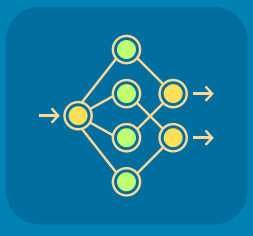
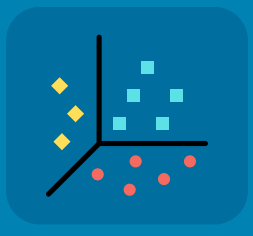
2024 EDITION

FREE

DATA SCIENCE

FULL ARCHIVE

Daily Dose of Data Science

Avi Chawla

DailyDoseofDS.com

How to make the most out of this book and your time?

Reading time of this book is about 9-10 hours. But not all chapters will be of relevance to you. This 2-minute assessment will test your current expertise and recommend chapters that will be most useful to you.

Scan the QR code below or open this link to start the assessment. It will only take 2 minutes to complete.

https://bit.ly/DS-assessment

Table of contents

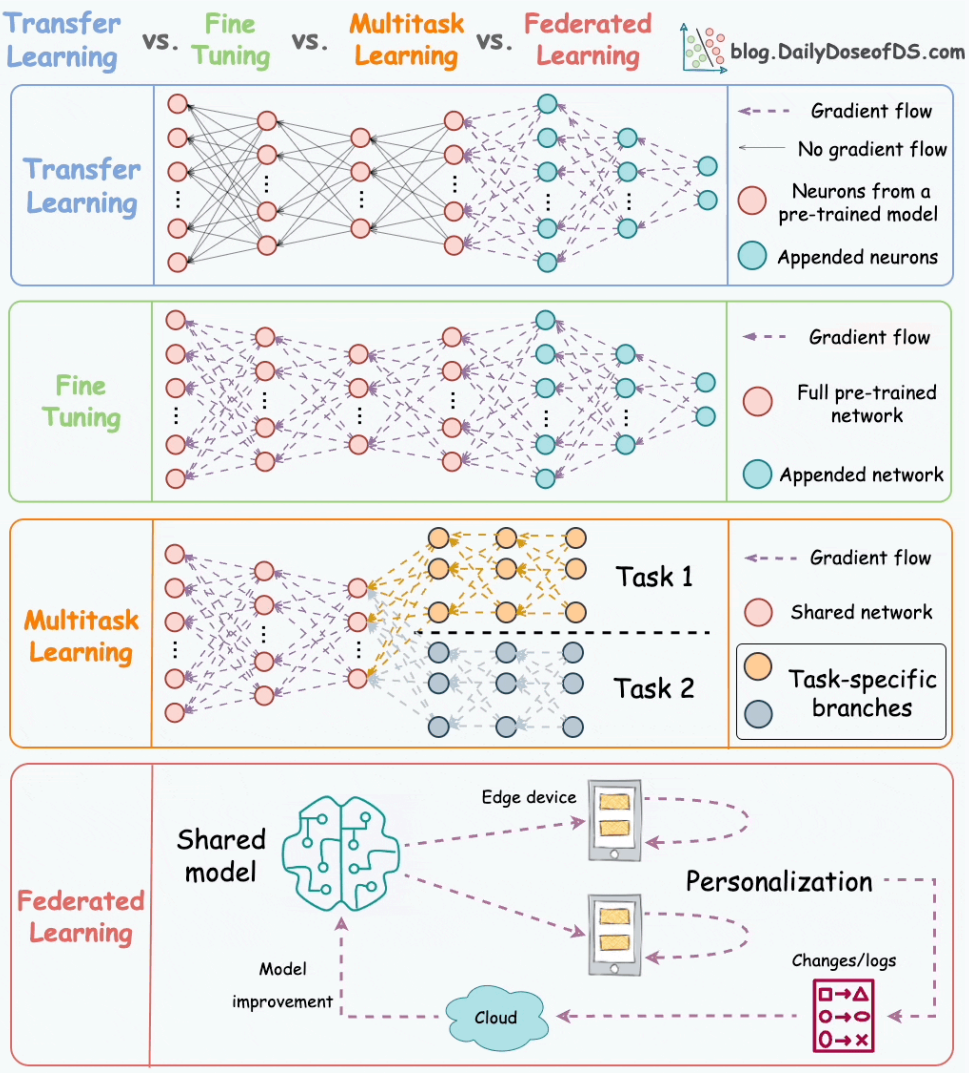
Deep learning

Learning Paradigms

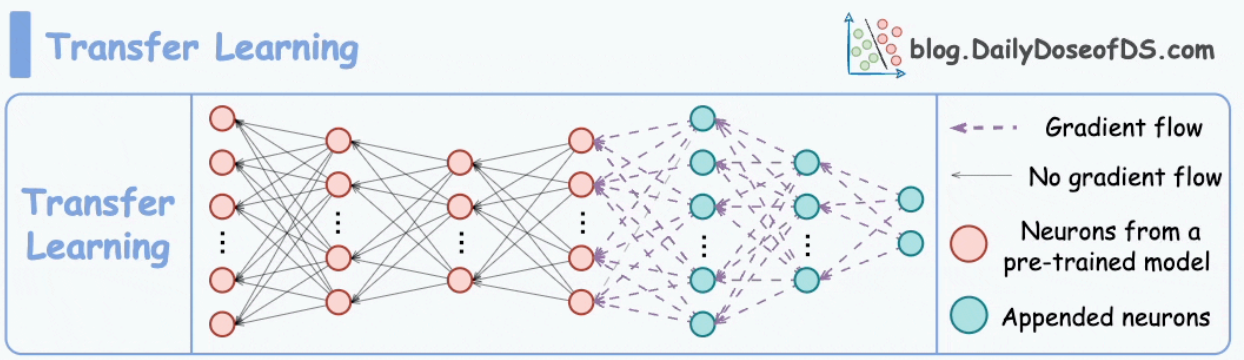
Transfer Learning, Fine-tuning, Multitask Learning and Federated Learning

Most ML models are trained independently without any interaction with other models. However, in the realm of real-world ML, there are many powerful learning techniques that rely on model interactions to improve performance.

The following image summarizes four such well-adopted and must-know training methodologies:



#1) Transfer learning



This is extremely useful when:

- The task of interest has less data.
- But a related task has abundant data.

This is how it works:

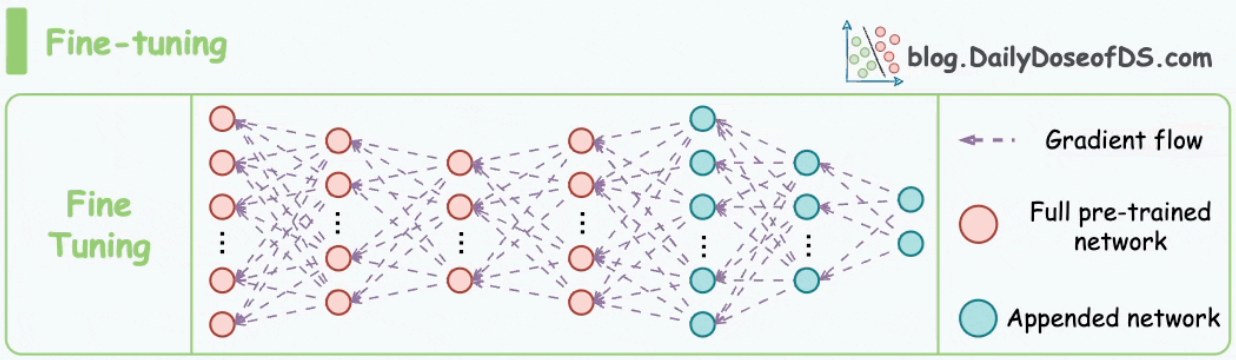
- Train a neural network model (base model) on the related task.
- Replace the last few layers on the base model with new layers.
- Train the network on the task of interest, but don't update the weights of the unreplaced layers during backpropagation.

By training a model on the related task first, we can capture the core patterns of the task of interest. Later, we can adjust the last few layers to capture task-specific behavior.

Another idea which is somewhat along these lines is knowledge distillation, which involves the "transfer" of knowledge. We will discuss it in the upcoming chapters.

Transfer learning is commonly used in many computer vision tasks.

#2) Fine-tuning

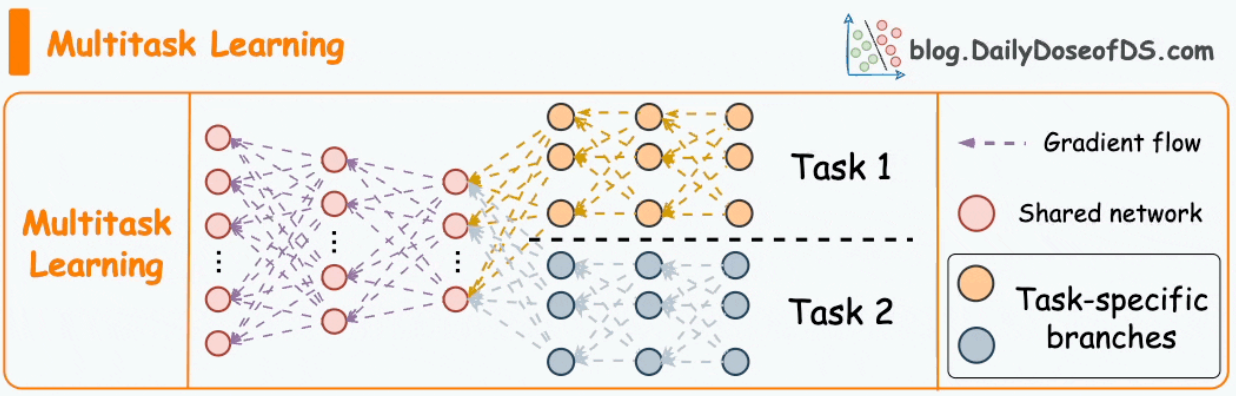


Fine-tuning involves updating the weights of some or all layers of the pre-trained model to adapt it to the new task.

The idea may appear similar to transfer learning, but in fine-tuning, we typically do not replace the last few layers of the pre-trained network.

Instead, the pretrained model itself is adjusted to the new data.

#3) Multi-task learning



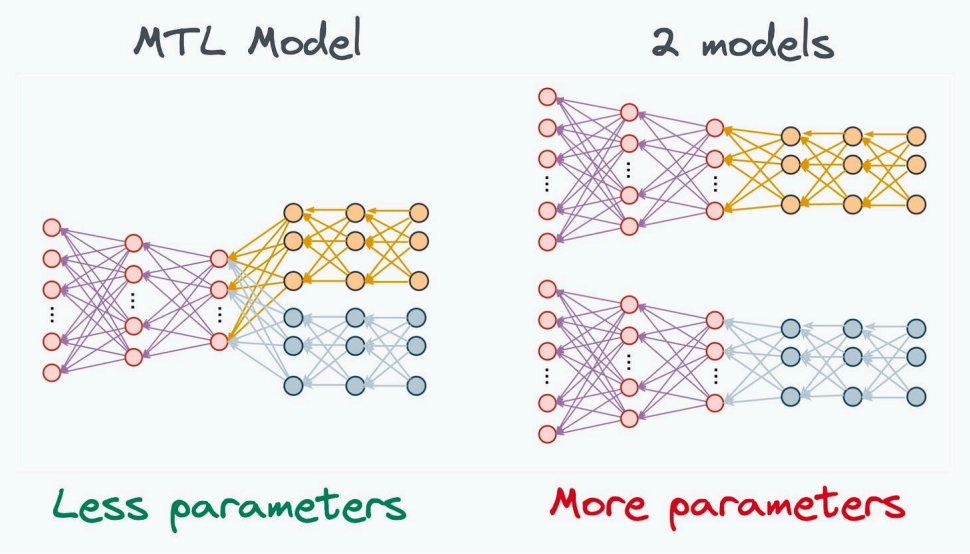
As the name suggests, a model is trained to perform multiple tasks simultaneously.

The model shares knowledge across tasks, aiming to improve generalization and performance on each task.

It can help in scenarios where tasks are related, or they can benefit from shared representations.

In fact, the motive for multi-task learning is not just to improve generalization.

We can also save compute power during training by having a shared layer and task-specific segments.



- Imagine training two models independently on related tasks.
- Now compare it to having a network with shared layers and then task-specific branches.

Option 2 will typically result in:

- Better generalization across all tasks.
- Less memory utilization to store model weights.
- Less resource utilization during training.

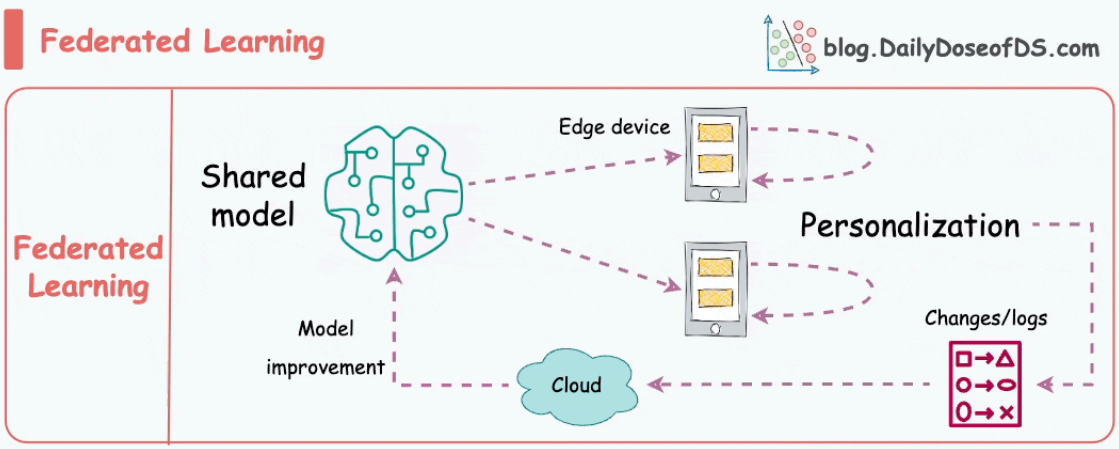
#4) Federated learning

Let's discuss it in the next chapter.

Introduction to Federated Learning

In my opinion, federated learning is among those very powerful ML techniques that is not given the true attention it deserves.

Here's a visual that depicts how it works:



Let's understand this topic in this chapter and why I consider this to be an immensely valuable skill to have.

The problem

Modern devices (like smartphones) have access to a wealth of data that can be suitable for ML models.

To get some perspective, consider the number of images you have on your phone right now, the number of keystrokes you press daily, etc.

That's plenty of data, isn't it?

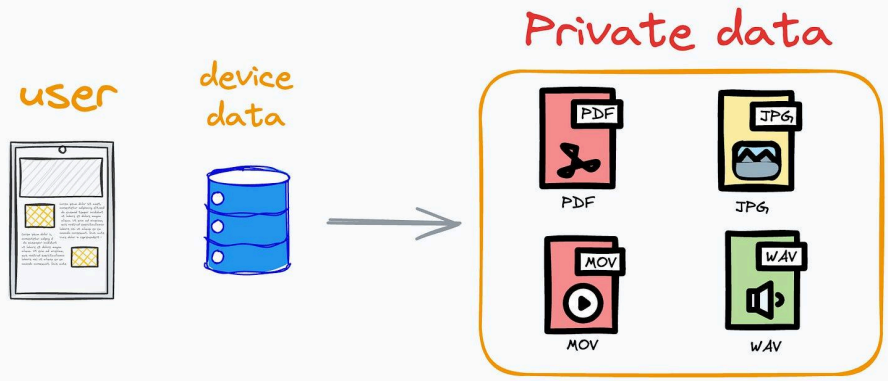
And this is just about one user — you.

But applications can have millions of users. The amount of data we can train ML models on is unfathomable.

So what is the problem here?

The problem is that almost all data available on modern devices is private.

- Images are private.
- Messages you send are private.
- Voice notes are private.



Being private, it is likely that it cannot be aggregated in a central location, as traditionally, ML models are always trained on centrally located datasets.

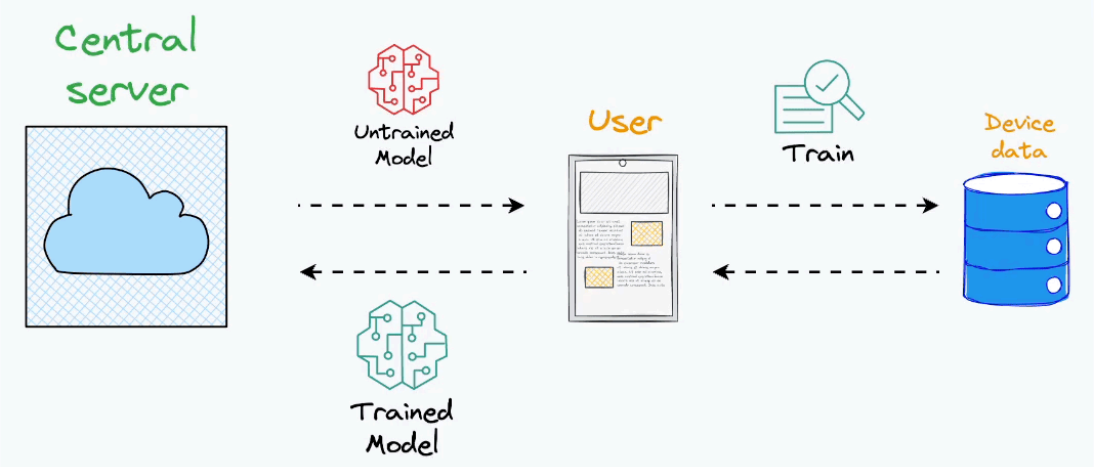
But this data is still valuable to us, isn't it?

We want to utilize it in some way.

The solution

Federated learning smartly addresses this challenge of training ML models on private data.

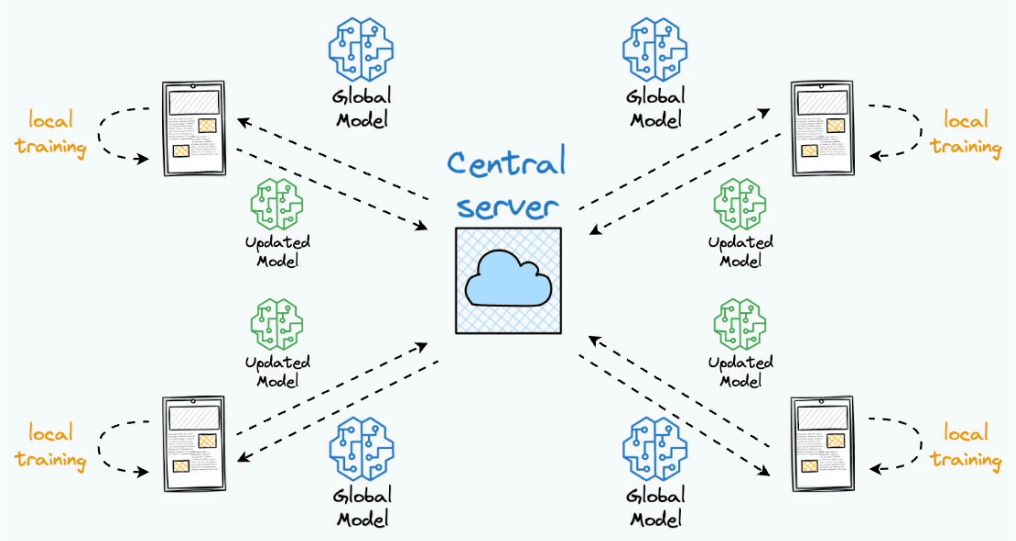
Here's the core idea:



- Instead of aggregating data on a central server, dispatch a model to an end device.
- Train the model on the user's private data on their device.
- Fetch the trained model back to the central server.
- Aggregate all models obtained from all end devices to form a complete model.

That's an innovative solution because each client possesses a local training dataset that remains exclusively on their device and is never uploaded to the server.

Yet, we still get to train a model on this private data.



Send a global model to the user's device, train a model on private data, and retrieve it back.

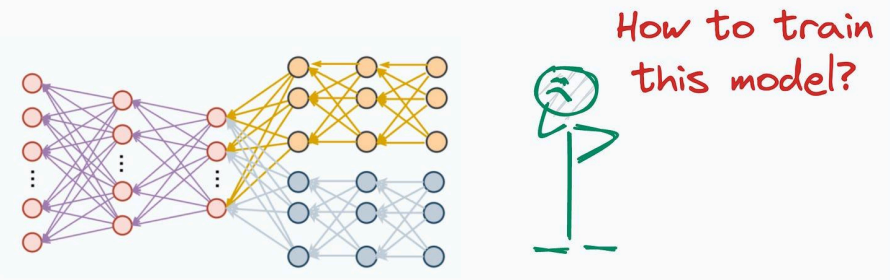
Furthermore, federated learning distributes most computation to a user's device.

As a result, the central server does not need the enormous computing that it would have demanded otherwise.

This is the core idea behind federated learning.

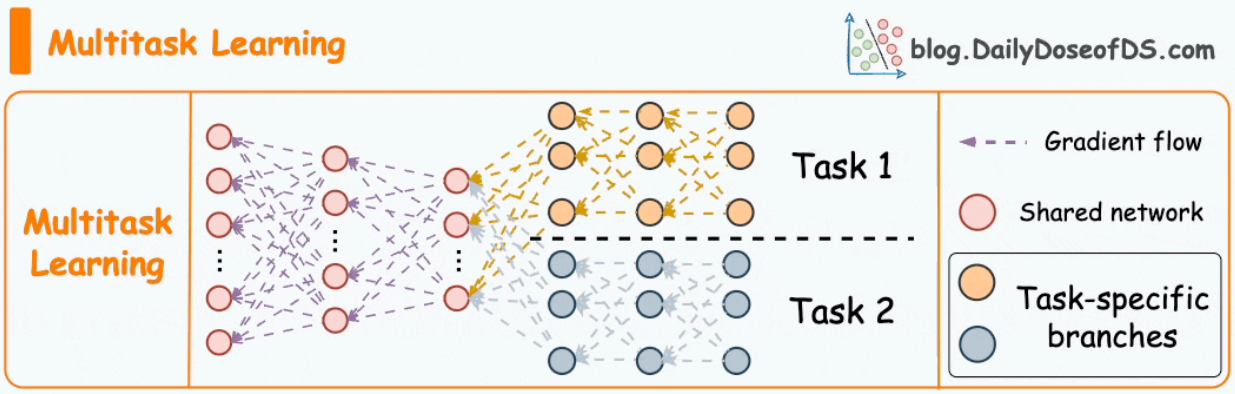
Building Multi-task Learning (MTL) Models

Most ML models are trained on one task. As a result, many struggle to intuitively understand how a model can be trained on multiple tasks simultaneously.



So let's discuss it in this chapter.

To reiterate, in MTL, the network has a few shared layers and task-specific segments. During backpropagation, gradients are accumulated from all branches, as depicted in the animation below:



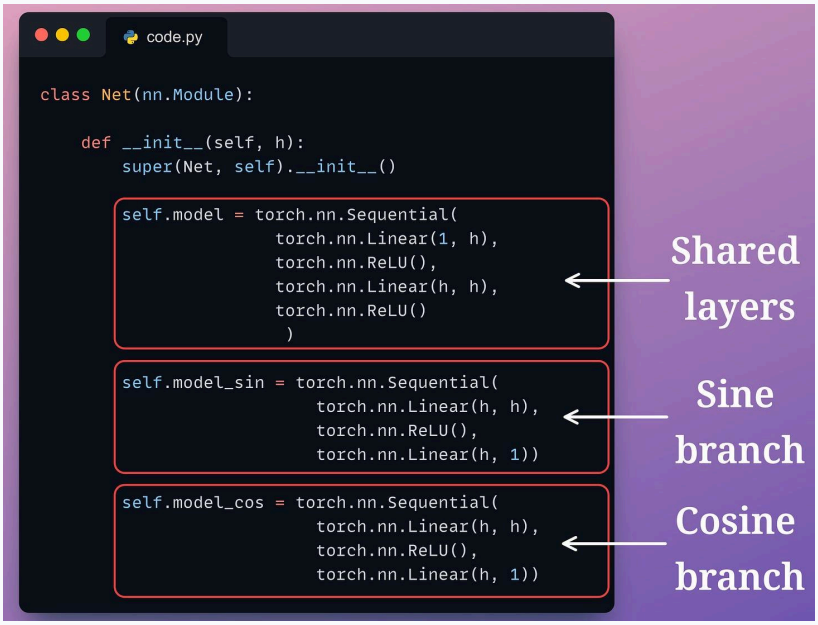
Let's take a simple example to understand its implementation.

