star Wars: Episode IV - A New Hope (1977)  \
0                                             NaN
1                                             NaN
2                                             NaN
3                                             NaN
4                                             NaN

   2028: Saving Private Ryan (1998)  296: Pulp Fiction (1994)  ...  \
0                               NaN                       NaN  ...
```

(continues on next page)

(continued from previous page)

```
1                            NaN                        NaN  ...
2                            NaN                        NaN  ...
3                            NaN                        NaN  ...
4                            NaN                        NaN  ...

   2396: Shakespeare in Love (1998)  2916: Total Recall (1990)  \
0                            NaN                         NaN
1                            NaN                         NaN
2                            NaN                         NaN
3                            NaN                         NaN
4                            NaN                         NaN

   780: Independence Day (ID4) (1996)  541: Blade Runner (1982)  \
0                              NaN                         NaN
1                              NaN                         NaN
2                              NaN                         NaN
3                              NaN                         NaN
4                              NaN                         NaN

   1265: Groundhog Day (1993)  2571: Matrix, The (1999)  \
0                      NaN                       NaN
1                      NaN                       NaN
2                      NaN                       NaN
3                      NaN                       NaN
4                      NaN                       NaN

   527: Schindler's List (1993)  2762: Sixth Sense, The (1999)  \
0                        NaN                            NaN
1                        NaN                            NaN
2                        NaN                            NaN
3                        NaN                            NaN
4                        NaN                            NaN

   1198: Raiders of the Lost Ark (1981)  34: Babe (1995)
0                                NaN              NaN
1                                NaN              NaN
2                                NaN              NaN
3                                NaN              NaN
4                                NaN              NaN

[5 rows x 21 columns]
```

```
matrix = pd.DataFrame(cosine_similarity(data.values[:-1,1:-1].T),index=matrix.index,
↪columns=matrix.columns[1:])
```

```
matrix = matrix.applymap(lambda x:max(0,x))
```

```
/tmp/ipykernel_1076/1610576545.py:1: FutureWarning: DataFrame.applymap has been␣
↪deprecated. Use DataFrame.map instead.
  matrix = matrix.applymap(lambda x:max(0,x))
```

```
matrix.columns
```

```
Index(['1: Toy Story (1995)',
       '1210: Star Wars: Episode VI - Return of the Jedi (1983)',
       '356: Forrest Gump (1994)', '318: Shawshank Redemption, The (1994)',
       '593: Silence of the Lambs, The (1991)', '3578: Gladiator (2000)',
       '260: Star Wars: Episode IV - A New Hope (1977)',
       '2028: Saving Private Ryan (1998)', '296: Pulp Fiction (1994)',
       '1259: Stand by Me (1986)', '2396: Shakespeare in Love (1998)',
       '2916: Total Recall (1990)', '780: Independence Day (ID4) (1996)',
       '541: Blade Runner (1982)', '1265: Groundhog Day (1993)',
       '2571: Matrix, The (1999)', '527: Schindler's List (1993)',
       '2762: Sixth Sense, The (1999)', '1198: Raiders of the Lost Ark (1981)',
       '34: Babe (1995)'],
      dtype='object')
```

```
matrix.iloc[0].nlargest(6).iloc[1:]
```

```
260: Star Wars: Episode IV - A New Hope (1977)    0.747409
780: Independence Day (ID4) (1996)                0.690665
296: Pulp Fiction (1994)                          0.667846
318: Shawshank Redemption, The (1994)             0.667424
1265: Groundhog Day (1993)                        0.661016
Name: 0, dtype: float64
```

```
def get_score(row,user):
    user_rating = data.loc[(data.User==user)]
    user_hist_item = user_rating.columns[pd.notnull(user_rating).values[0]]
    movie_name = row.name

    neighbor_names = user_hist_item.tolist()#row.index.tolist()

    if 'User' in neighbor_names:
        neighbor_names.remove('User')
    if 'Mean' in neighbor_names:
        neighbor_names.remove('Mean')
    a = row.loc[neighbor_names].values
    b = data.loc[data.User==user,neighbor_names]

    return np.dot(a,b.values[0])/np.sum(a)
```

```
user_rating = data.loc[data.User==5277]
```

```
idx = user_rating.columns[pd.notnull(user_rating).values[0]].tolist()
idx.remove('User')
idx.remove('Mean')

idx
```

```
['1: Toy Story (1995)',
 '1210: Star Wars: Episode VI - Return of the Jedi (1983)',
```

(continues on next page)

(continued from previous page)