Consider we want our model to take a real value (x) as input and generate two outputs:

- sin(x)
- cos(x)

This can be formulated as an MTL problem.

First, we define our model class using PyTorch.



- We have some fully connected layers in self.model → These are the shared layers.
- Furthermore, we have the output-specific layers to predict sin(x) and cos(x).

Next, let's define the forward pass in the class above:

- First, we pass the input through the shared layers (self.model).
- The output of the shared layers is passed through the sin and cos branches.
- We return the output from both branches.

We are almost done. The final part of this implementation is to train the model. Let's use mean squared error as the loss function. The training loop is implemented below:

- We pass the input data through the model.
- It returns two outputs, one from each segment of the network.
- We compute the branch-specific loss values (loss1 and loss2) using true predictions.
- We add the two loss values to get the total loss for the network.
- Finally, we run the backward pass.

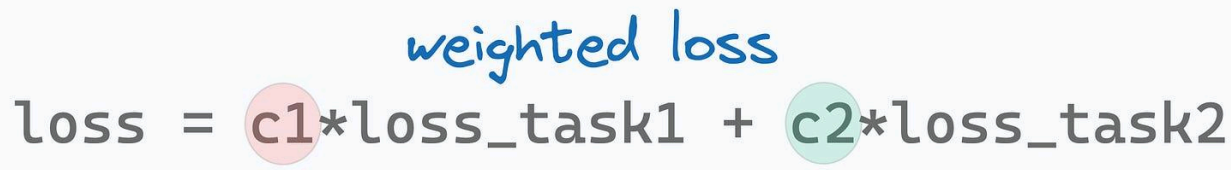
With this, we have trained our MTL model. Also, we get a decreasing loss, which depicts that the model is being trained.

And that's how we train an MTL model. You can extend the same idea to build any MTL model of your choice.

Do remember that building an MTL model on unrelated tasks will not produce good results.

Thus, "task-relatedness" is a critical component of all MTL models because of the shared layers. Also, it is NOT necessary that every task must equally contribute to the entire network's loss.

We may assign weights to each task as well, as depicted below:



The weights could be based on task importance.

Or...

At times, I also use dynamic task weights, which could be inversely proportional to the validation accuracy achieved on that task.

$$c1 \propto \frac{1}{\text{accuracy\_task1}} \qquad c2 \propto \frac{1}{\text{accuracy\_task2}}$$

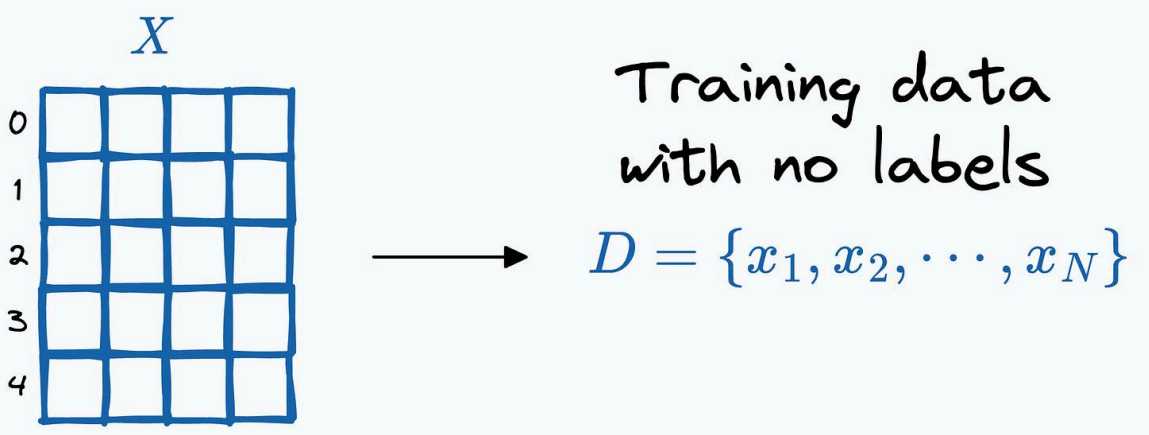
My rationale behind this technique is that in an MTL setting, some tasks can be easy while others can be difficult.

If the model achieves high accuracy on one task during training, we can safely reduce its loss contribution so that the model focuses more on the second task.

You can download the notebook for this chapter here: https://bit.ly/3ztY5hy.

Active Learning

There's not much we can do to build a supervised system when the data we begin with is unlabeled.



Using unsupervised techniques (if they fit the task) can be a solution, but supervised systems are typically on par with unsupervised ones.

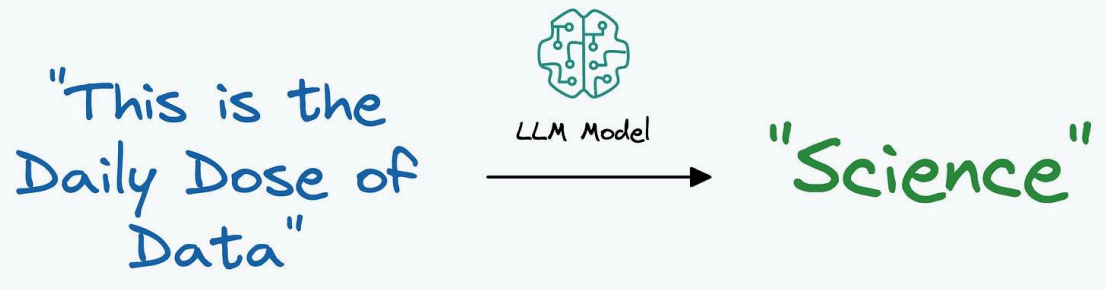
Another way, if feasible, is to rely on self-supervised learning.

Self-supervised learning is when we have an unlabeled dataset (say text data), but we somehow figure out a way to build a supervised learning model out of it.

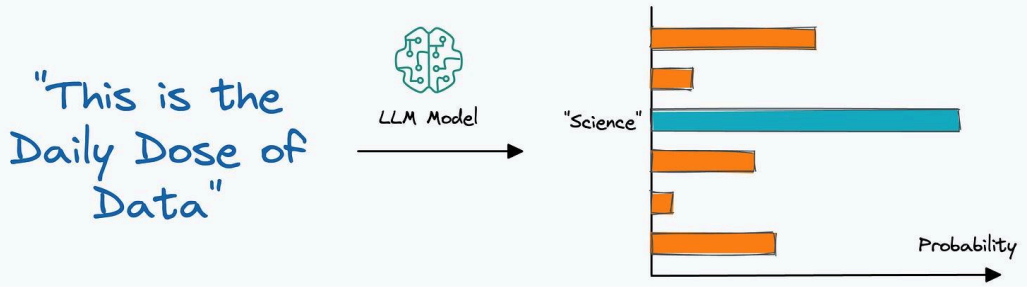
This becomes possible due to the inherent nature of the task.

Consider an LLM, for instance.

In a nutshell, its core objective is to predict the next token based on previously predicted tokens (or the given context).

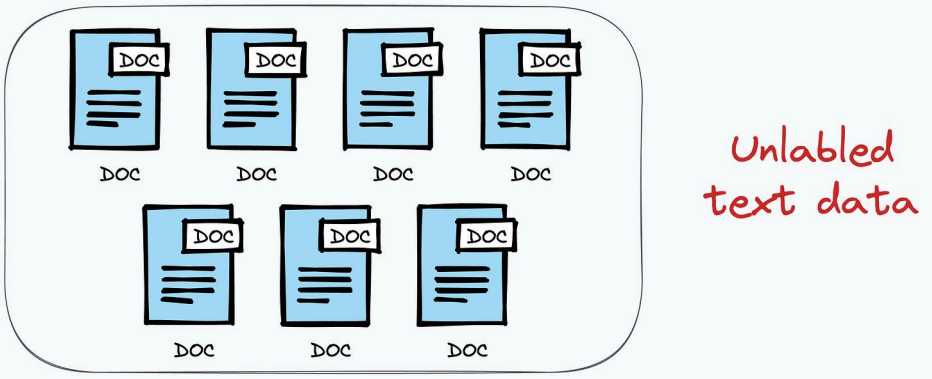


This is a classification task, and the labels are tokens.



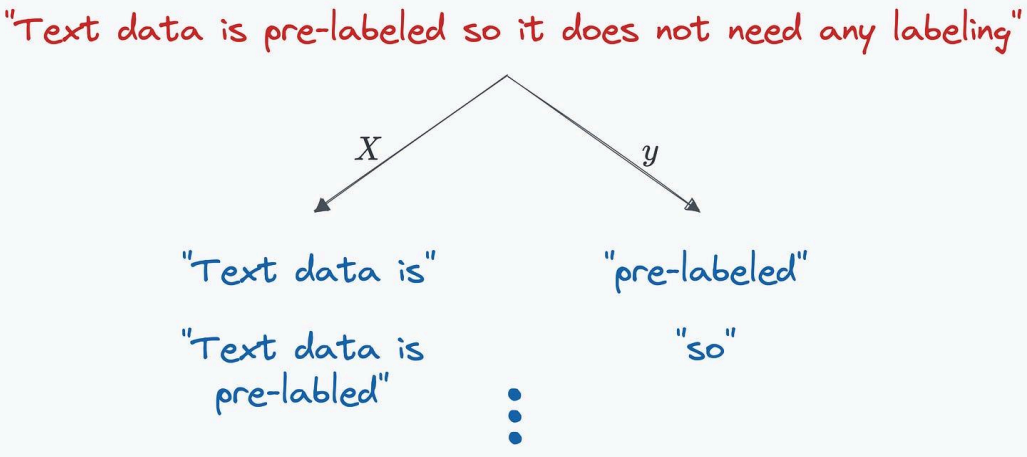
But text data is raw. It has no labels.

Then how did we train this classification task?



Self-supervised techniques solve this problem.

Due to the inherent nature of the task (next-token prediction, to be specific), every piece of raw text data is already self-labeled.



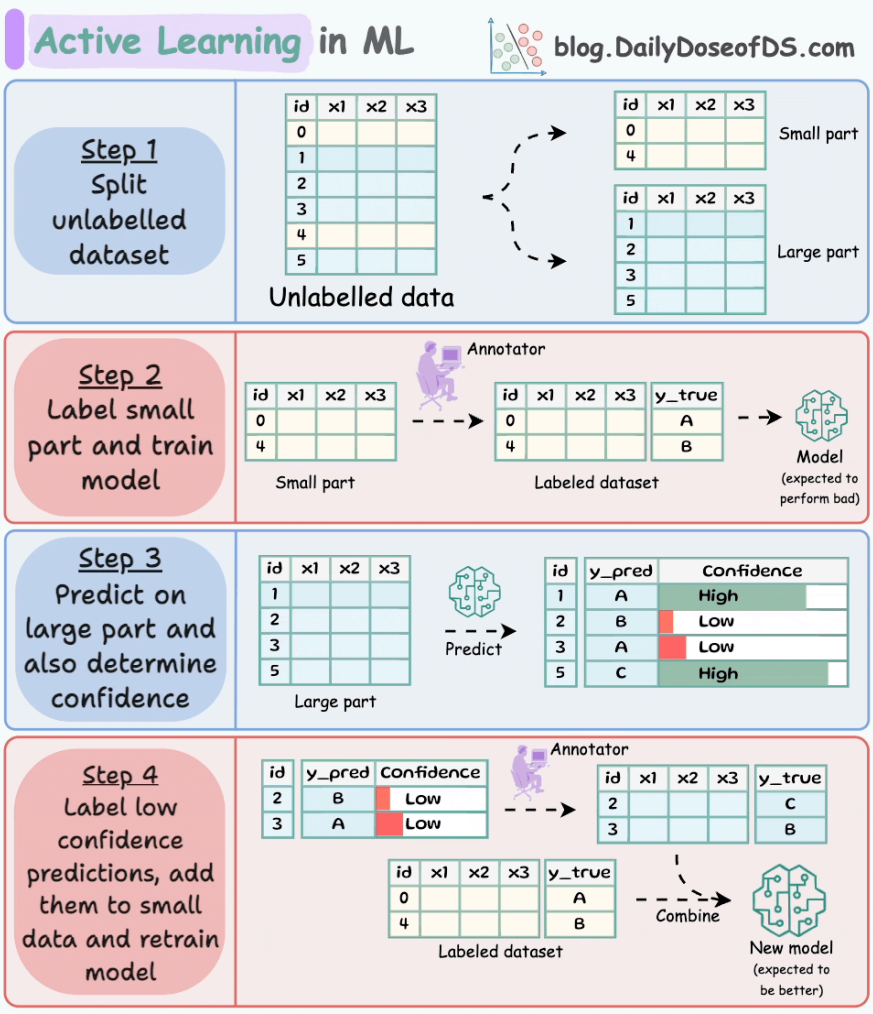
The model is only supposed to learn the mapping from previous tokens to the next token.

This is called self-supervised learning, which is quite promising, but it has limited applicability, largely depending on the task.

At this stage, the only possibility one notices is annotating the dataset. However, data annotation is difficult, expensive, time-consuming, and tedious.

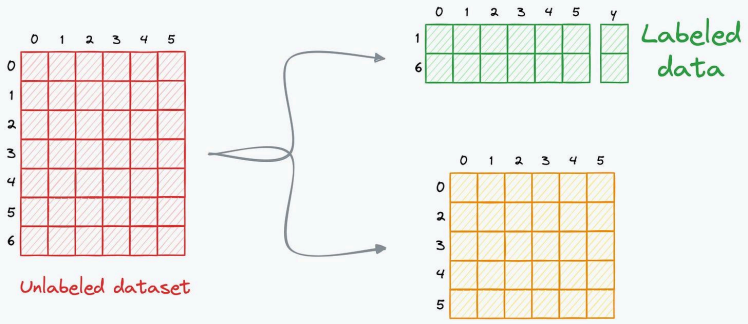
Active learning is a relatively easy, inexpensive, quick, and interesting way to address this.

As the name suggests, the idea is to build the model with active human feedback on examples it is struggling with. The visual below summarizes this:



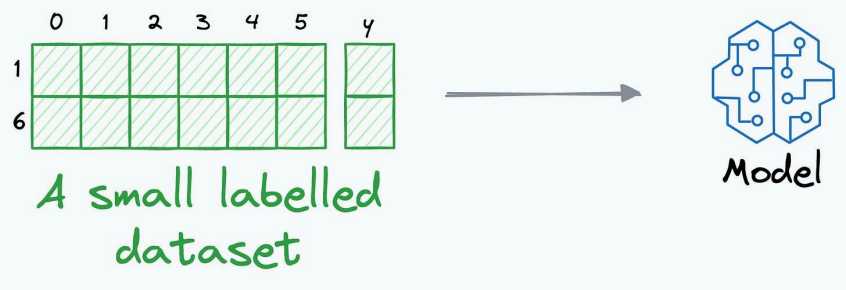
Let's get into the details.

We begin by manually labeling a tiny percentage of the dataset.

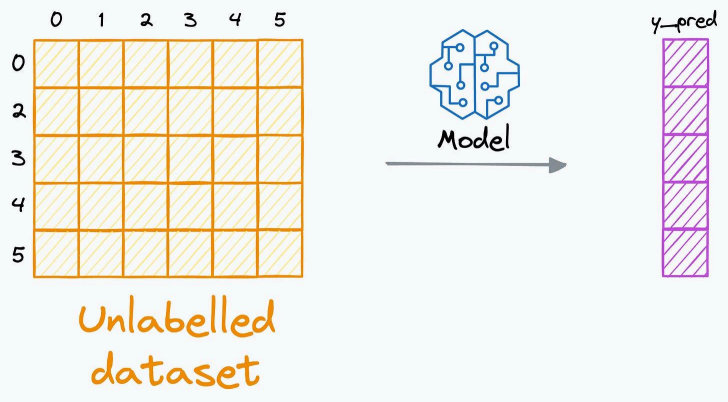


While there's no rule on how much data should be labeled, I have used active learning (successfully) while labeling as low as ~1% of the dataset, so try something in that range.

Next, build a model on this small labeled dataset.



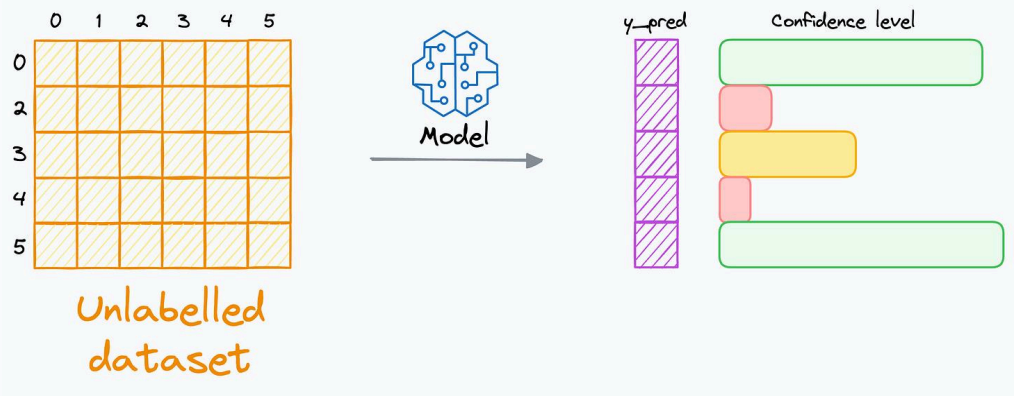
Of course, this won't be a perfect model, but that's okay. Next, generate predictions on the dataset we did not label:



It's obvious that we cannot determine if these predictions are correct as we do not have any labels.

That's why we need to be a bit selective with the type of model we choose.

More specifically, we need a model that, either implicitly or explicitly, can also provide a confidence level with its predictions.

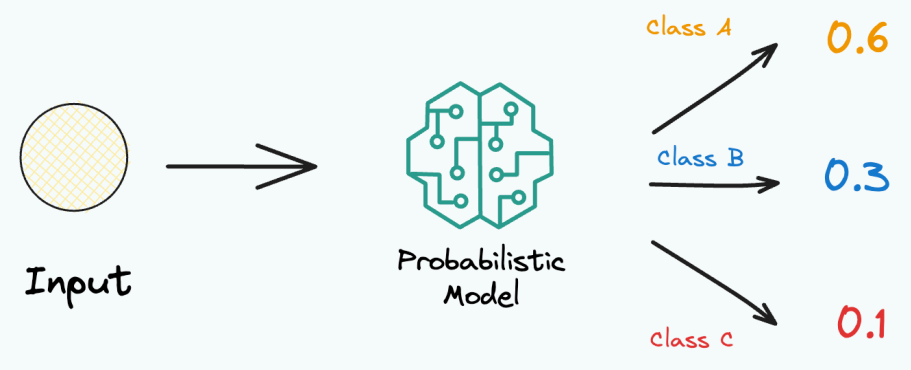


As the name suggests, a confidence level reflects the model's confidence in generating a prediction.

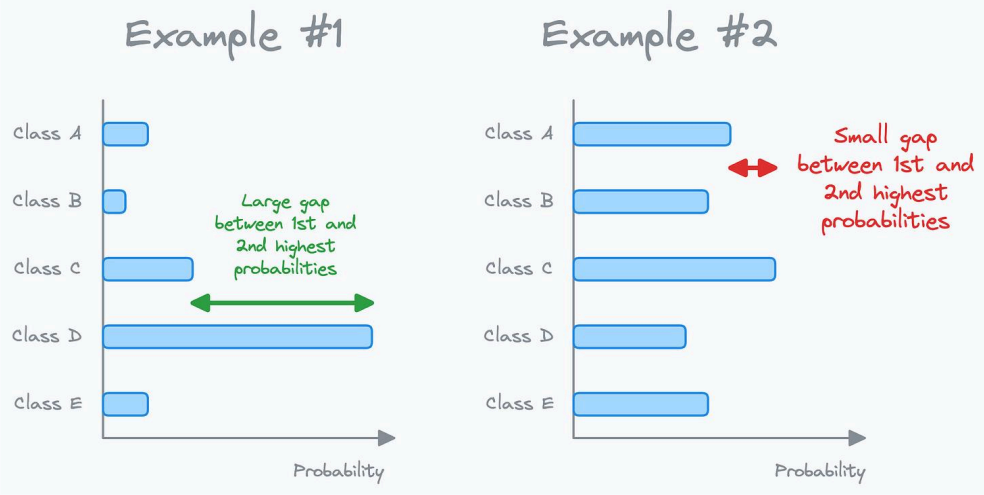
If a model could speak, it would be like:

- I am predicting a "cat" and am 95% confident about my prediction.
- I am predicting a "cat" and am 5% confident about my prediction.
- And so on...

Probabilistic models (ones that provide a probabilistic estimate for each class) are typically a good fit here.



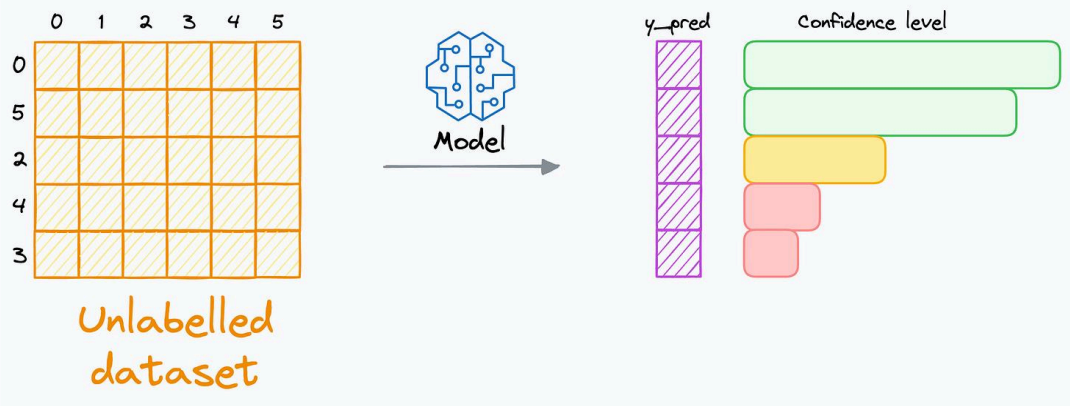
This is because one can determine a proxy for confidence level from probabilistic outputs.



In the above two examples, consider the gap between 1st and 2nd highest probabilities:

- In example #1, the gap is large. This can indicate that the model is quite confident in its prediction.
- In example #2, the gap is small. This can indicate that the model is NOT quite confident in its prediction.

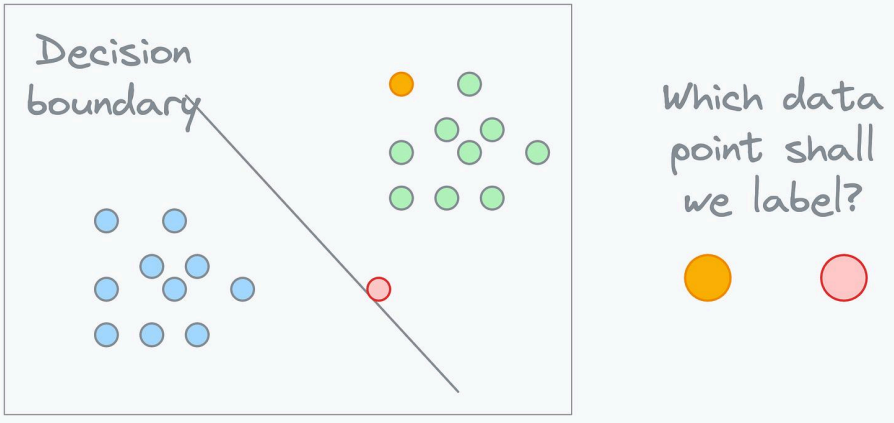
Now, go back to the predictions generated above and rank them in order of confidence:



In the above image:

- The model is already quite confident with the first two instances. There's no point checking those.
- Instead, it would be best if we (the human) annotate the instances with which it is least confident.

To get some more perspective, consider the image below. Logically speaking, which data point's human label will provide more information to the model? I know you already know the answer.



Thus, in the next step, we provide our human label to the low-confidence predictions and feed it back to the model with the previously labeled dataset:

Repeat this a few times and stop when you are satisfied with the performance.

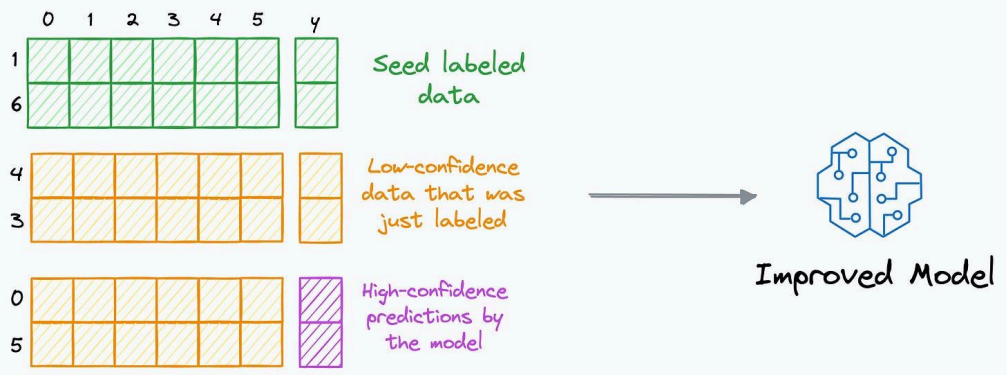
In my experience, active learning has always been an immensely time-saving approach to building supervised models on unlabeled datasets.

The only thing that you have to be careful about is generating confidence measures.

If you mess this up, it will affect every subsequent training step.

There's one more thing I like to do when using active learning.

While combining the low-confidence data with the seed data, we can also use the high-confidence data. The labels would be the model's predictions.

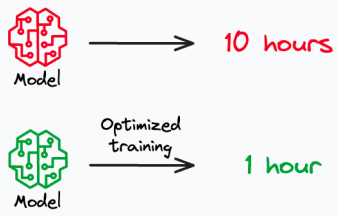


This variant of active learning is called cooperative learning.

Run-time and Memory Optimization

Momentum

As we progress towards building larger and larger models, every bit of possible optimization becomes crucial.



And, of course, there are various ways to speed up model training, like:

- Batch processing
- Leverage distributed training using frameworks like PySpark MLLib.
- Use better Hyperparameter Optimization, like Bayesian Optimization, which we will discuss in this chapter.
- and many other techniques.

Momentum is another reliable and effective technique to speed up model training. While Momentum is pretty popular, many people struggle to intuitively understand how it works and why it is effective. Let's understand in this chapter.

Issues with Gradient Descent

In gradient descent, every parameter update solely depends on the current gradient. This is clear from the gradient weight update rule shown below:

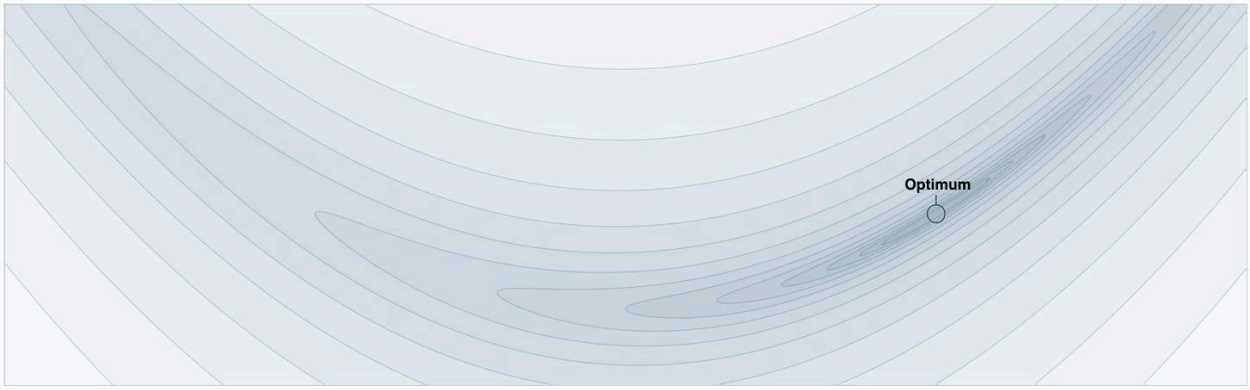
$$\theta^t = \theta^{t-1} - \alpha \frac{\delta J}{\delta \theta}$$

Only dependent on current gradient

As a result, we end up having many unwanted oscillations during the optimization process.

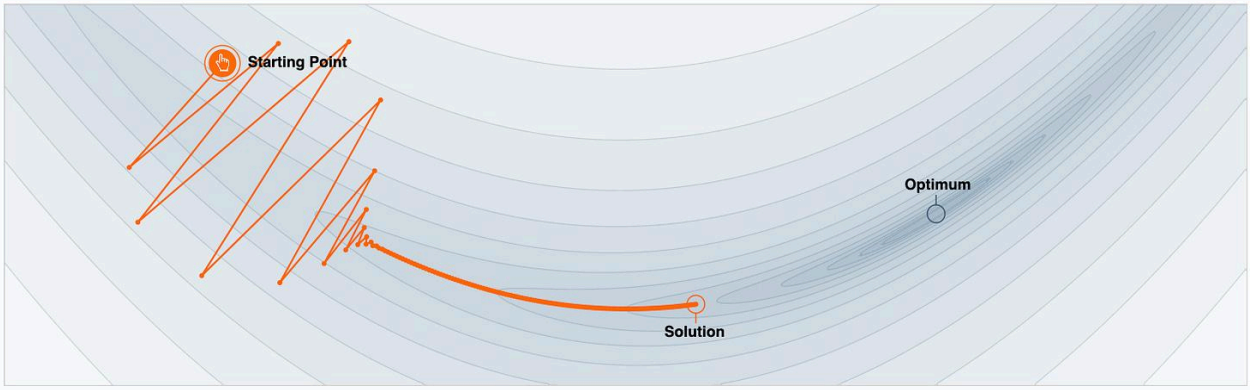
Let's understand this more visually.

Imagine this is the loss function contour plot, and the optimal location (parameter configuration where the loss function is minimum) is marked here:



Simply put, this plot illustrates how gradient descent moves towards the optimal solution. At each iteration, the algorithm calculates the gradient of the loss function at the current parameter values and updates the weights.

This is depicted below:



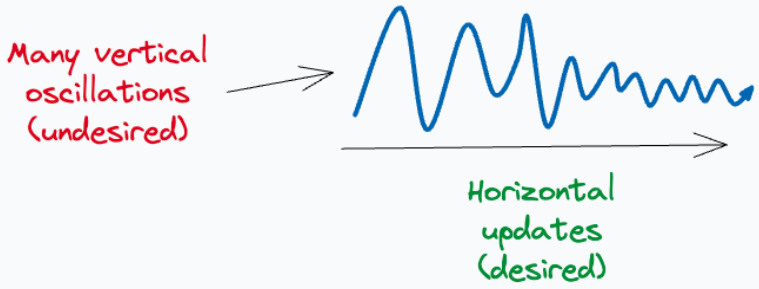
Notice two things here:

- It unnecessarily oscillates vertically.
- It ends up at the non-optimal solution after some epochs.

Ideally, we would have expected our weight updates to look this:

- It must have taken longer steps in the horizontal direction...
- ...and smaller vertical steps because a movement in this direction is unnecessary.

This idea is also depicted below:



How Momentum solves this problem?

Momentum-based optimization slightly modifies the update rule of gradient descent. More specifically, it also considers a moving average of past gradients:

$$\theta^t = \theta^{t-1} - \alpha\frac{\delta J}{\delta\theta} + \beta \cdot \boxed{F(\theta^{t-1}, \theta^{t-2}, \cdots)}$$

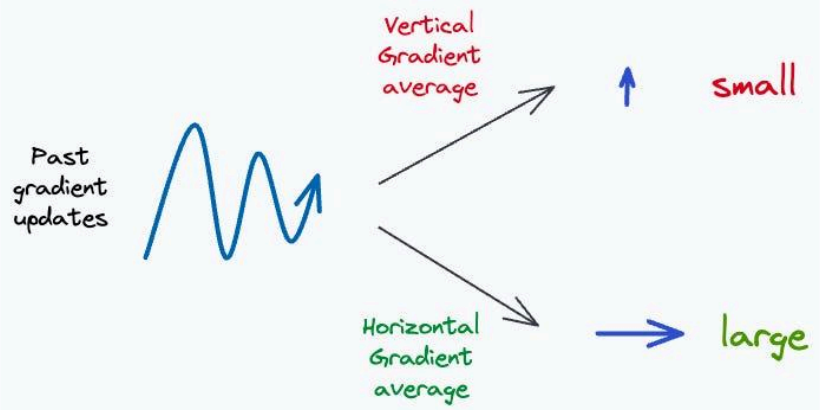
dependent on current gradient

dependent on previous updates

This helps us handle the unnecessary vertical oscillations we saw earlier.

How?

As Momentum considers a moving average of past gradients, so if the recent gradient update trajectory looks as shown in the following image, then it is clear that its average in the vertical direction will be very low while that in the horizontal direction will be large (which is precisely what we want):



As this moving average gets added to the gradient updates, it helps the optimization algorithm take larger steps in the desired direction.

$$\theta^t = \theta^{t-1} - \alpha\frac{\delta J}{\delta\theta} + \beta \cdot \boxed{F(\theta^{t-1}, \theta^{t-2}, \cdots)}$$

dependent on current gradient

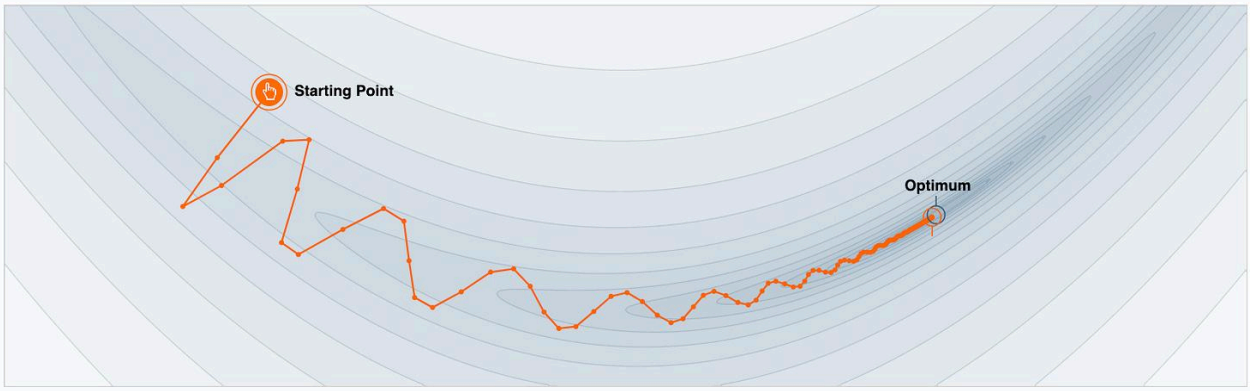
dependent on previous updates

This way, we can:

- Smoothen the optimization trajectory.

- Reduce unnecessary oscillations in parameter updates, which also speeds up training.

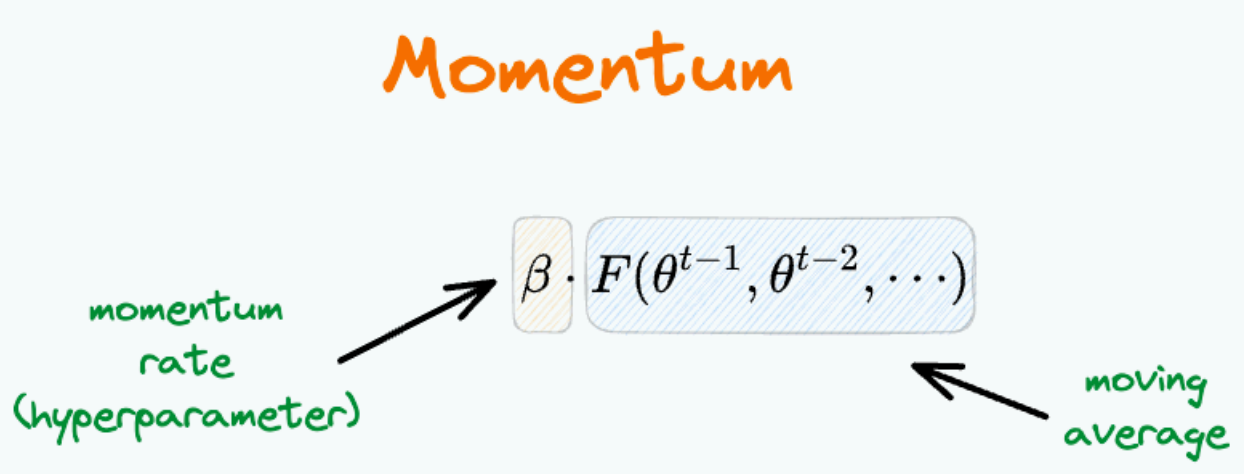
This is also evident from the image below:



This time, the gradient update trajectory shows much smaller oscillations in the vertical direction, and it also manages to reach an optimum under the same number of epochs as earlier.

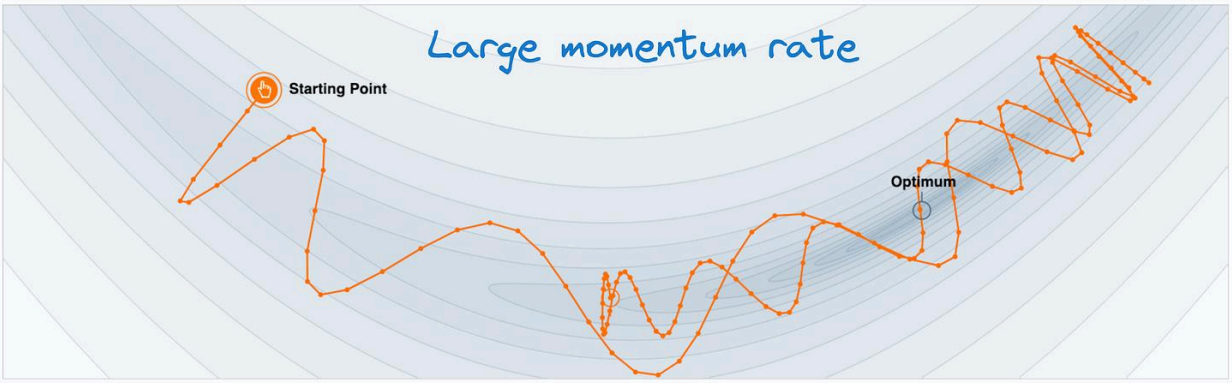
This is the core idea behind Momentum and how it works.

Of course, Momentum does introduce another hyperparameter (Momentum rate) in the model, which should be tuned appropriately like any other hyperparameter:

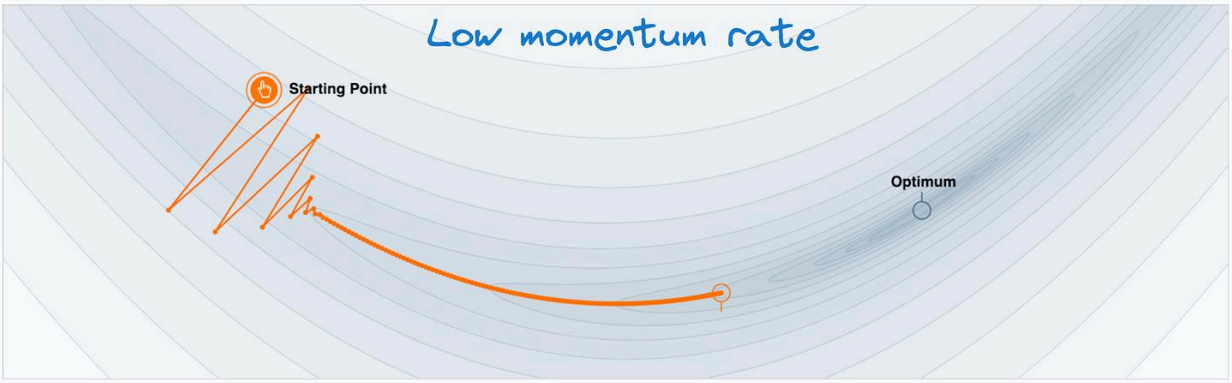


For instance, considering the 2D contours we discussed above:

- Setting an extremely large value of Momentum rate will significantly expedite gradient update in the horizontal direction. This may lead to overshooting the minima, as depicted below:



- What's more, setting an extremely small value of Momentum will slow down the optimal gradient update, defeating the whole purpose of Momentum.



If you want to have a more hands-on experience, check out this tool: https://bit.ly/4cOrJN1.

Mixed Precision Training

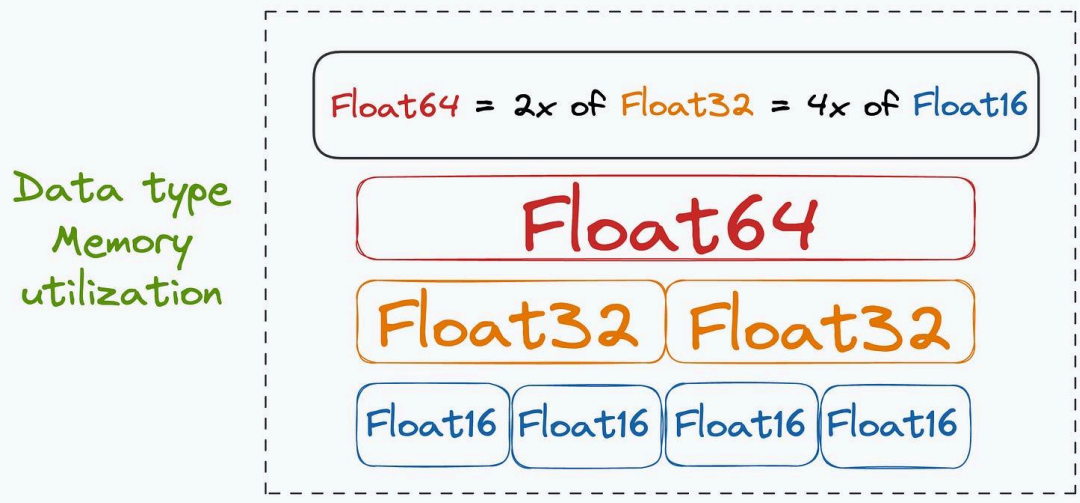
Context

Typical deep learning libraries are really conservative when it comes to assigning data types.

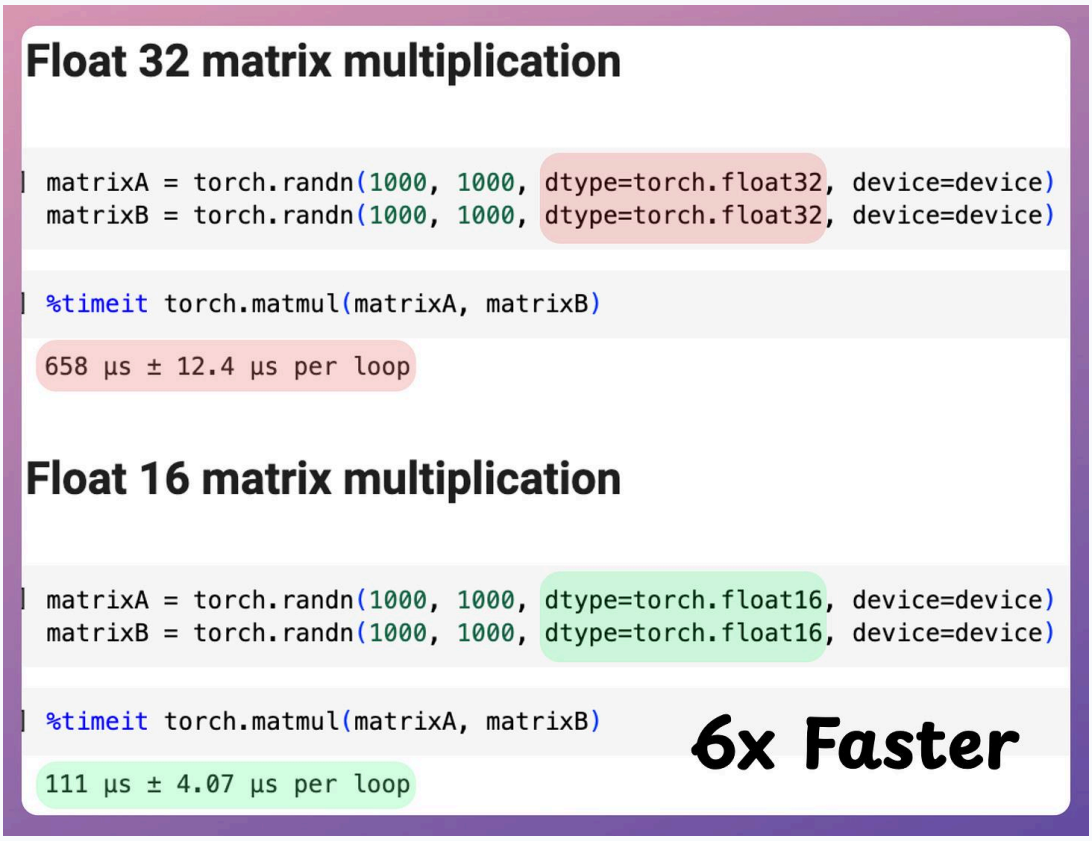
The data type assigned by default is usually 64-bit or 32-bit, when there is also scope for 16-bit, for instance. This is also evident from the code below:

As a result, we are not entirely optimal at efficiently allocating memory. Of course, this is done to ensure better precision in representing information.

However, this precision always comes at the cost of additional memory utilization, which may not be desired in all situations.



In fact, it is also observed that many tensor operations, especially matrix multiplication, are much faster when we operate under smaller precision data types than larger ones, as demonstrated below:

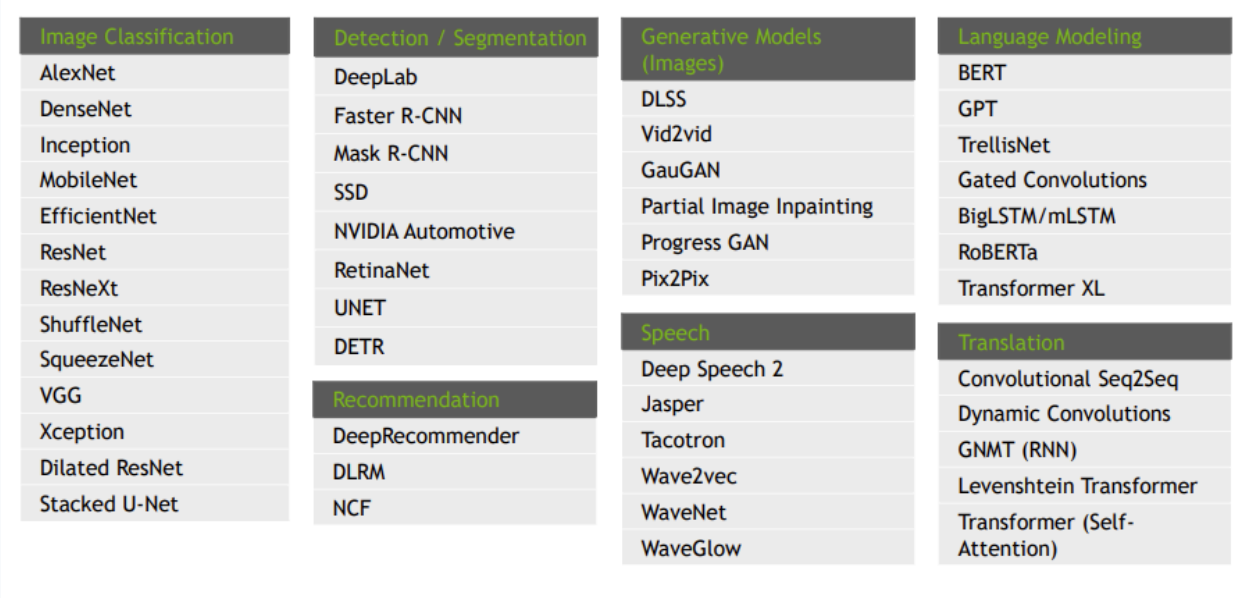


Moreover, since `float16` is only half the size of `float32`, its usage reduces the memory required to train the network. This also allows us to train larger models, train on larger mini-batches (resulting in even more speedup), etc.

Mixed precision training is a pretty reliable and widely adopted technique in the industry to achieve this.

As the name suggests, the idea is to employ lower precision `float16` (wherever feasible, like in convolutions and matrix multiplications) along with `float32` — that is why the name "mixed precision."