```
'356: Forrest Gump (1994)',
'318: Shawshank Redemption, The (1994)',
'593: Silence of the Lambs, The (1991)',
'3578: Gladiator (2000)',
'260: Star Wars: Episode IV - A New Hope (1977)',
'2028: Saving Private Ryan (1998)',
'296: Pulp Fiction (1994)',
'1259: Stand by Me (1986)',
'2396: Shakespeare in Love (1998)',
'2916: Total Recall (1990)',
'780: Independence Day (ID4) (1996)',
'541: Blade Runner (1982)',
'1265: Groundhog Day (1993)',
'2571: Matrix, The (1999)',
"527: Schindler's List (1993)",
'2762: Sixth Sense, The (1999)',
'1198: Raiders of the Lost Ark (1981)',
'34: Babe (1995)']
```

```
ans = matrix.apply(get_score,axis=1,args=(5277,))
```

1.38.2 Content-Based

1.39 Matrix Factorization

Matrix factorization is a class of collaborative filtering algorithms used in recommender systems. Matrix factorization algorithms work by decomposing the user-item interaction matrix into the product of two lower dimensionality rectangular matrices. The rows of the first matrix represent the latent user factors and the columns of the second matrix represent the latent item factors. The dot product of these two matrices approximates the original user-item interaction matrix. The latent factors are also known as embeddings and are typically of much lower dimensionality than the original user and item vectors. The latent factors are learned through an iterative process that minimizes the error between the dot product of the latent factors and the original user-item interaction matrix. The error is measured using a loss function such as mean squared error (MSE) or binary cross entropy (BCE). The loss function is minimized using gradient descent or one of its variants.

1.39.1 Singular Value Decomposition (SVD)

So the singular value decomposition comes from linear algebra, and it's a way of breaking down a matrix into constituent parts. we can factorize it into three matrices. And this is called factorization because it works a lot like factoring numbers. You take 15, and you can factorize it into 3 and 5, such that you multiply 3 and 5 together, and you get 15.

$$R = P\Sigma Q^{\mathrm{T}}$$

- R is $m \times n$ ratings matrix
- P is $m \times k$ user-feature affinity matrix
- Q is $n \times k$ item-feature relevance matrix
- Σ is $k \times k$ diagonal feature weight matrix
- For linear algebra people: P and Q are orthogonal

- Linear algebra guarantees this exists for any real R

latent features

Latent means not directly observable. The common use of the term in PCA and Factor Analysis is to reduce dimension of a large number of directly observable features into a smaller set of indirectly observable features.

- SVD describes preference in terms of latent features
- These features are learned from the rating data
- Not necessarily interpretable
  - Optimized for predictive power
- Defines a shared vector space for users and items (feature space)
  - Enables compact representation of each

Example using Superise library

```
import pandas as pd
from surprise import Reader
from surprise import Dataset
from surprise.model_selection import cross_validate
from surprise import NormalPredictor
from surprise import KNNBasic
from surprise import KNNWithMeans
from surprise import KNNWithZScore
from surprise import KNNBaseline
from surprise import SVD
from surprise import BaselineOnly
from surprise import SVDpp
from surprise import NMF
from surprise import SlopeOne
from surprise import CoClustering
from surprise.accuracy import rmse
from surprise import accuracy
from surprise.model_selection import train_test_split
from surprise.model_selection import GridSearchCV
```

Importing data

GroupLens Research has collected and made available rating data sets from the MovieLens web site (http://movielens.org). The data sets were collected over various periods of time, depending on the size of the set.

We are using Small: 100,000 ratings and 3,600 tag applications applied to 9,000 movies by 600 users. Last updated 9/2018.

Download: ml-latest-small.zip (size: 1 MB)

```
df = pd.read_csv ("https://raw.githubusercontent.com/singhsidhukuldeep/Recommendation-
↪System/master/data/ratings.csv")
```

```
df.head()
```

```
   userId  movieId  rating  timestamp
0       1        1     4.0  964982703
1       1        3     4.0  964981247
2       1        6     4.0  964982224
3       1       47     5.0  964983815
4       1       50     5.0  964982931
```