This is a list of some models I found that were trained using mixed precision:



It's pretty clear that mixed precision training is much more popularly used, but we don't get to hear about it often.

Before we get into the technical details...

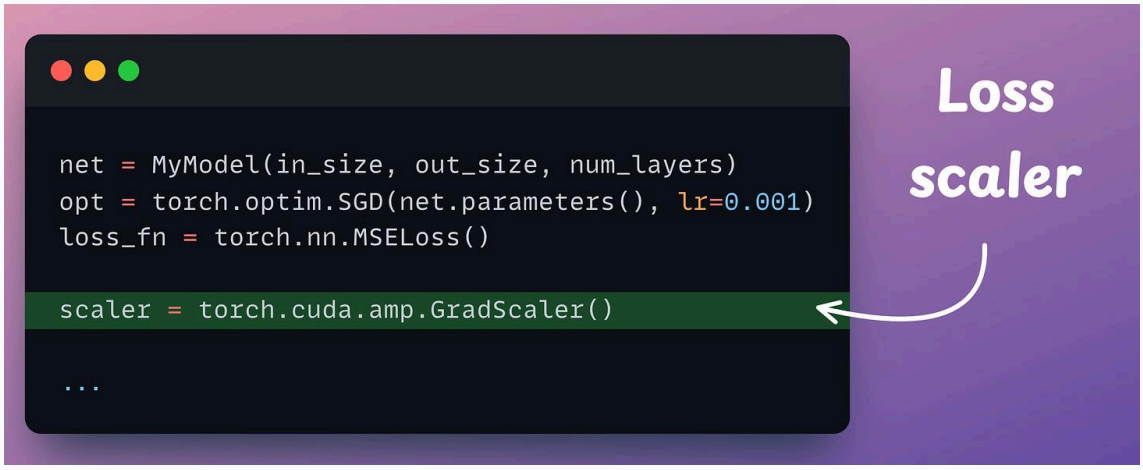
From the above discussion, it must be clear that as we use a low-precision data type (`float16`), we might unknowingly introduce some numerical inconsistencies and inaccuracies.

To avoid them, there are some best practices for mixed precision training that I want to talk about next, along with the code.

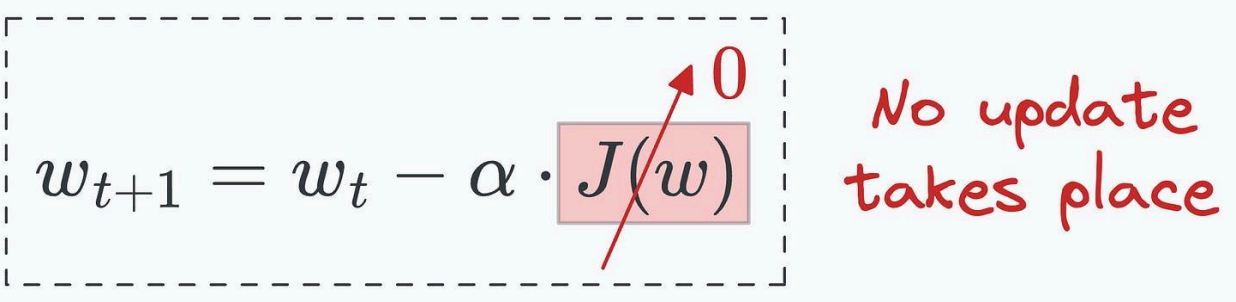
Mixed precision training in PyTorch and Best Practices

Leveraging mixed precision training in PyTorch requires a few modifications in the existing network training implementation. Consider this is our current PyTorch model training implementation:

The first thing we introduce here is a scaler object that will scale the loss value:



We do this because, at times, the original loss value can be so low, that we might not be able to compute gradients in float16 with full precision. Such situations may not produce any update to the model's weights.

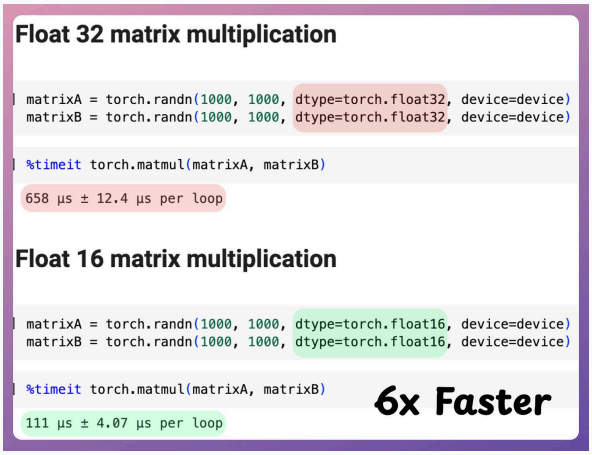


Scaling the loss to a higher numerical range ensures that even small gradients can contribute to the weight updates.

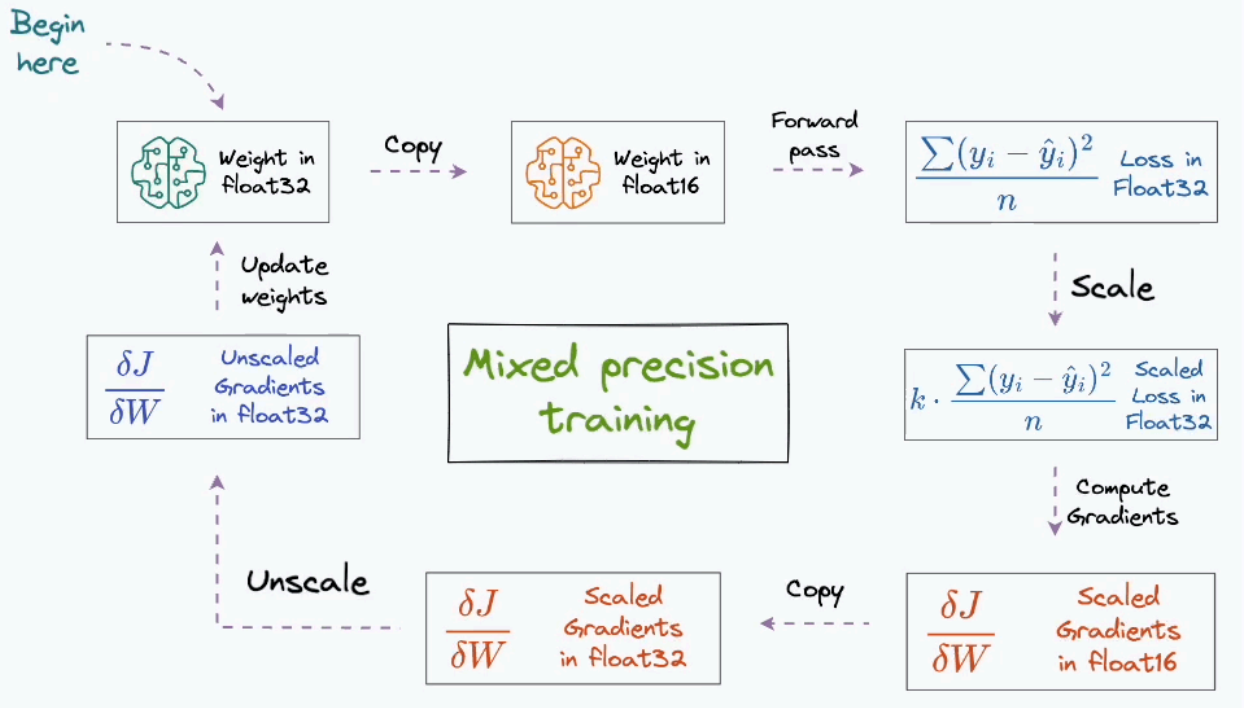
But these minute gradients can only be accommodated into the weight matrix when the weight matrix itself is represented in high precision, i.e., float32. Thus, as a conservative measure, we tend to keep the weights in float32.

That said, the loss scaling step is not entirely necessary because, in my experience, these little updates typically appear towards the end stages of the model training. Thus, it can be fair to assume that small updates may not drastically impact the model performance. But don't take this as a definite conclusion, so it's something that I want you to validate when you use mixed precision training.

Moving on, as the weights (which are matrices) are represented in float32, we can not expect the speedup from representing them in float16, if they remain this way:



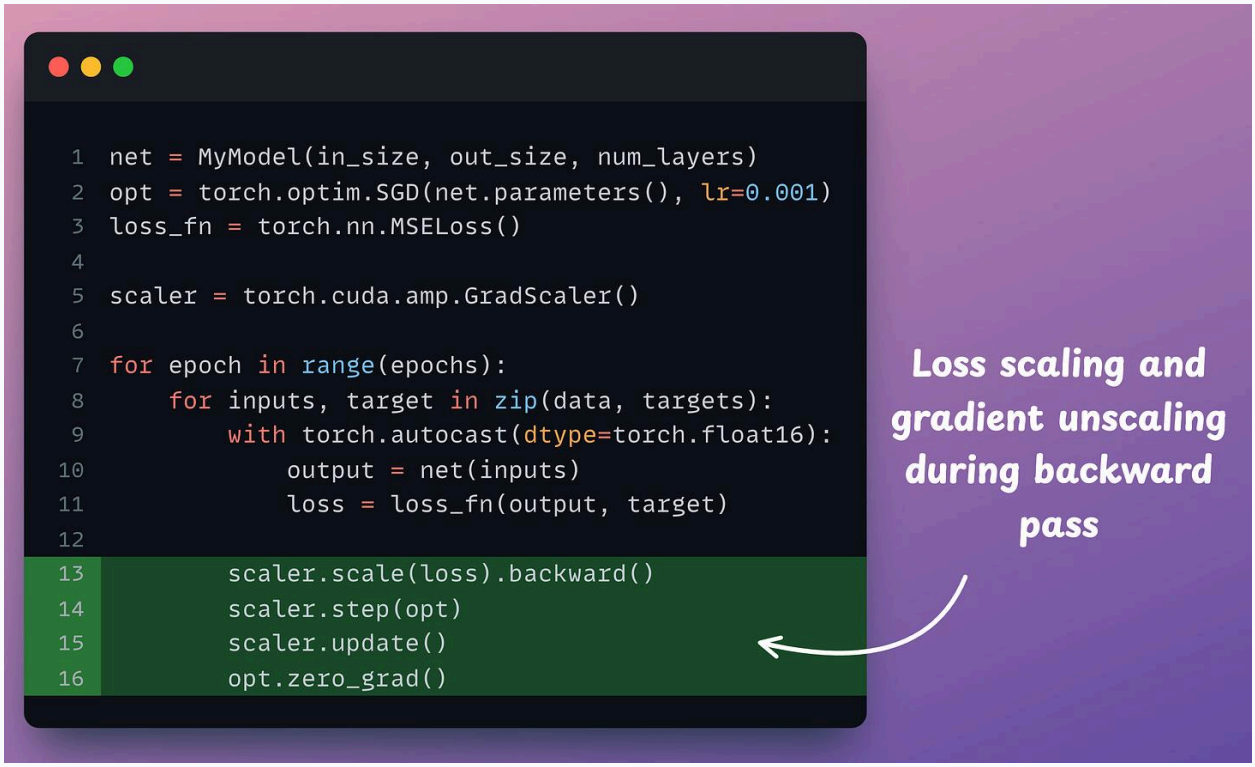
To leverage these flaot16-based speedups, here are the steps we follow:



1. We make a `float16` copy of weights during the forward pass.
2. Next, we compute the loss value in `float32` and scale it to have more precision in gradients, which works in `float16`.
    a. The reason we compute gradients in `float16` is because, like forward pass, gradient computations also involve matrix multiplications.
    b. Thus, keeping them in `float16` can provide additional speedup.
3. Once we have computed the gradients in `float16`, the heavy matrix multiplication operations have been completed. Now, all we need to do is update the original weight matrix, which is in `float32`.
4. Thus, we make a `float32` copy of the above gradients, remove the scale we applied in Step 2, and update the `float32` weights.
5. Done!

The mixed-precision settings in the forward pass are carried out by the `torch.autocast()` context manager:

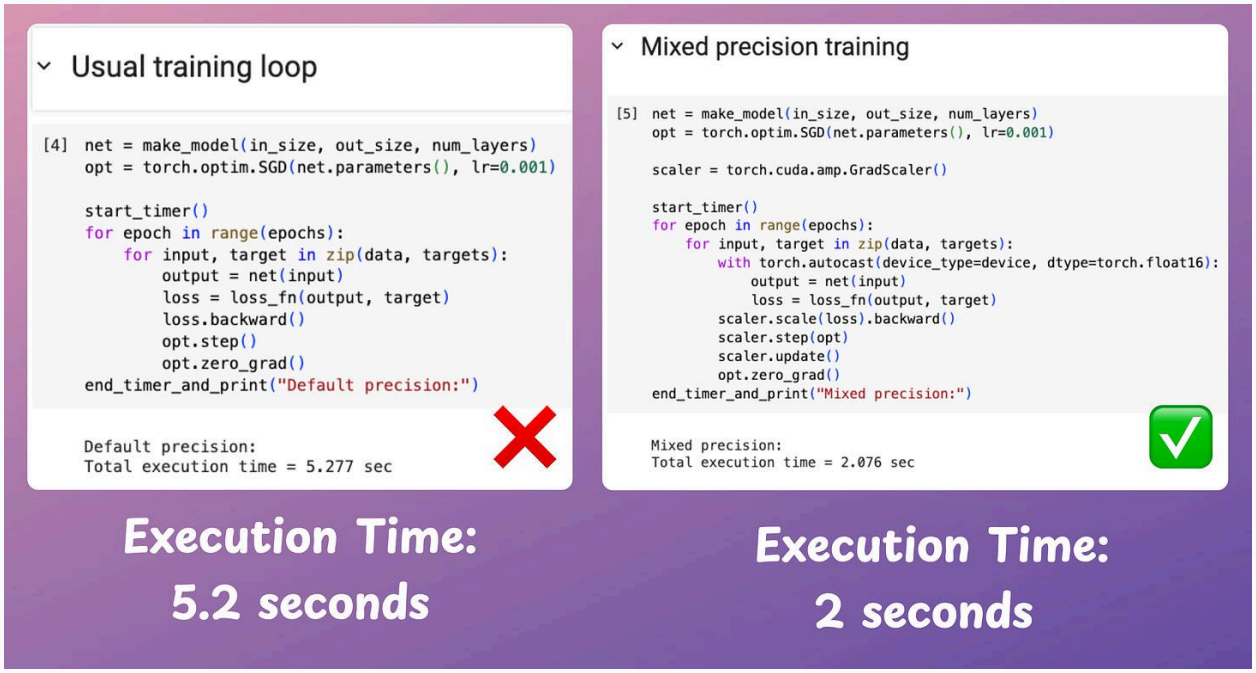
Now, it's time to handle the backward pass.



- Line 13 → `scaler.scale(loss).backward()`: The scaler object scales the loss value and `backward()` is called to compute the gradients.
- Line 14 → `scaler.step(opt)`: Unscale gradients and update weights.
- Line 15 → `scaler.update()`: Update the scale for the next iteration.
- Line 16 → `opt.zero_grad()`: Zero gradients.

Done!

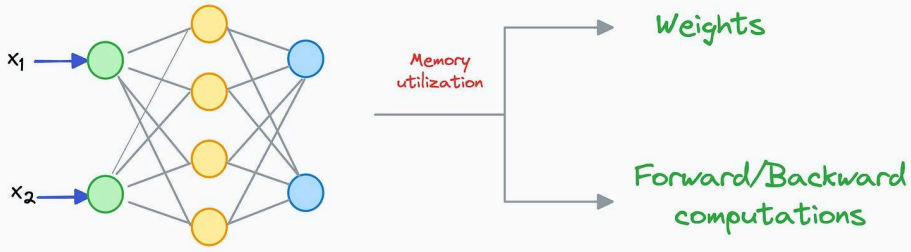
The efficacy of mixed precision scaling over traditional training is evident from the image below:



Mixed precision training is over 2.5x faster than conventional training.

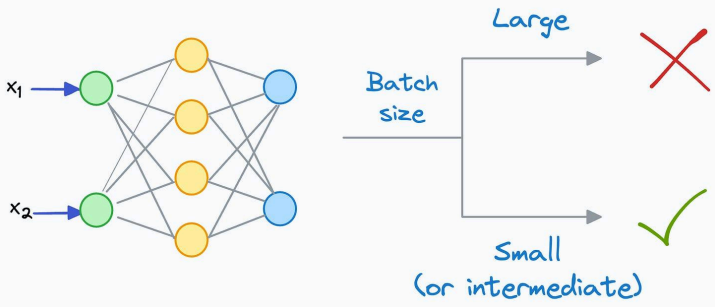
Gradient Checkpointing

Neural networks primarily utilize memory in two ways:



1. When they store model weights (this is fixed memory utilization).
2. When they are trained (this is dynamic). It happens in two ways:
    a. During forward pass while computing and storing activations of all layers.
    b. During backward pass while computing gradients at each layer.

The latter, i.e., dynamic memory utilization, often restricts us from training larger models with bigger batch sizes.



This is because memory utilization scales proportionately with the batch size.

That said, there's a pretty incredible technique that lets us increase the batch size while maintaining the overall memory utilization.

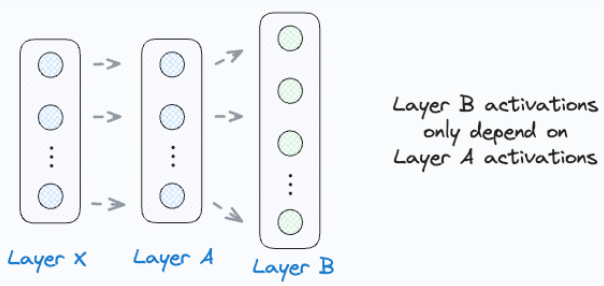
It is called Gradient checkpointing, and in my experience, it's a highly underrated technique to reduce the memory overheads of neural networks.

Let's understand this in more detail.

How gradient checkpointing works?

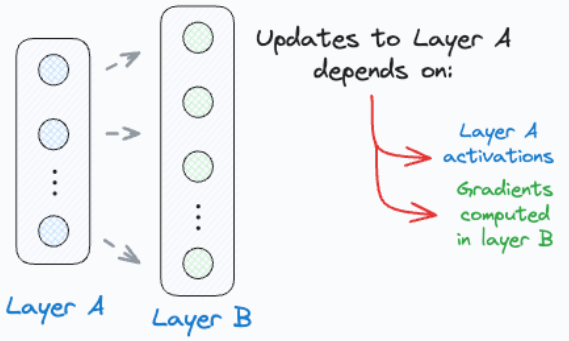
Gradient checkpointing is based on two key observations on how neural networks typically work:

1) The activations of a specific layer can be solely computed using the activations of the previous layer. For instance, in the image below, "Layer B" activations can be computed from "Layer A" activations only.



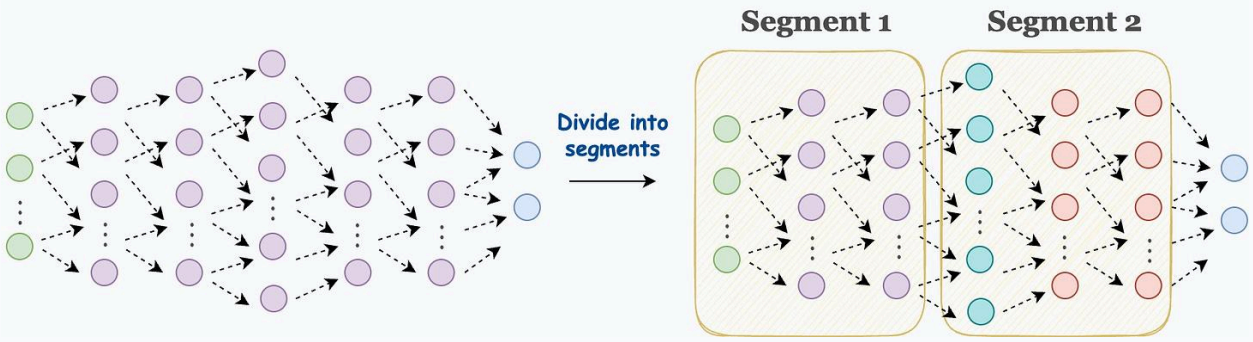
2) Updating the weights of a layer only depends on two things:

a. The activations of that layer.
b. The gradients computed in the next (right) layer (or rather, the running gradients).

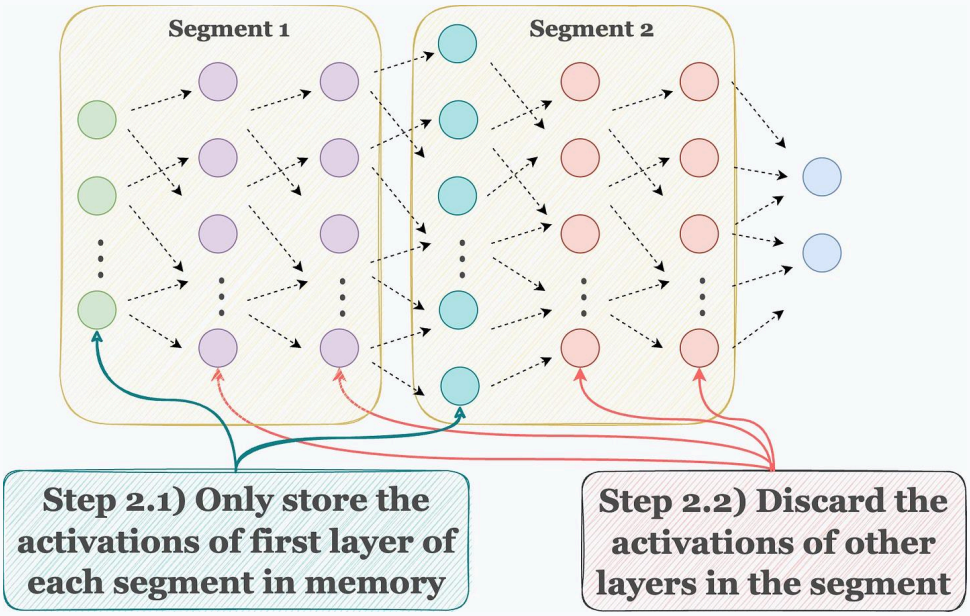


Gradient checkpointing exploits these two observations to optimize memory utilization. Here's how it works:

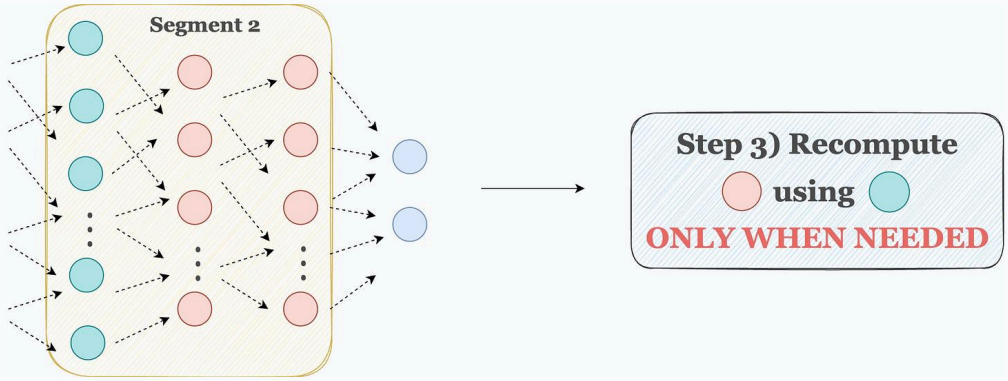
- Step 1) Divide the network into segments before the forward pass:



- Step 2) During the forward pass, only store the activations of the first layer in each segment. Discard the rest when they have been used to compute the activations of the next layer.



- Step 3) Now comes backpropagation. To update the weights of a layer, we need its activations. Thus, we recompute those activations using the first layer in that segment. For instance, as shown in the image below, to update the weights of the red layers, we recompute their activations using the activations of the cyan layer, which are already available in memory.

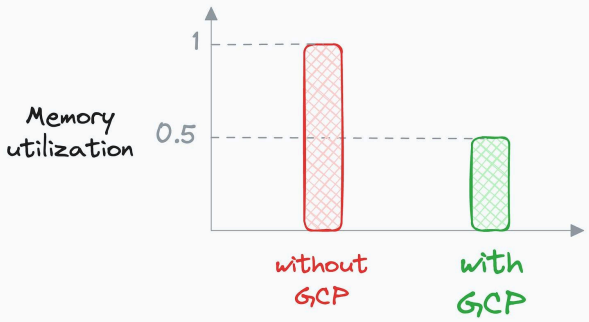


Done!