```
df.tail()
```

```
        userId  movieId  rating   timestamp
100831     610   166534     4.0  1493848402
100832     610   168248     5.0  1493850091
100833     610   168250     5.0  1494273047
100834     610   168252     5.0  1493846352
100835     610   170875     3.0  1493846415
```

```
df.drop(['timestamp'], axis=1, inplace=True)
df.columns = ['userID', 'item', 'rating']
```

```
df.head()
```

```
   userID  item  rating
0       1     1     4.0
1       1     3     4.0
2       1     6     4.0
3       1    47     5.0
4       1    50     5.0
```

```
df.shape
```

```
(100836, 3)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100836 entries, 0 to 100835
Data columns (total 3 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   userID  100836 non-null  int64  
 1   item    100836 non-null  int64  
 2   rating  100836 non-null  float64
dtypes: float64(1), int64(2)
memory usage: 2.3 MB
```

```
print('Dataset shape: {}'.format(df.shape))
print('-Dataset examples-')
print(df.iloc[::20000, :])
```

```
Dataset shape: (100836, 3)
-Dataset examples-
        userID  item  rating
0            1     1     4.0
20000      132  1079     3.5
40000      274  5621     2.0
60000      387  6748     3.0
80000      501    11     3.0
100000     610  6978     4.0
```

```
# To reduce the dimensionality of the dataset, we will filter out rarely rated movies␣
↪and rarely rating users.

min_ratings = 5
filter_items = df['item'].value_counts() > min_ratings
filter_items = filter_items[filter_items].index.tolist()

min_user_ratings = 5
filter_users = df['userID'].value_counts() > min_user_ratings
filter_users = filter_users[filter_users].index.tolist()

df_new = df[(df['item'].isin(filter_items)) & (df['userID'].isin(filter_users))]
print('The original data frame shape:\t{}'.format(df.shape))
print('The new data frame shape:\t{}'.format(df_new.shape))
```

```
The original data frame shape:        (100836, 3)
The new data frame shape:        (88364, 3)
```

Surprise library

To load a dataset from a pandas dataframe, we will use the load_from_df() method, we will also need a Reader object, and the rating_scale parameter must be specified. The dataframe must have three columns, corresponding to the user ids, the item ids, and the ratings in this order. Each row thus corresponds to a given rating.

```
reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(df_new[['userID', 'item', 'rating']], reader)
```

Basic algorithms

With the Surprise library, we will benchmark the following algorithms

NormalPredictor

- NormalPredictor algorithm predicts a random rating based on the distribution of the training set, which is assumed to be normal. This is one of the most basic algorithms that do not do much work.

BaselineOnly

- BasiclineOnly algorithm predicts the baseline estimate for given user and item.

k-NN algorithms

KNNBasic

- KNNBasic is a basic collaborative filtering algorithm.

KNNWithMeans

- KNNWithMeans is basic collaborative filtering algorithm, taking into account the mean ratings of each user.

KNNWithZScore

- KNNWithZScore is a basic collaborative filtering algorithm, taking into account the z-score normalization of each user.

KNNBaseline

- KNNBaseline is a basic collaborative filtering algorithm taking into account a baseline rating.

Matrix Factorization-based algorithms

SVD

- SVD algorithm is equivalent to Probabilistic Matrix Factorization (http://papers.nips.cc/paper/3208-probabilistic-matrix-factorization.pdf)

SVDpp

- The SVDpp algorithm is an extension of SVD that takes into account implicit ratings.

NMF

- NMF is a collaborative filtering algorithm based on Non-negative Matrix Factorization. It is very similar with SVD.

Slope One

- Slope One is a straightforward implementation of the SlopeOne algorithm. (https://arxiv.org/abs/cs/0702144)

Co-clustering

- Co-clustering is a collaborative filtering algorithm based on co-clustering (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.6458&rep=rep1&type=pdf)

We use rmse as our accuracy metric for the predictions.