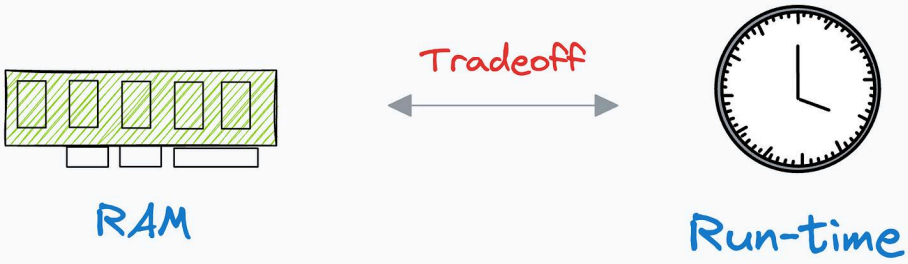
This is how gradient checkpointing works.

To summarize, the idea is that we don't need to store all the intermediate activations in memory. Instead, storing a few of them and recomputing the rest only when they are needed can significantly reduce the memory requirement. The whole idea makes intuitive sense as well.

This also allows us to train the network on larger batches of data. Typically, my observation has been that gradient checkpointing (GCP) can reduce memory usage by at least 50-60%, which is massive.



Of course, as we compute some activations twice, this does come at the cost of increased run-time, which can typically range between 15-25%. So there's always a tradeoff between memory and run-time.



That said, another advantage is that it allows us to use a larger batch size, which can slightly (not entirely though) counter the increased run-time.

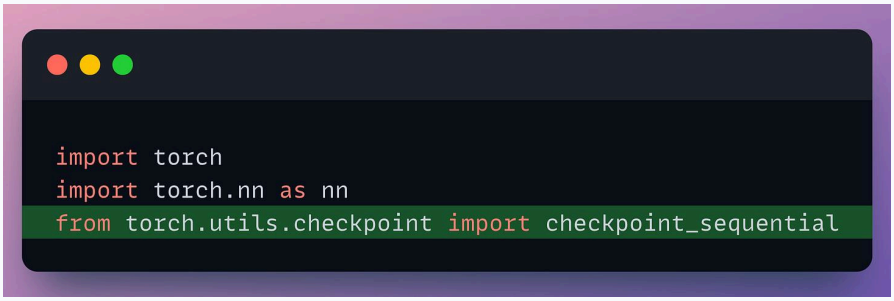
Nonetheless, gradient checkpointing is an extremely powerful technique to train larger models, which I have found to be pretty helpful at times, without resorting to more intensive techniques like distributed training, for instance.

Thankfully, gradient checkpointing is also implemented by many open-source deep learning frameworks like Pytorch, etc.

Here's a demo.

Gradient checkpointing in PyTorch

To utilize this, we begin by importing the necessary libraries and functions:



```
import torch
import torch.nn as nn
from torch.utils.checkpoint import checkpoint_sequential
```

Next, we define our neural network:

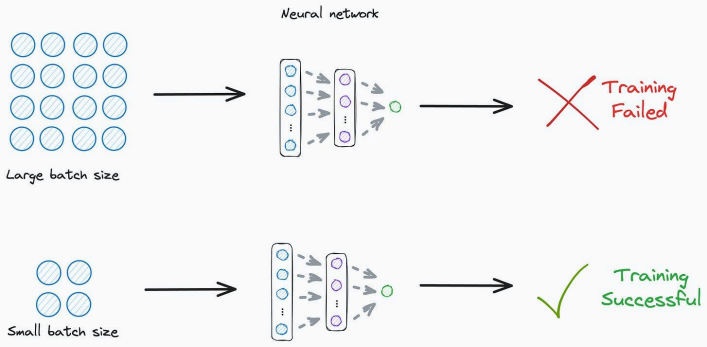

```
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.layer1 = nn.Linear(1000, 500)
        self.layer2 = nn.Linear(500, 200)
        self.layer3 = nn.Linear(200, 100)
        self.layer4 = nn.Linear(100, 10)

    def forward(self, x):
        # Define layers
        layers = torch.nn.Sequential(
            self.layer1, nn.ReLU(),
            self.layer2, nn.ReLU(),
            self.layer3, nn.ReLU(),
            self.layer4)

        # Checkpoint intermediate activations
        x = checkpoint_sequential(layers,
                                  segments=2,
                                  input=x)
        return x
```

As demonstrated above, in the forward method, we use the checkpoint_sequential method to use gradient checkpointing and divide the network into two segments.

Next, we can proceed with network training as we usually would.

Gradient Accumulation

Under memory constraints, it is always recommended to train the neural network with a small batch size.



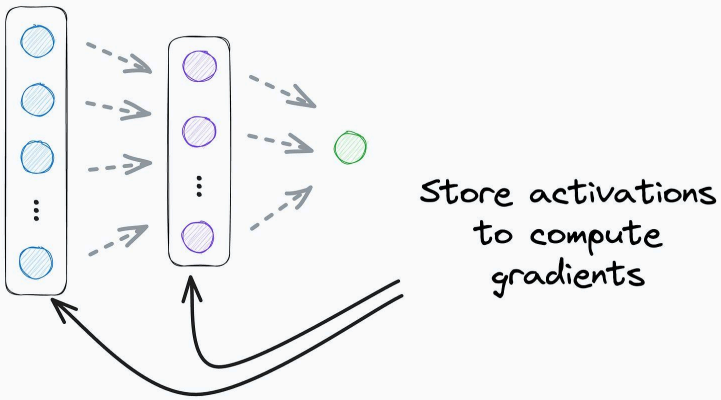
Despite that, there's a technique called **gradient accumulation**, which lets us (logically) increase batch size without explicitly increasing the batch size.

Confused?

Let's understand in this chapter. But before that, we must understand...

Why do neural networks typically explode during training?

The primary memory overhead in a neural network comes from backpropagation. This is because, during backpropagation, we must store the layer activations in memory. After all, they are used to compute the gradients.



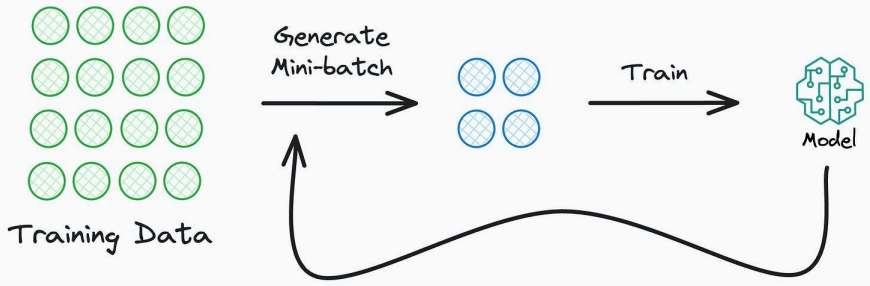
The bigger the network, the more activations a network must store in memory. Also, under memory constraints, having a large batch size will result in:

- storing many activations
- using those many activations to compute the gradients

This may lead to more resource consumption than available — resulting in training failure. But by reducing the batch size, we can limit the memory usage and train the network.

What is Gradient Accumulation and how does it help in increasing batch size in memory constraints?

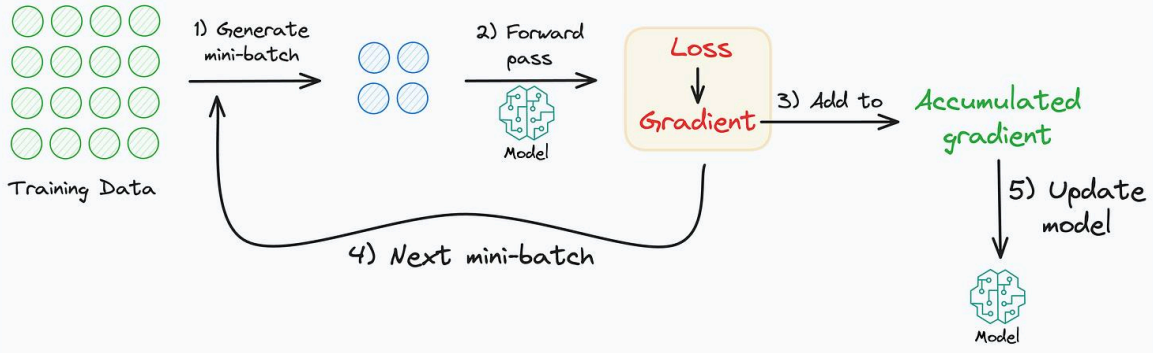
Consider we are training a neural network on mini-batches.



We train the network as follows:

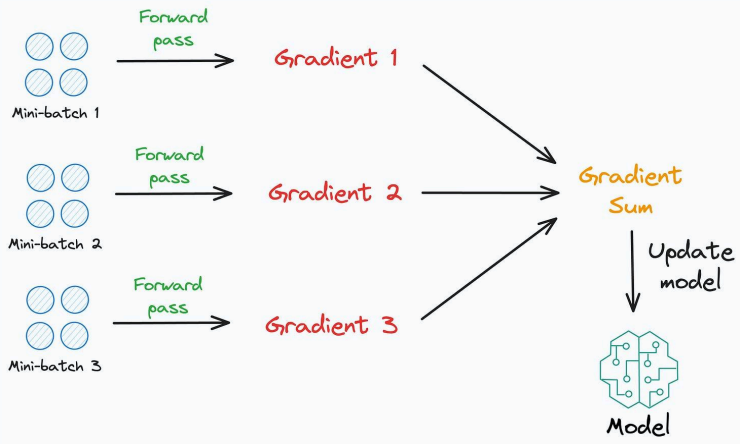
- On every mini-batch:
  - Run the forward pass while storing the activations.
  - During backward pass:
    - Compute the loss
    - Compute the gradients
    - Update the weights

Gradient accumulation modifies the last step of the backward pass, i.e., weight updates. More specifically, instead of updating the weights on every mini-batch, we can do this:



1. Run the forward pass on a mini-batch.
2. Compute the gradient values for weights in the network.
3. Don't update the weights yet.
4. Run the forward pass on the next mini-batch.
5. Compute the gradient values for weights and add them to the gradients obtained in step 2.
6. Repeat steps 3-5 for a few more mini-batches.
7. Update the weights only after processing a few mini-batches.

This technique works because accumulating the gradients across multiple mini-batches results in the same sum of gradients as if we were processing them together. Thus, logically speaking, using gradient accumulation, we can mimic a larger batch size without having to explicitly increase the batch size.



For instance, say we want to use a batch size of 64. However, current memory can only support a batch size of 16.

No worries!

- We can use a batch size of size 16.
- We can accumulate the gradients from every mini-batch.
- We can update the weights only once every 8 mini-batches.

Thus, effectively, we used a batch size of 16*8 (=128) instead of what we originally intended — 64.

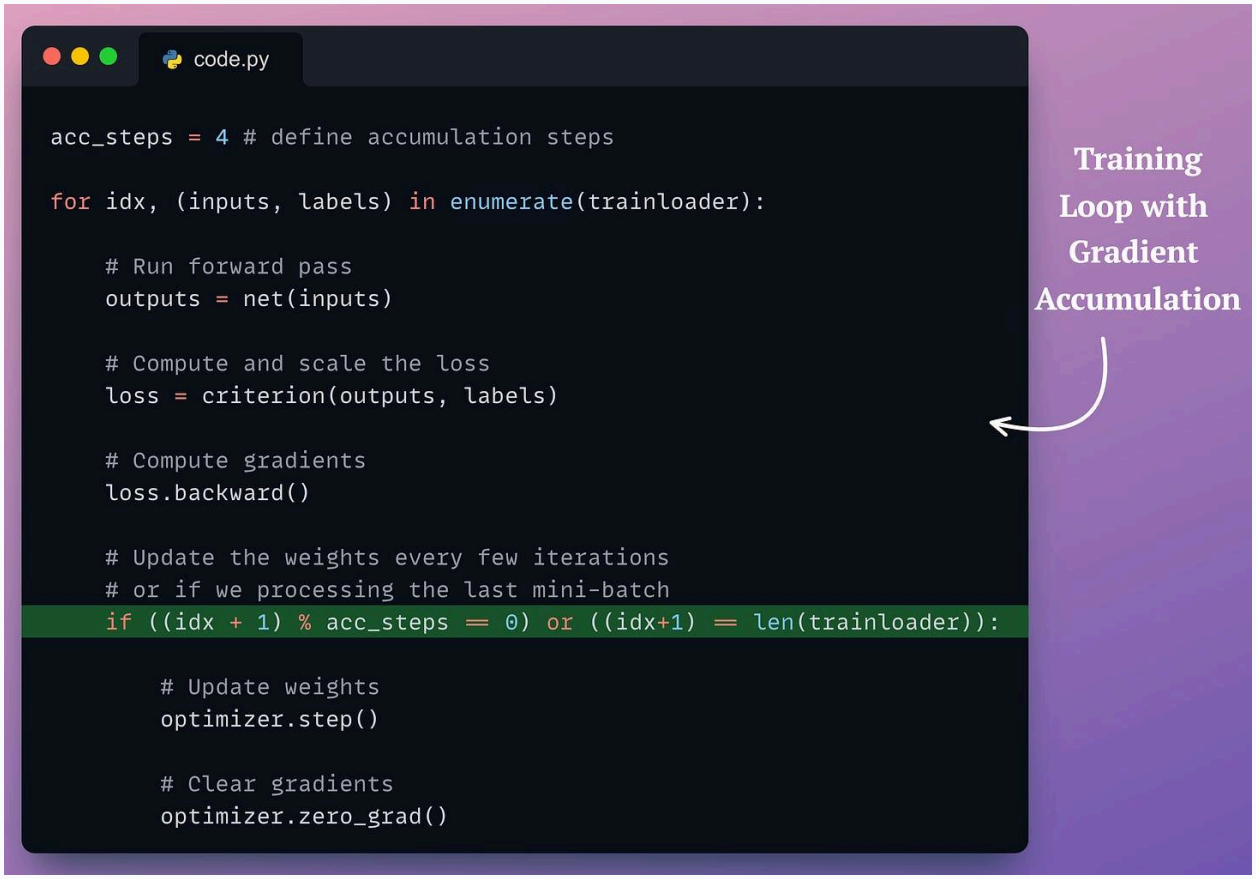
Implementation

Let's look at how we can implement this. In PyTorch, a typical training loop is implemented as follows:

- We clear the gradients
- Run the forward pass
- Compute the loss
- Compute the gradients
- Update the weights

However, as discussed earlier, if needed, we can only update the weights after a few iterations. Thus, we must continue to accumulate the gradients, which is precisely what `loss.backward()` does.

Also, as optimizer.zero_grad() clears the gradients, we must only execute it after updating the weights. This idea is implemented below:

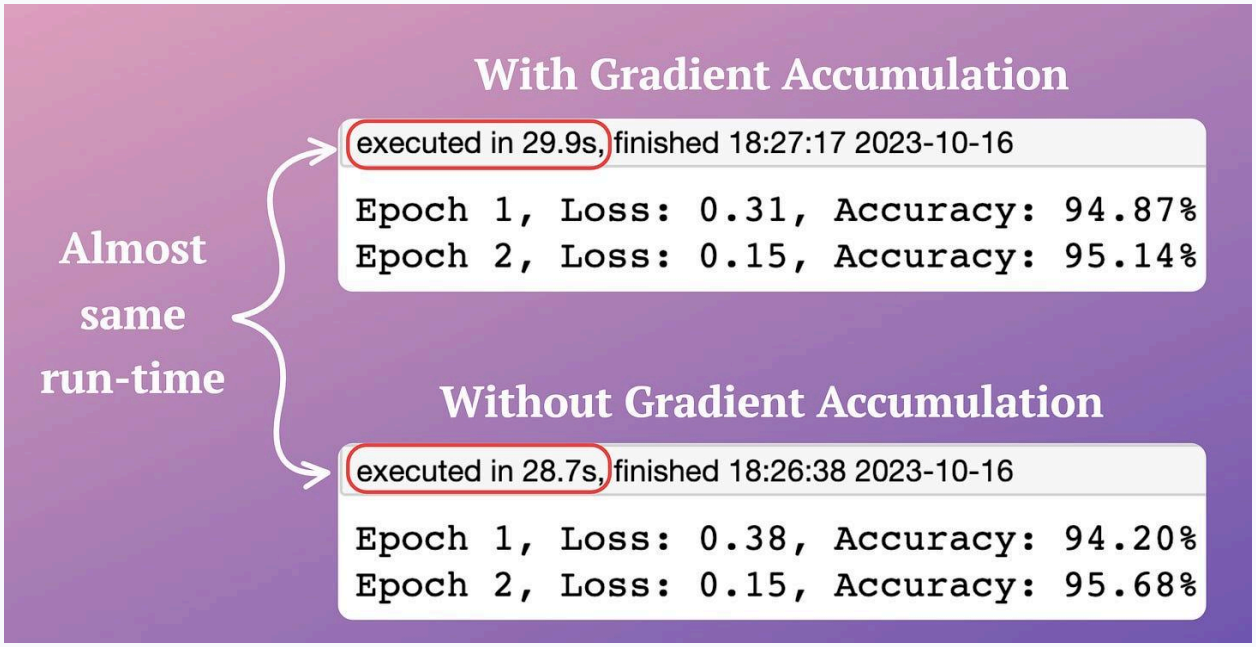


- First, we define acc_steps — the number of mini-batches after which we want to update the weights.
- Next, we run the forward pass.
- Moving on, we compute the loss and the gradients.
- As discussed earlier, we will not update the weights yet and instead let the gradients accumulate for a few more mini-batches.
- We only update the weights when the if condition is true.
- After updating, we clear the accumulated gradients.

This way, we can optimize neural network training in memory-constrained settings.

Departing note

Before we end, it is essential to note that gradient accumulation is NOT a remedy to improve run-time in memory-constrained situations. In fact, we can also verify this from my experiment:



Instead, its objective is to reduce overall memory usage.

Of course, it's true that we are updating the weights only after a few iterations. So, it will be a bit faster than updating on every iteration. Yet, we are still processing and computing gradients on small mini-batches, which is the core operation here.

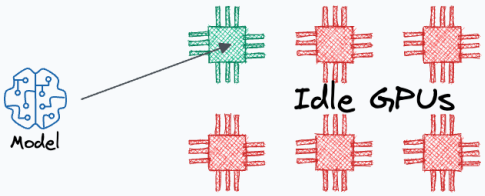
Nonetheless, the good thing is that even if you are not under memory constraints, you can still use gradient accumulation.

- Specify your typical batch size.
- Run forward pass.
- Compute loss and gradients.
- Update only after a few iterations.

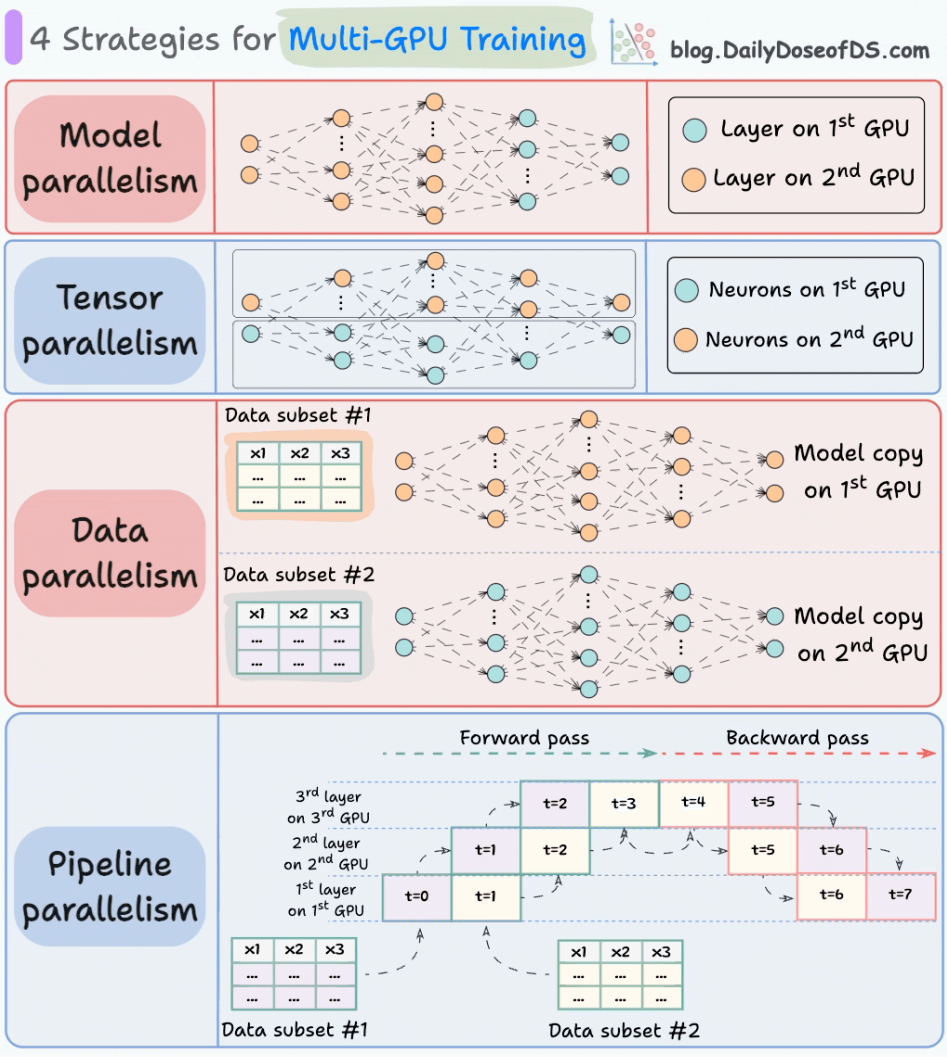
You can download the notebook here: https://bit.ly/3xNCfFt.

4 Strategies for Multi-GPU Training

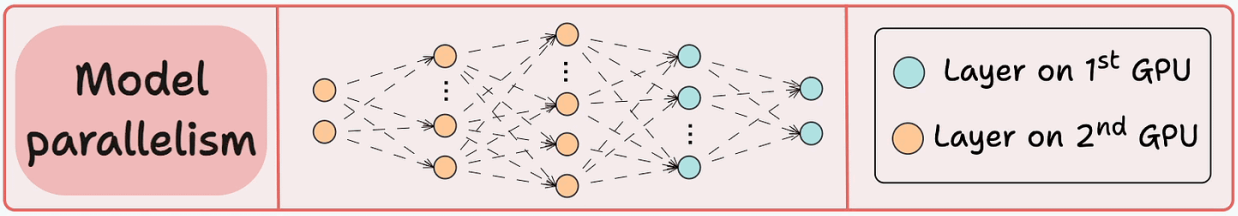
By default, deep learning models only utilize a single GPU for training, even if multiple GPUs are available.



An ideal way to proceed (especially in big-data settings) is to distribute the training workload across multiple GPUs. The graphic below depicts four common strategies for multi-GPU training:

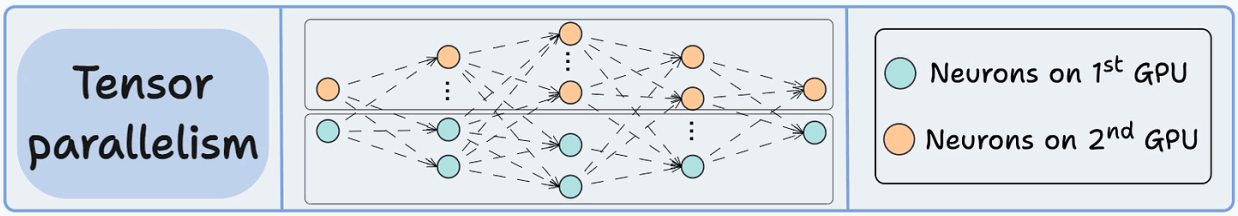


#1) Model parallelism

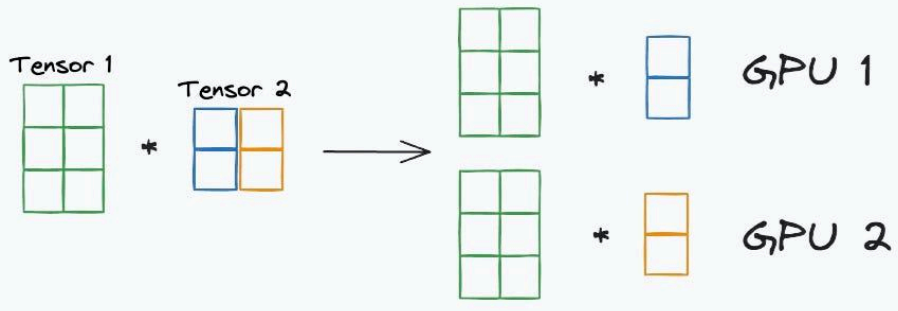


- Different parts (or layers) of the model are placed on different GPUs.
- Useful for huge models that do not fit on a single GPU.
- However, model parallelism also introduces severe bottlenecks as it requires data flow between GPUs when activations from one GPU are transferred to another GPU.

#2) Tensor parallelism

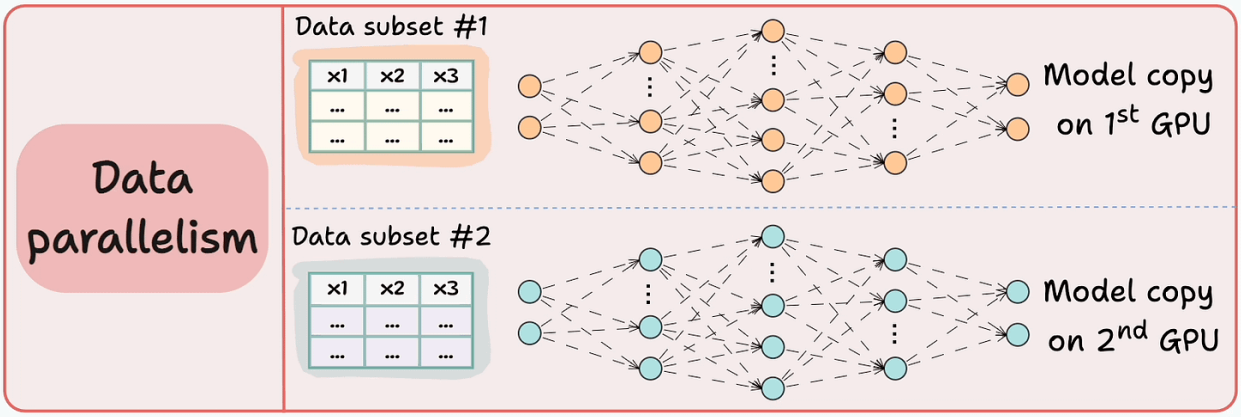


- Distributes and processes individual tensor operations across multiple devices or processors.
- It is based on the idea that a large tensor operation, such as matrix multiplication, can be divided into smaller tensor operations, and each smaller operation can be executed on a separate device or processor.



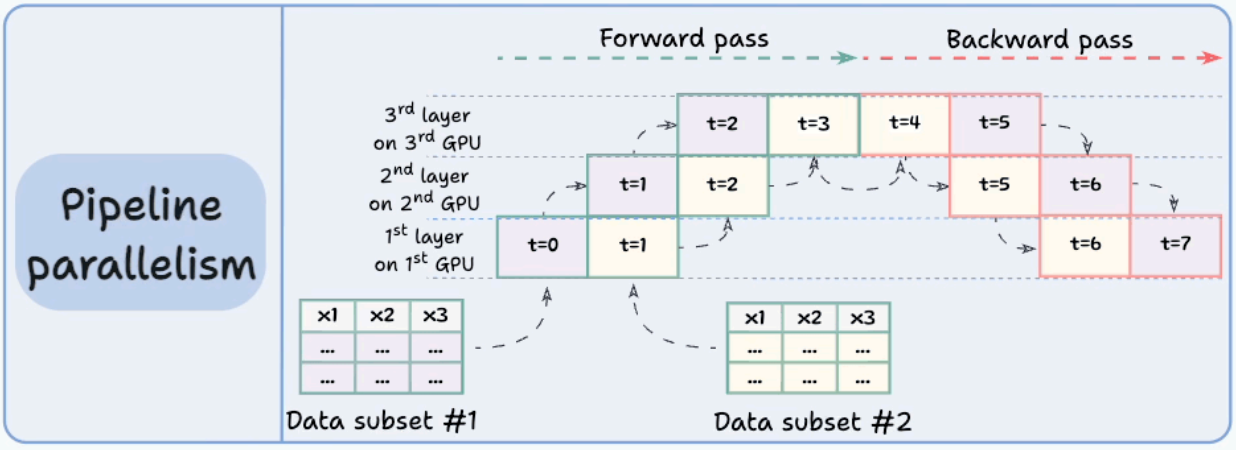
- Such parallelization strategies are inherently built into standard implementations of PyTorch and other deep learning frameworks, but they become much more pronounced in a distributed setting.

#3) Data parallelism



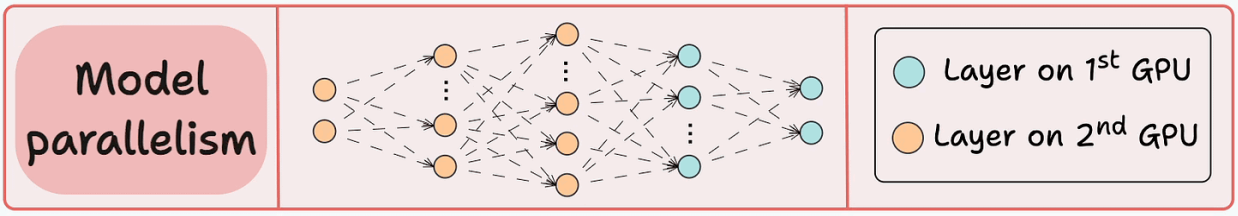
- Replicate the model across all GPUs.
- Divide the available data into smaller batches, and each batch is processed by a separate GPU.
- The updates (or gradients) from each GPU are then aggregated and used to update the model parameters on every GPU.

#4) Pipeline parallelism

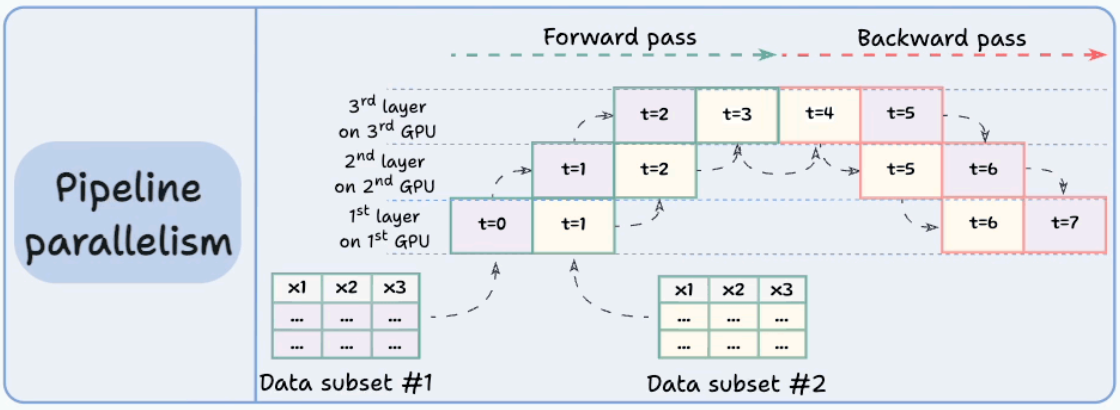


- This is often considered a combination of data parallelism and model parallelism.

- So the issue with standard model parallelism is that 1st GPU remains idle when data is being propagated through layers available in 2nd GPU:



- Pipeline parallelism addresses this by loading the next micro-batch of data once the 1st GPU has finished the computations on the 1st micro-batch and transferred activations to layers available in the 2nd GPU. The process looks like this:
    - 1st micro-batch passes through the layers on 1st GPU.
    - 2nd GPU receives activations on 1st micro-batch from 1st GPU.
    - While the 2nd GPU passes the data through the layers, another micro-batch is loaded on the 1st GPU.
    - And the process continues.
- GPU utilization drastically improves this way. This is evident from the animation below where multi-GPUs are being utilized at the same timestamp (look at t=1, t=2, t=5, and t=6):

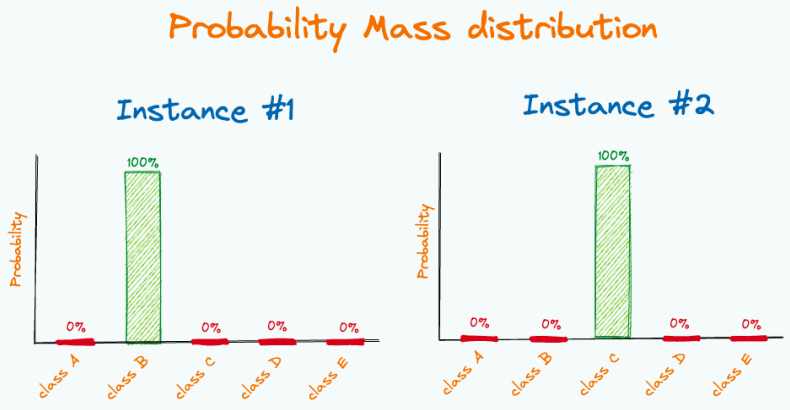


Those were four common strategies for multi-GPU training.

Miscellaneous

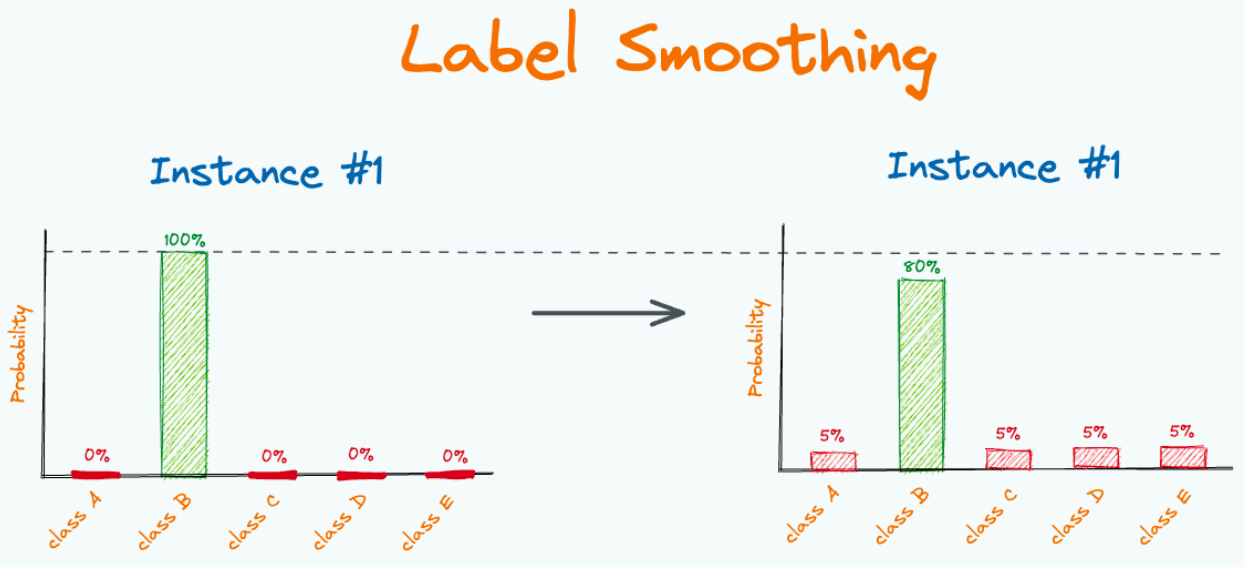
Label Smoothing

For every instance in single-label classification datasets, the entire probability mass belongs to a single class, and the rest are zero. This is depicted below:



The issue is that, at times, such label distributions excessively motivate the model to learn the true class for every sample with pretty high confidence. This can impact its generalization capabilities.

Label smoothing is a lesser-talked regularisation technique that elegantly addresses this issue.

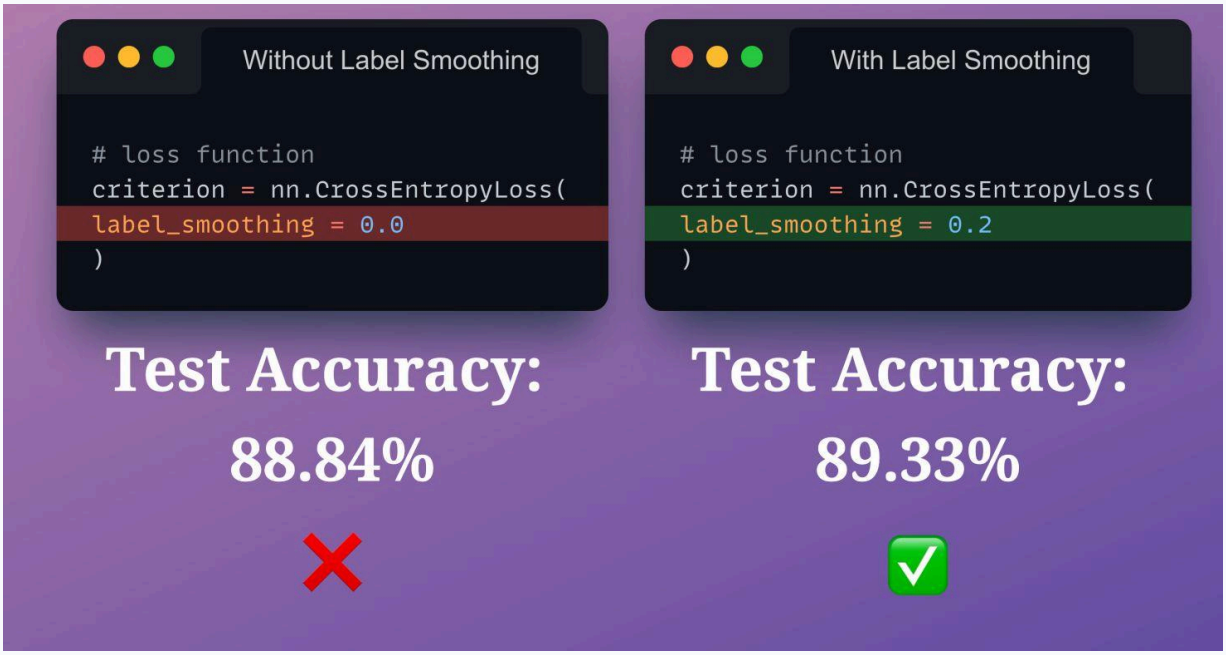


As depicted above, with label smoothing:

- We intentionally reduce the probability mass of the true class slightly.
- The reduced probability mass is uniformly distributed to all other classes.

Simply put, this can be thought of as asking the model to be "less overconfident" during training and prediction while still attempting to make accurate predictions.

The efficacy of this technique is evident from the image below:



In this experiment, I trained two neural networks on the Fashion MNIST dataset with the exact same weight initialization.

- One without label smoothing.
- Another with label smoothing.

The model with label smoothing resulted in a better test accuracy, i.e., better generalization.

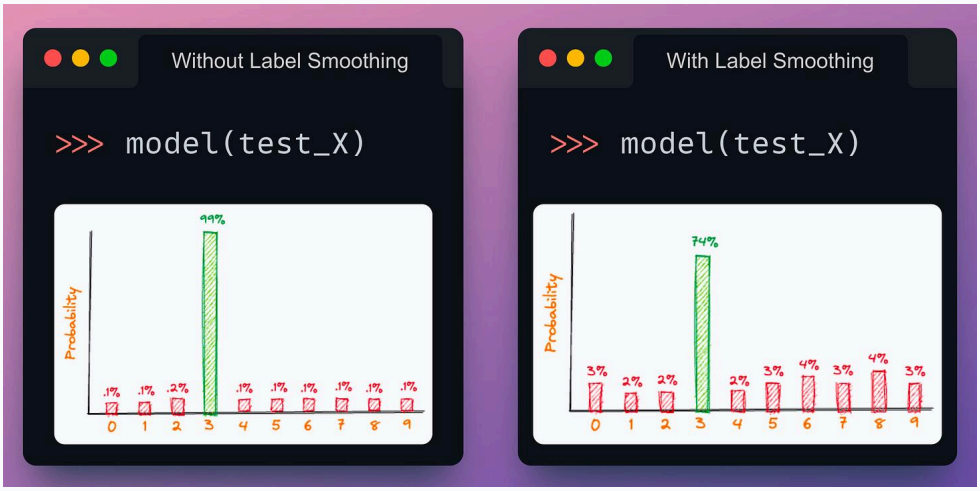
When not to use label smoothing?

After using label smoothing for many of my projects, I have also realized that it is not well suited for all use cases. So it's important to know when you should not use it.

See, if you only care about getting the final prediction correct and improving generalization, label smoothing will be a pretty handy technique. However, I wouldn't recommend utilizing it if you care about:

- Getting the prediction correct.
- **And** understanding the model's confidence in generating a prediction.

This is because as we discussed above, label smoothing guides the model to become "less overconfident" about its predictions. Thus, we typically notice a drop in the confidence values for every prediction, as depicted below:



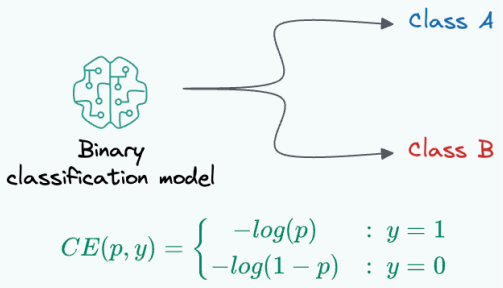
On a specific test instance:

- The model without label smoothing outputs 99% probability for class 3.
- With label smoothing, although the prediction is still correct, the confidence drops to 74%.

This is something to keep in mind when using label smoothing. Nonetheless, the technique is indeed pretty promising for regularizing deep learning models. You can download the code notebook for this chapter here: https://bit.ly/4ePt08d.

Focal loss

Binary classification tasks are typically trained using the binary cross entropy (BCE) loss function:



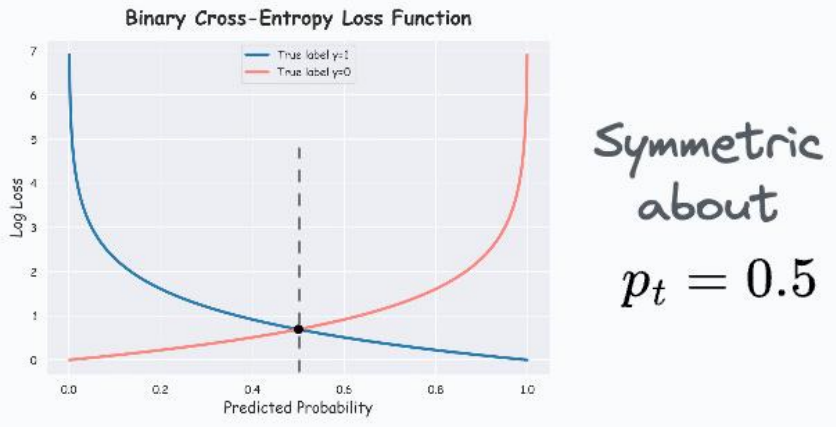
For notational convenience, if we define p_t as the following:

$$p_t = \begin{cases} p & : y = 1 \\ 1 - p & : y = 0 \end{cases}$$

...then we can also write the cross-entropy loss function as:

$$CE(p, y) = -log(p_t)$$

That said, one limitation of BCE loss is that it weighs probability predictions for both classes equally, which is evident from its symmetry:



For more clarity, consider the table below, which depicts two instances, one from the minority class and another from the majority class, both with the same loss:

True label	$y = 1$ minority class	$y = 0$ majority class
Probability output	$\hat{y} = 0.3$	$\hat{y} = 0.7$
log loss	$-log(0.3)$	$-log(0.3)$

This causes problems when we use BCE for imbalanced datasets, wherein most instances from the dominating class are "easily classifiable." Thus, a loss value of, say, -log(0.3) from the majority class instance should (ideally) be weighed LESS than the same loss value from the minority class.

$-log(0.3)$ from minority class $>>$ $-log(0.3)$ from majority class

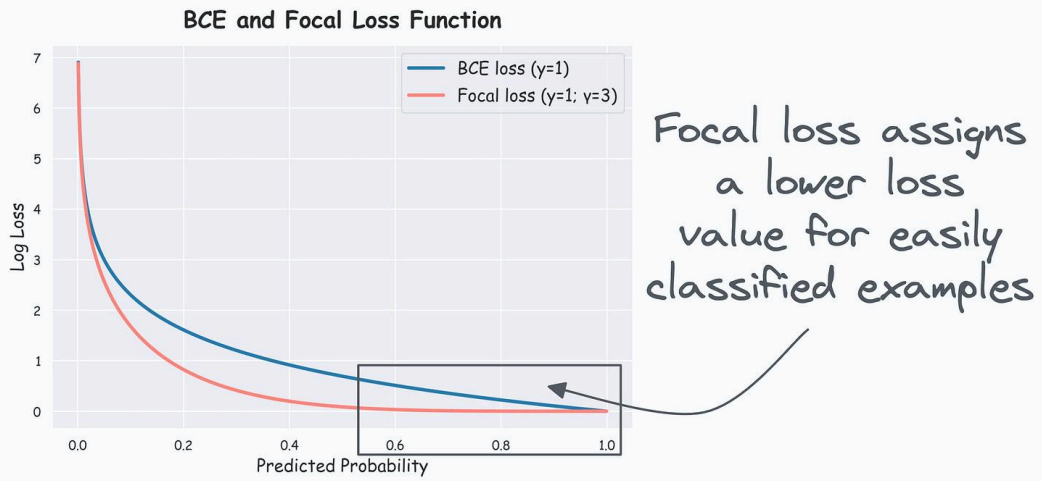
Focal loss is a pretty handy and useful alternative to address this issue. It is defined as follows:

$$CE(p, y) = -log(p_t)$$

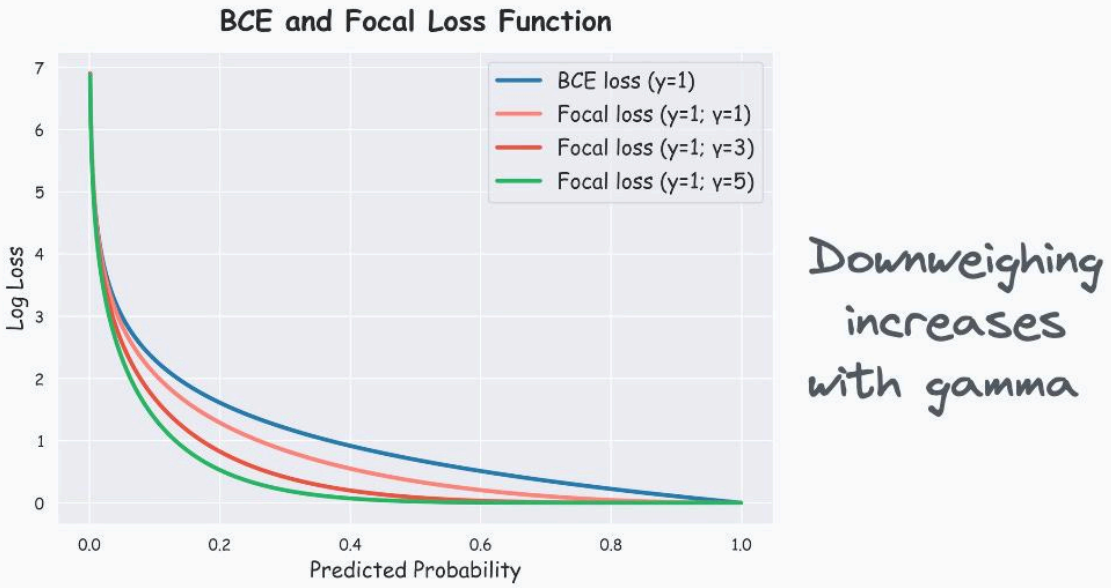
$$Focal(p, y) = -(1 - p_t)^{\gamma} \cdot log(p_t)$$

As depicted above, it introduces an additional multiplicative factor called downweighing, and the parameter γ (Gamma) is a hyperparameter.