```
benchmark = []
# Iterate over all algorithms

# algorithms = [SVD(), SVDpp(), SlopeOne(), NMF(), NormalPredictor(), KNNBaseline(),␣
↪KNNBasic(), KNNWithMeans(), KNNWithZScore(), BaselineOnly(), CoClustering()]
algorithms = [SVD(), KNNWithMeans(), CoClustering()]
# print ("Attempting: ", str(algorithms), '\n\n\n')

for algorithm in algorithms:
    # print("Starting: " ,str(algorithm))
    print("Starting: ",str(algorithm).split(' ')[0].split('.')[-1])
    # Perform cross validation
    results = cross_validate(algorithm, data, measures=['RMSE'], cv=3, verbose=True)
    # results = cross_validate(algorithm, data, measures=['RMSE','MAE'], cv=3,␣
↪verbose=False)
    # Get results & append algorithm name
    tmp = pd.DataFrame.from_dict(results).mean(axis=0)
    tmp = tmp._append(pd.Series([str(algorithm).split(' ')[0].split('.')[-1]], index=[
↪'Algorithm']))
    benchmark.append(tmp)
    print("Done: " ,str(algorithm), "\n\n")

print ('\n\tDONE\n')
```

```
Starting:  SVD
```

```
Evaluating RMSE of algorithm SVD on 3 split(s).

                  Fold 1  Fold 2  Fold 3  Mean    Std
RMSE (testset)    0.8577  0.8660  0.8664  0.8634  0.0040
```

(continues on next page)

(continued from previous page)

```
Fit time          0.62    0.61    0.57    0.60    0.02
Test time         0.17    0.22    0.16    0.18    0.03
Done:  <surprise.prediction_algorithms.matrix_factorization.SVD object at 0x7f4e99bb9b10>


Starting:  KNNWithMeans
Computing the msd similarity matrix...
Done computing similarity matrix.
```

```
Computing the msd similarity matrix...
Done computing similarity matrix.
```

```
Computing the msd similarity matrix...
Done computing similarity matrix.
```

```
Evaluating RMSE of algorithm KNNWithMeans on 3 split(s).

                  Fold 1  Fold 2  Fold 3  Mean    Std
RMSE (testset)    0.8719  0.8702  0.8656  0.8692  0.0026
Fit time          0.07    0.08    0.08    0.08    0.01
Test time         1.46    1.48    1.47    1.47    0.01
Done:  <surprise.prediction_algorithms.knns.KNNWithMeans object at 0x7f4e9ad77450>


Starting:  CoClustering
```

```
Evaluating RMSE of algorithm CoClustering on 3 split(s).

                  Fold 1  Fold 2  Fold 3  Mean    Std
RMSE (testset)    0.9148  0.9180  0.9245  0.9191  0.0040
Fit time          1.13    0.96    1.05    1.05    0.07
Test time         0.19    0.19    0.20    0.20    0.00
Done:  <surprise.prediction_algorithms.co_clustering.CoClustering object at␣
↪0x7f4e99bb9290>



        DONE
```

```
surprise_results = pd.DataFrame(benchmark).set_index('Algorithm').sort_values('test_rmse
↪')
```

```
surprise_results
```

```
              test_rmse  fit_time  test_time
Algorithm
SVD            0.863371  0.599150   0.183198
KNNWithMeans   0.869243  0.077147   1.473157
CoClustering   0.919128  1.046856   0.195326
```

SVDpp is performing best but it is taking a lot of time so we will use SED instean but apply GridSearch CV.

```
# param_grid = {
#     "n_epochs": [5, 10, 15, 20, 30, 40, 50, 100],
#     "lr_all": [0.001, 0.002, 0.005],
#     "reg_all": [0.02, 0.08, 0.4, 0.6]
# }

# smaller grid for testing
param_grid = {
    "n_epochs": [10, 20],
    "lr_all": [0.002, 0.005],
    "reg_all": [0.02]
}
gs = GridSearchCV(SVD, param_grid, measures=["rmse", "mae"], refit=True, cv=5)

gs.fit(data)

training_parameters = gs.best_params["rmse"]

print("BEST RMSE: \t", gs.best_score["rmse"])
print("BEST MAE: \t", gs.best_score["mae"])
print("BEST params: \t", gs.best_params["rmse"])
```

```
BEST RMSE:          0.8556226629647827
BEST MAE:          0.6566392762815944
BEST params:          {'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
```

```
from datetime import datetime
print(training_parameters)
reader = Reader(rating_scale=(1, 5))

print("\n\n\t\t STARTING\n\n")
start = datetime.now()

print("> Loading data...")
data = Dataset.load_from_df(df_new[['userID', 'item', 'rating']], reader)
print("> OK")

print("> Creating trainset...")
trainset = data.build_full_trainset()
print("> OK")


startTraining = datetime.now()
print("> Training...")