Plotting BCE (class y=1) and Focal loss (for class y=1 and γ=3), we get the following curve:

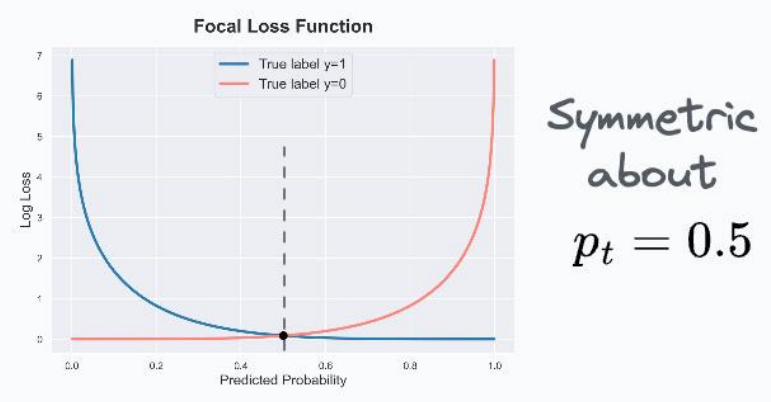


As shown in the figure above, focal loss reduces the contribution of the predictions the model is pretty confident about. Also, the higher the value of γ (Gamma), the more downweighing takes place, as shown in this plot below:

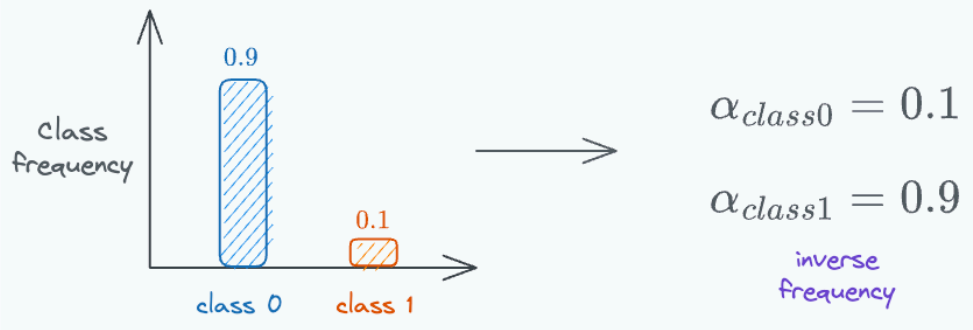


Moving on, while the Focal loss function reduces the contribution of confident predictions, we aren't done yet.

The focal loss function now is still symmetric like BCE:



To address this, we must add another weighing parameter (α), which is the inverse of the class frequency, as depicted below:



The α parameter is the inverse of the class frequency Thus, the final loss function comes out to be the following:

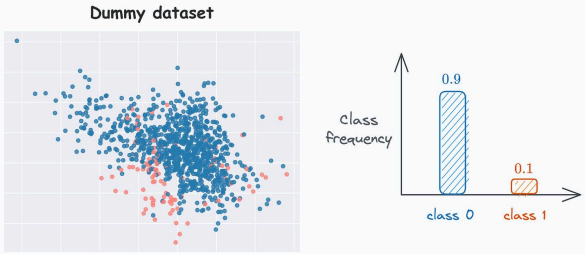
$$Focal(p, y) = -\boxed{\alpha_t} \cdot \boxed{(1 - p_t)^{\gamma}} \cdot log(p_t)$$

class inverse weighing

Probability downweighing

By using both downweighing and inverse weighing, the model gradually learns patterns specific to the hard examples instead of always being overly confident in predicting easy instances.

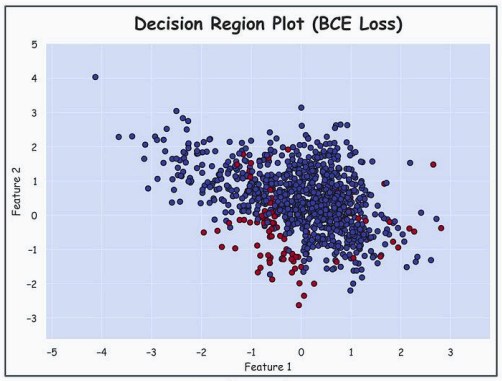
To test the efficacy of focal loss in a class imbalance setting, I created a dummy classification dataset with a 90:10 imbalance ratio:



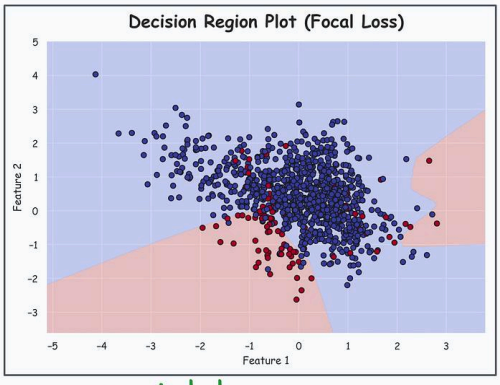
Next, I trained two neural network models (with the same architecture of 2 hidden layers):

- One with BCE loss
- Another with Focal loss

The decision region plot and test accuracy for these two models is depicted below:



Model always predicts majority class (Test accuracy = 90%)



Model learns some minority class patterns (Test accuracy = 93%)

It is clear that:

- The model trained with BCE loss (left) always predicts the majority class.
- The model trained with focal loss (right) focuses relatively more on minority class patterns. As a result, it performs better.

Download this Jupyter notebook to get started with Focal loss: https://bit.ly/45XzNZC.

How Dropout Actually Works?

Some time back, I was invited by a tech startup to conduct their ML interviews. I interviewed 12 candidates and mostly asked practical ML questions.

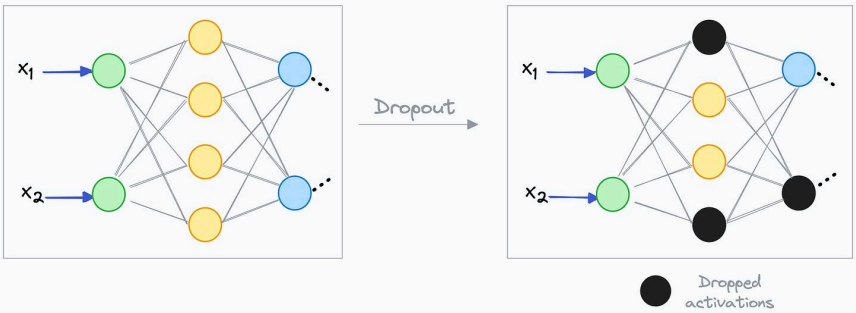
However, there were some conceptual questions as well, like the one below, which I intentionally asked every candidate:

How does Dropout work?

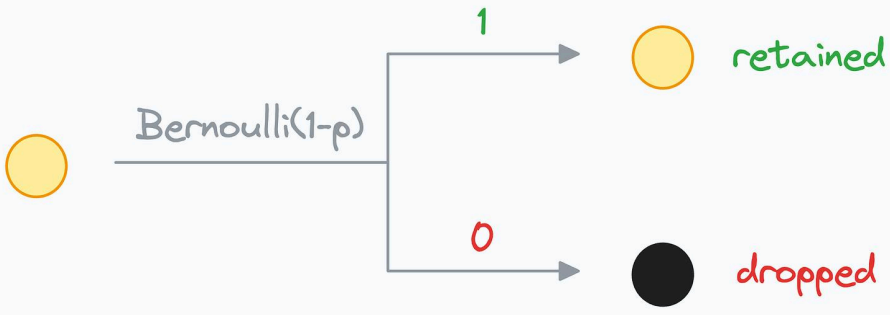
Pretty simple, right? Apparently, every candidate gave me an incomplete answer, which I have mentioned below:

Candidates' Answer

In a gist, the idea is to zero out neurons randomly in a neural network. This is done to regularize the network.



Dropout is only applied during training, and which neuron activations to zero out (or drop) is decided using a Bernoulli distribution:



"p" is the dropout probability specified in, say, PyTorch → nn.Dropout(p).

My follow-up question: Is there anything else that we do in Dropout?

Candidates: No, that's it. We only zero out neurons and train the network as we usually would.

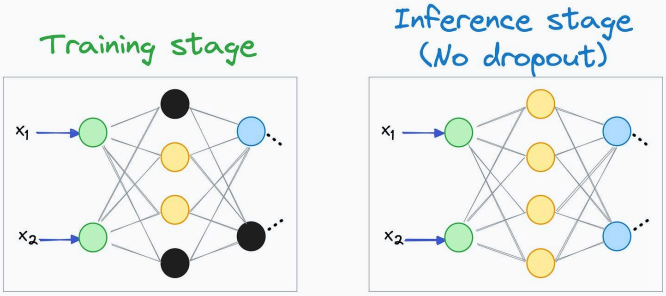
Now, coming back to the topic...

Of course, I am not saying that the above details are incorrect. They are correct.

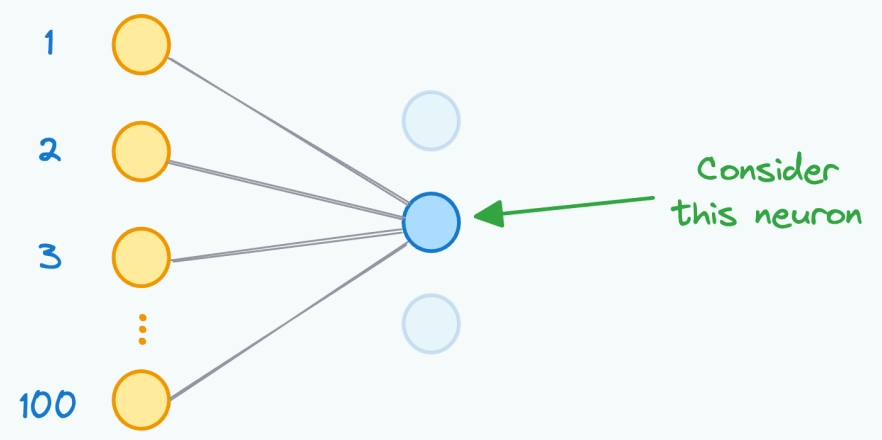
However, this is just 50% of how Dropout works, and disappointingly, most resources don't cover the remaining 50%. If you too are only aware of the 50% details I mentioned above, continue reading.

How Dropout actually works?

To begin, we must note that Dropout is only applied during training, but not during the inference/evaluation stage:

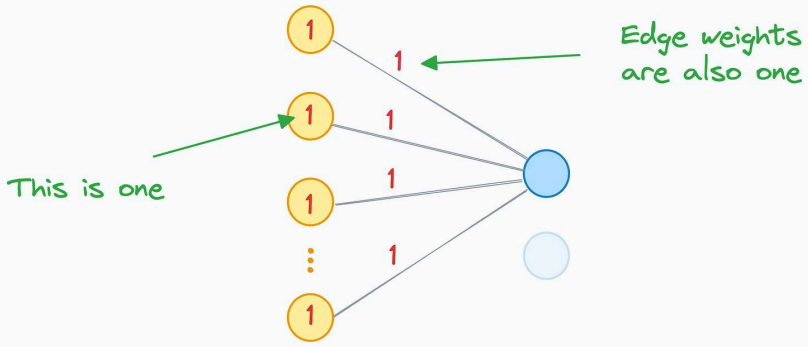


Now, consider that a neuron's input is computed using 100 neurons in the previous hidden layer:

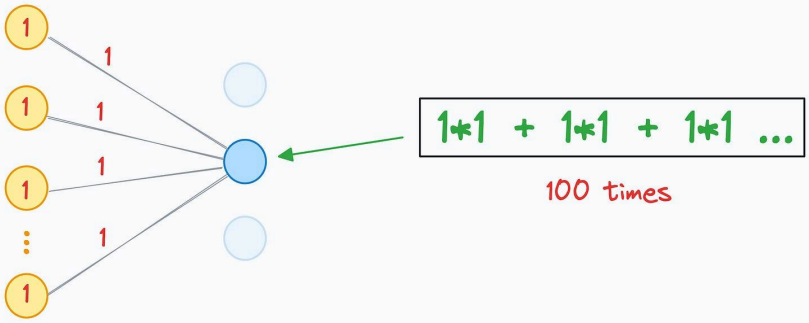


For simplicity, let's assume a couple of things here:

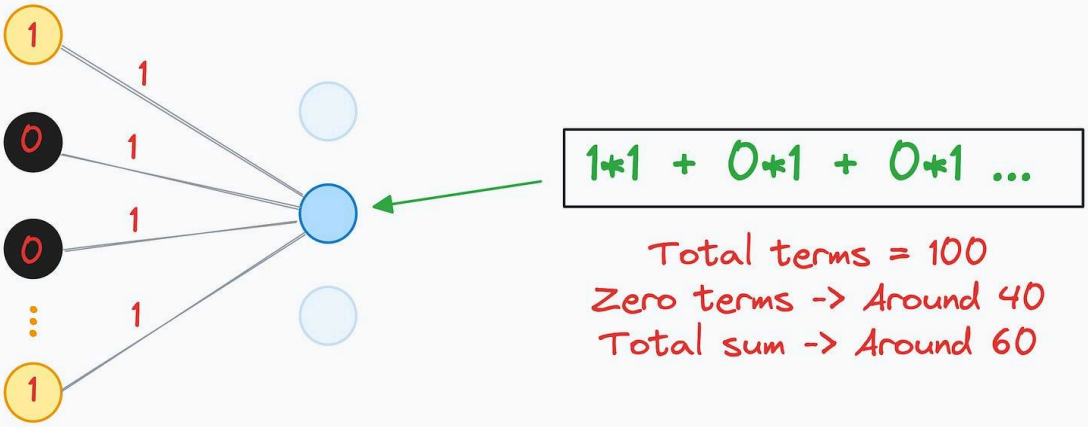
- The activation of every yellow neuron is 1.
- The edge weight from the yellow neurons to the blue neuron is also 1.



As a result, the input received by the blue neuron will be 100, as depicted below:



Now, during training, if we were using Dropout with, say, a 40% dropout rate, then roughly 40% of the yellow neuron activations would have been zeroed out. As a result, the input received by the blue neuron would have been around 60:



However, the above point is only valid for the training stage.

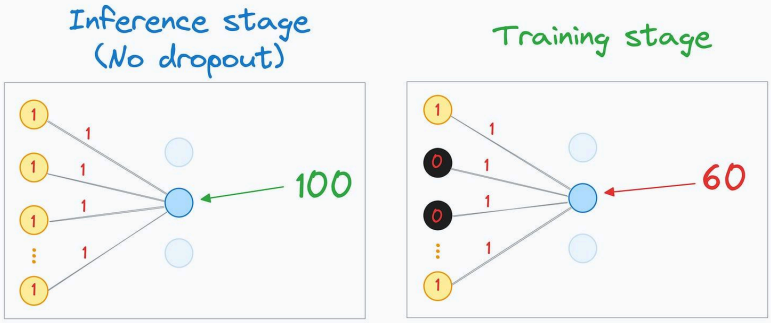
If the same scenario had existed during the inference stage instead, then the input received by the blue neuron would have been 100.

Thus, under similar conditions:

- The input received during training → 60.
- The input received during inference → 100.

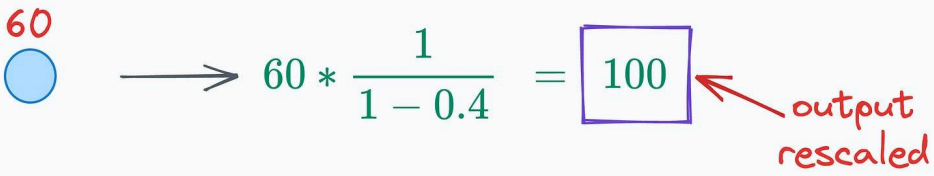
Do you see any problem here?

During training, the average neuron inputs are significantly lower than those received during inference.



More formally, using Dropout significantly affects the scale of the activations. However, it is desired that the neurons throughout the model must receive the roughly same mean (or expected value) of activations during training and inference. To address this, Dropout performs one additional step.

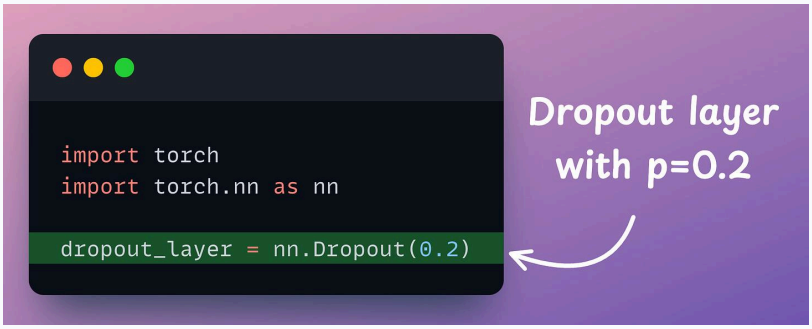
This idea is to scale the remaining active inputs during training. The simplest way to do this is by scaling all activations during training by a factor of 1/(1-p), where p is the dropout rate. For instance, using this technique on the neuron input of 60, we get the following (recall that we set p=40%):



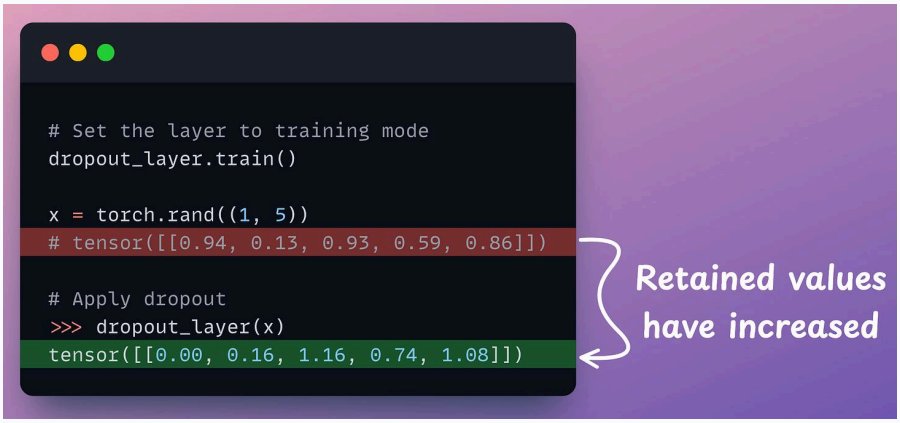
As depicted above, scaling the neuron input brings it to the desired range, which makes training and inference stages coherent for the network.

Verifying experimentally

In fact, we can verify that typical implementations of Dropout, from PyTorch, for instance, do carry out this step. Let's define a dropout layer as follows:



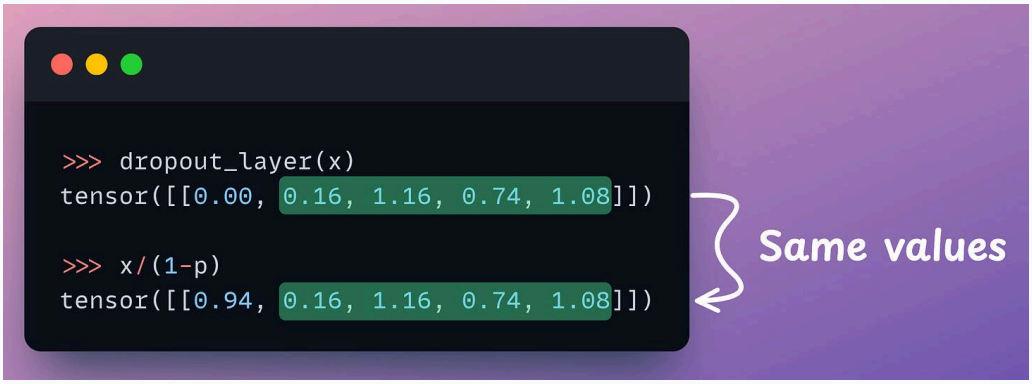
Now, let's consider a random tensor and apply this dropout layer to it:



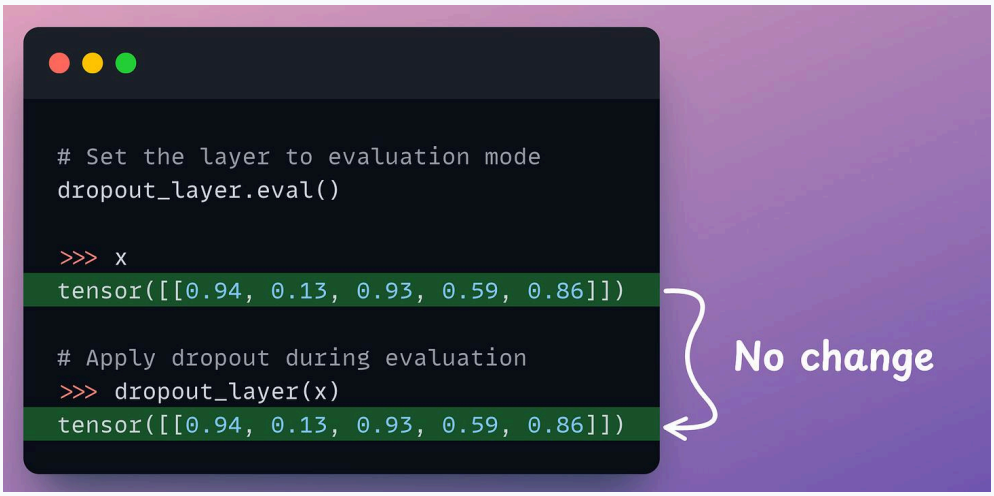
As depicted above, the retained values have increased.

- The second value goes from 0.13 → 0.16.
- The third value goes from 0.93 → 1.16.
- and so on...

What's more, the retained values are precisely the same as we would have obtained by explicitly scaling the input tensor with 1/(1-p):



If we were to do the same thing in evaluation mode instead, we notice that no value is dropped and no scaling takes place either, which makes sense as Dropout is only used during training:



This is the remaining 50% details, which, in my experience, most resources do not cover, and as a result, most people aren't aware of.

But it is a highly important step in Dropout, which maintains numerical coherence between training and inference stages.

With that, now you know 100% of how Dropout works.

Next, let's discuss an issue with Dropout in case of CNNs.

Issues with Dropout in CNNs

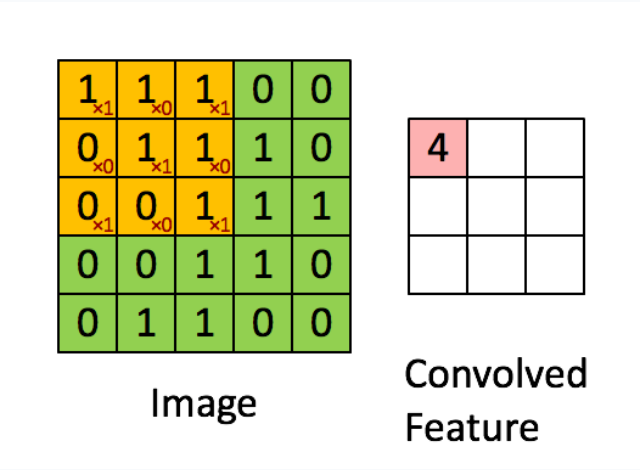
When it comes to training neural networks, it is always recommended to use Dropout to improve its generalization power.

This applies not just to CNNs but to all other neural networks. And I am sure you already know the above details, so let's get into the interesting part.

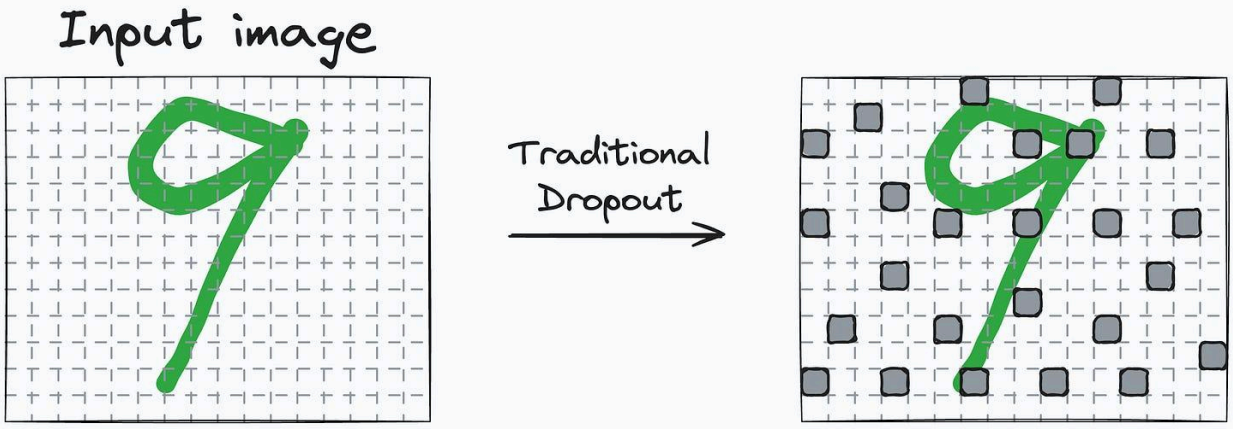
The problem of using Dropout in CNNs

The core operation that makes CNNs so powerful is convolution, which allows them to capture local patterns, such as edges and textures, and helps extract relevant information from the input.