algo = SVD(n_epochs = training_parameters['n_epochs'], lr_all = training_parameters['lr_
↪all'], reg_all = training_parameters['reg_all'])

algo.fit(trainset)

endTraining = datetime.now()
print("> OK \t\t It Took: ", (endTraining-startTraining).seconds, "seconds")
```

(continues on next page)

(continued from previous page)

```

end = datetime.now()
print (">> DONE \t\t It Took", (end-start).seconds, "seconds" )
```

```
{'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}


                 STARTING


> Loading data...
> OK
> Creating trainset...
> OK
> Training...
```

```
> OK                  It Took:  0 seconds
>> DONE                  It Took 0 seconds
```

```
## SAVING TRAINED MODEL
# from surprise import dump
# import os
# model_filename = "./model.pickle"
# print (">> Starting dump")
# # Dump algorithm and reload it.
# file_name = os.path.expanduser(model_filename)
# dump.dump(file_name, algo=algo)
# print (">> Dump done")
# print(model_filename)
```

```
# ## LOAD SAVED MODEL
# def load_model(model_filename):
#     print (">> Loading dump")
#     from surprise import dump
#     import os
#     file_name = os.path.expanduser(model_filename)
#     _, loaded_model = dump.load(file_name)
#     print (">> Loaded dump")
#     return loaded_model
```

```
from pprint import pprint as pp
# model_filename = "./model.pickle"
def itemRating(user, item):
    uid = str(user)
    iid = str(item)
    # loaded_model = load_model(model_filename)
    prediction = algo.predict(user, item, verbose=True)
    rating = prediction.est
    details = prediction.details
    uid = prediction.uid
    iid = prediction.iid
```

(continues on next page)

(continued from previous page)

```
    true = prediction.r_ui
    ret = {
        'user': user,
        'item': item,
        'rating': rating,
        'details': details,
        'uid': uid,
        'iid': iid,
        'true': true
        }
    pp (ret)
    print ('\n\n')
    return ret
print(itemRating(user = "610", item = "10"))
```

```
user: 610        item: 10         r_ui = None   est = 3.54   {'was_impossible': False}
{'details': {'was_impossible': False},
 'iid': '10',
 'item': '10',
 'rating': 3.543813091304151,
 'true': None,
 'uid': '610',
 'user': '610'}



{'user': '610', 'item': '10', 'rating': 3.543813091304151, 'details': {'was_impossible':␣
↪False}, 'uid': '610', 'iid': '10', 'true': None}
```

1.40 Probability solutions

1.41 R Solutions

#. Imagine rolling a fair, six-sided die, and then flipping a fair, two-sided coin the number of times specified with the die (i.e., if we roll a 3, flip the coin 3 times). Let X be the number of heads you get in this experiment. Use a simulation in R to estimate the mean, median and mode of X.

```
#replicate
set.seed(110)
sims = 1000

#keep track of X
X = rep(0, sims)


#run the loop
for(i in 1:sims){

  #generate a roll
```

(continues on next page)

(continued from previous page)

```
  roll = sample(1:6, 1)

  #flip the coin the specified number of times
  for(j in 1:roll){

    #flip the coin
    #recall that 'runif(1)' draws a random value between 0 and 1, so
    #   count 'heads' as getting a value below 1/2
    flip = runif(1)

    #see if we got heads; increment if we did
    if(flip <= 1/2){
      X[i] = X[i] + 1
    }
  }
}

#find the mean and median
mean(X); median(X)
```

The default highlighting language is Python: it can be be changed using the `highlight` directive within a document:

```
.. highlight:: html

The literal blocks are now highlighted as HTML, until a new directive is found.

::
   <html><head></head>
   <body>This is a text.</body>
   </html>

The following directive changes the hightlight language to SQL.

.. highlight:: sql

::
   SELECT * FROM mytable

.. highlight:: none

From here on no highlighting will be done.

::
   SELECT * FROM mytable
```

```
def some_function():
    interesting = False
    print 'This line is highlighted.'
    print 'This one is not...'
    print '...but this one is.'
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

C

Continuous random variable, **17**

D

Discrete random variable, **17**