From a purely mathematical perspective, we slide a filter (shown in yellow below) over the input (shown in green below) and take the element-wise sum between the filter and the overlapped input to get the convolution output:



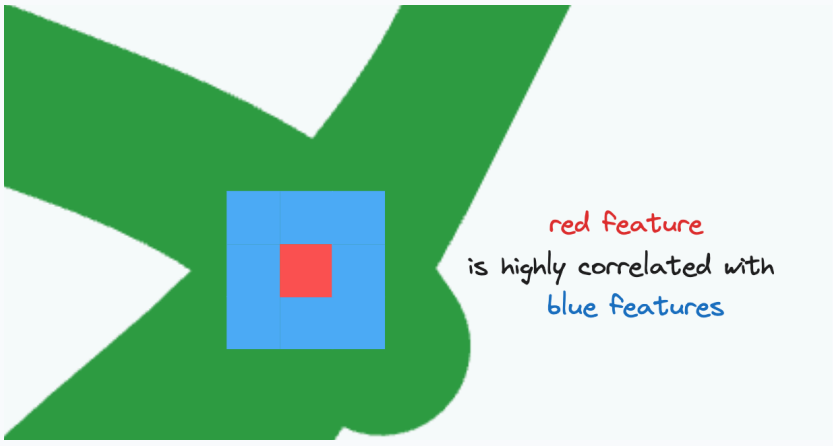
Here, if were to apply the traditional Dropout, the input features would look something like this:



In fully connected layers, we zero out neurons. In CNNs, however, we randomly zero out the pixel values before convolution, as depicted above.

But this isn't found to be that effective specifically for convolution layers. To understand this, consider we have some image data. In every image, we would find that nearby features (or pixels) are highly correlated spatially.

For instance, imagine zooming in on the pixel level of the digit '9'. Here, we would notice that the red pixel (or feature) is highly correlated with other features in its vicinity:

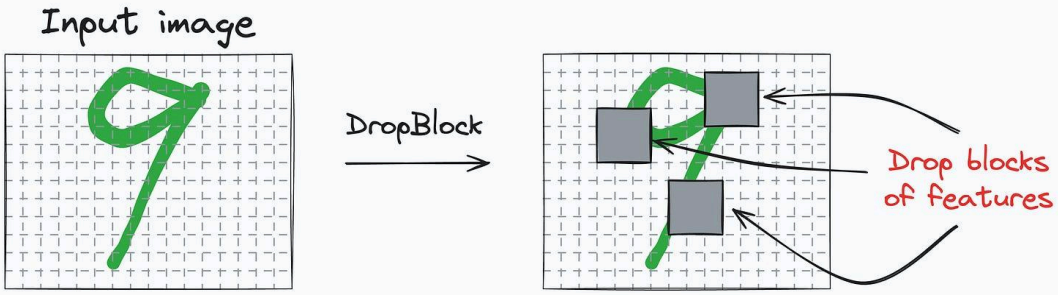


Thus, dropping the red feature using Dropout will likely have no effect and its information can still be sent to the next layer.

Simply put, the nature of the convolution operation defeats the entire purpose of the traditional Dropout procedure.

The solution

DropBlock is a much better, effective, and intuitive way to regularize CNNs. The core idea in DropBlock is to drop a contiguous region of features (or pixels) rather than individual pixels. This is depicted below:

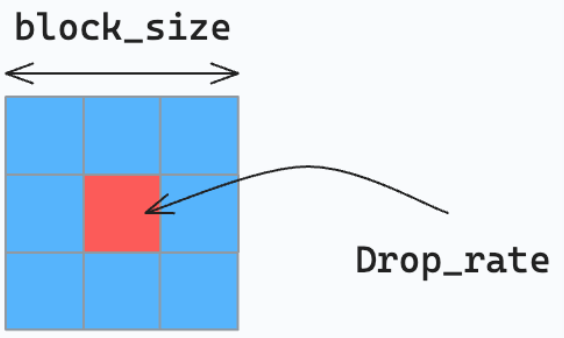


Similar to Dropout in fully connected layers, wherein the network tries to generate more robust ways to fit the data in the absence of some activations, in the case of DropBlock, the convolution layers get more robust to fit the data despite the absence of a block of features.

Moreover, the idea of DropBlock also makes intuitive sense — if a contiguous region of a feature is dropped, the problem of using Dropout with convolution operation can be avoided.

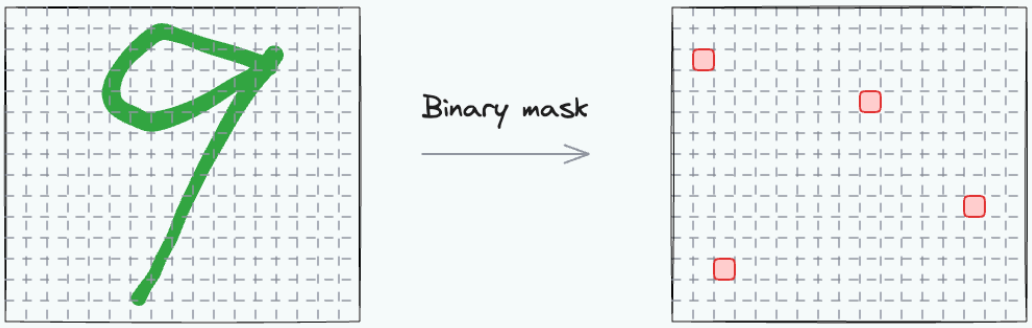
DropBlock parameters

DropBlock has two main parameters:

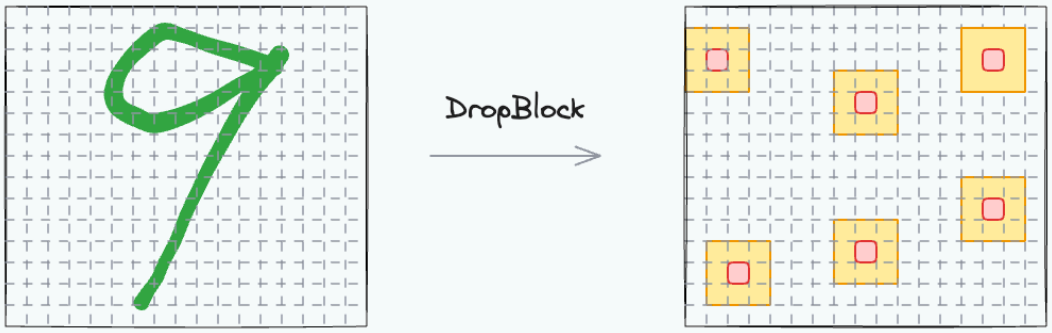


- `Block_size`: The size of the box to be dropped.
- `Drop_rate`: The drop probability of the central pixel.

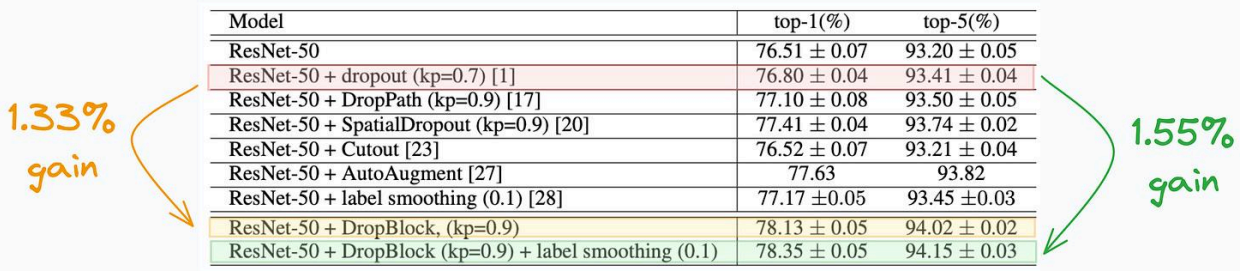
To apply DropBlock, first, we create a binary mask on the input sampled from the Bernoulli distribution:



Next, we create a block of size `block_size*block_size` which has the sampled pixels at the center:



The efficacy of DropBlock over Dropout is evident from the results table below:



Model	top-1(%)	top-5(%)
ResNet-50	76.51 ± 0.07	93.20 ± 0.05
ResNet-50 + dropout (kp=0.7) [1]	76.80 ± 0.04	93.41 ± 0.04
ResNet-50 + DropPath (kp=0.9) [17]	77.10 ± 0.08	93.50 ± 0.05
ResNet-50 + SpatialDropout (kp=0.9) [20]	77.41 ± 0.04	93.74 ± 0.02
ResNet-50 + Cutout [23]	76.52 ± 0.07	93.21 ± 0.04
ResNet-50 + AutoAugment [27]	77.63	93.82
ResNet-50 + label smoothing (0.1) [28]	77.17 ±0.05	93.45 ±0.03
ResNet-50 + DropBlock, (kp=0.9)	78.13 ± 0.05	94.02 ± 0.02
ResNet-50 + DropBlock (kp=0.9) + label smoothing (0.1)	78.35 ± 0.05	94.15 ± 0.03

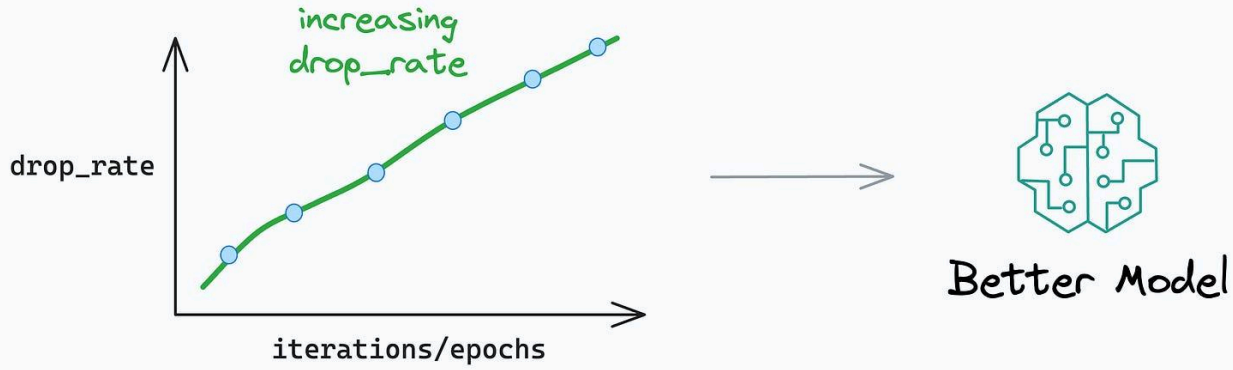
On the ImageNet classification dataset:

- DropBlock provides a 1.33% gain over Dropout.
- DropBlock with Label smoothing (discussed in the last chapter) provides a 1.55% gain over Dropout.

Thankfully, DropBlock is also integrated with PyTorch.

There's also a library for DropBlock, called "dropblock," which also provides the linear scheduler for drop_rate.

So the thing is that the researchers who proposed DropBlock found the technique to be more effective when the drop_rate was increased gradually.



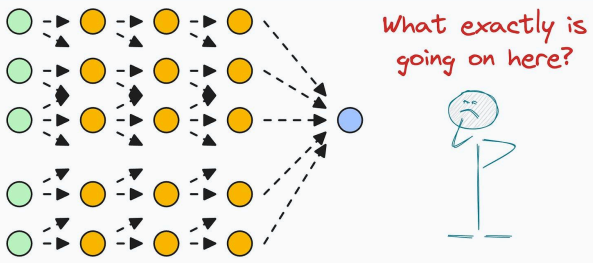
The DropBlock library implements the scheduler. But of course, there are ways to do this in PyTorch as well. So it's entirely up to you which implementation you want to use:

- DropBlock PyTorch: https://bit.ly/3xZfT3E.
- DropBlock library: https://github.com/miguelvr/dropblock.

What Hidden Layers and Activation Functions Actually Do

Everyone knows the objective of an activation function in a neural network. They let the network learn non-linear patterns. There is nothing new here, and I am sure you are aware of that too.

However, one thing I have often realized is that most people struggle to build an intuitive understanding of what exactly a neural network consistently tries to achieve during its layer-after-layer transformations.

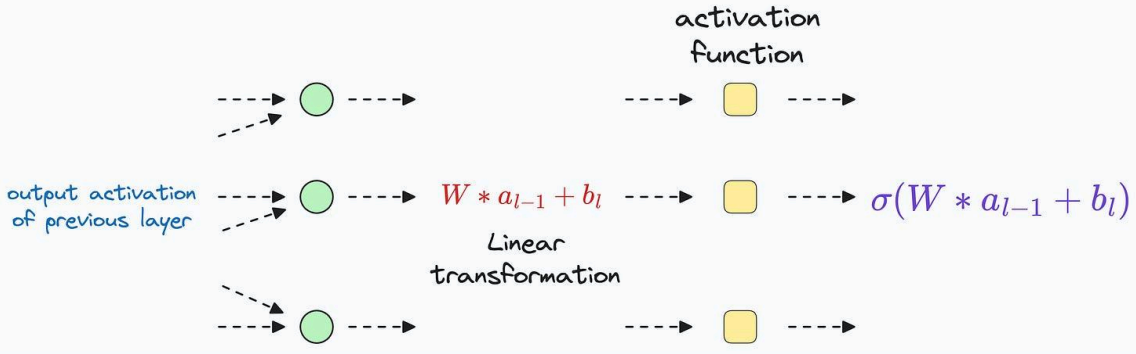


In this chapter, let me share a unique perspective on this, which would really help you understand the internal workings of a neural network.

I have supported this chapter with plenty of visuals for better understanding. Also, for simplicity, we shall consider a binary classification use case.

Background

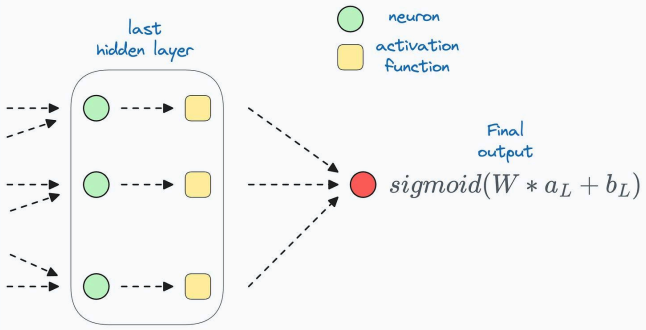
The data undergoes a series of transformations at each hidden layer:



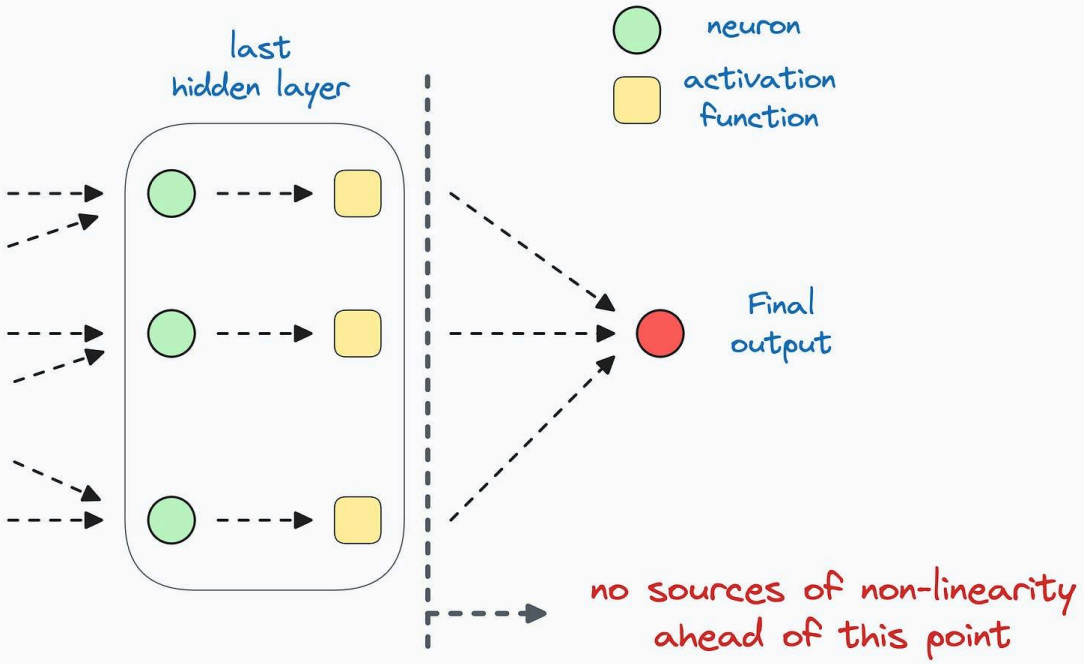
- Linear transformation of the data obtained from the previous layer
- ...followed by a non-linearity using an activation function — ReLU, Sigmoid, Tanh, etc.

The above transformations are performed on every hidden layer of a neural network. Now, notice something here.

Assume that we just applied the above data transformation on the **very last hidden layer** of the neural network. Once we do that, the activations progress toward the output layer of the network for one final transformation, which is entirely linear.



The above transformation is entirely linear because all sources of non-linearity (activations functions) exist on or before the last hidden layer. And during the forward pass, once the data leaves the last hidden layer, there is no further scope for non-linearity.



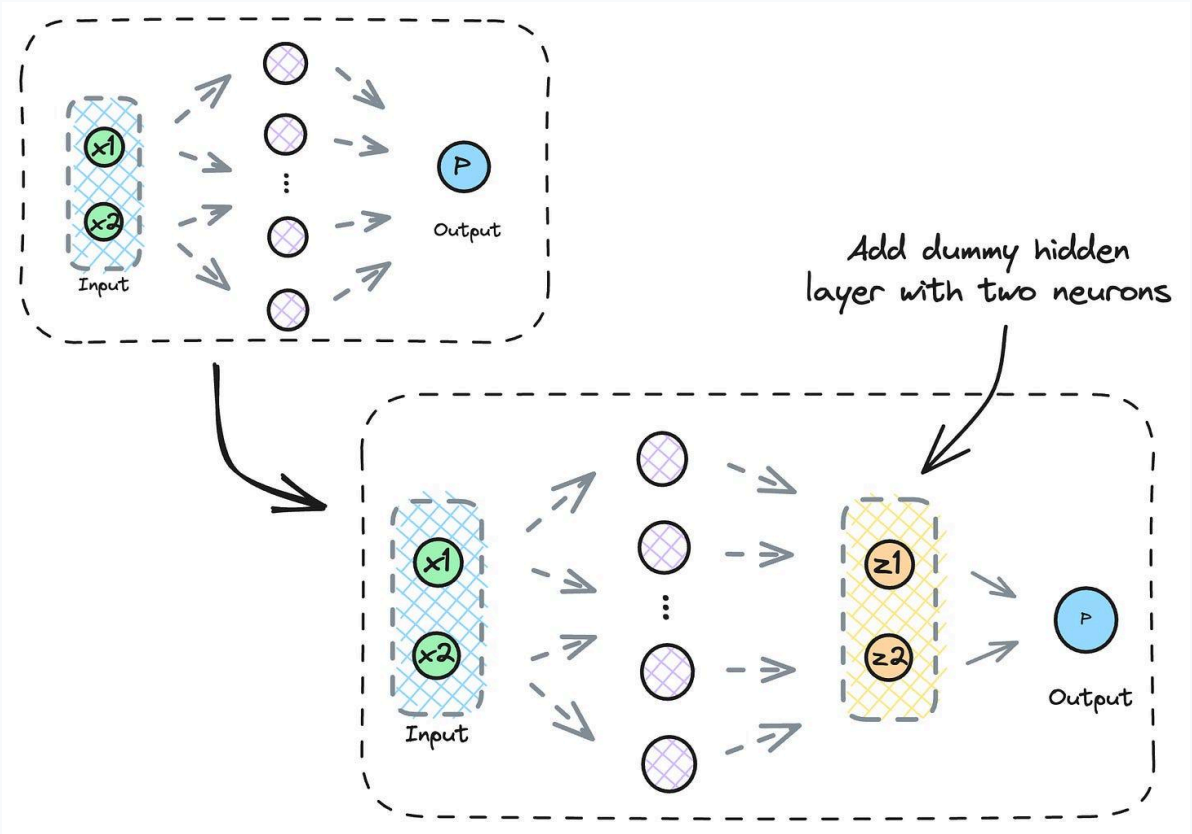
Thus, to make accurate predictions, the data received by the output layer from the last hidden layer MUST BE linearly separable.

To summarize....

While transforming the data through all its hidden layers and just before reaching the output layer, a neural network is constantly hustling to project the data to a space where it somehow becomes linearly separable. If it does, the output layer becomes analogous to a logistic regression model, which can easily handle this linearly separable data.

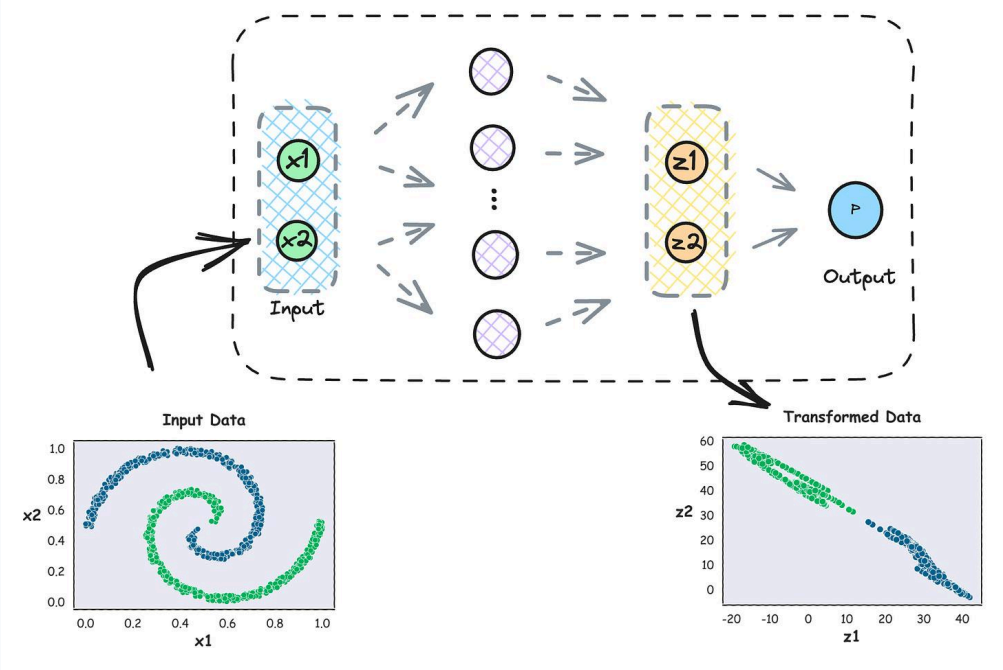
In fact, we can also verify this experimentally.

To visualize the input transformation, we can add a dummy hidden layer with just two neurons right before the output layer and train the neural network again.



Why do we add a layer with just two neurons?

This way, we can easily visualize the transformation. We expect that if we plot the activations of this 2D dummy hidden layer, they must be linearly separable. The below visual precisely depicts this.



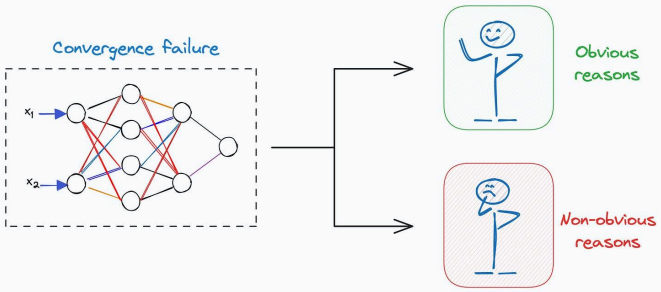
As we notice above, while the input data was linearly inseparable, the input received by the output layer is indeed linearly separable.

This transformed data can be easily handled by the output classification layer.

And this shows that all a neural network is trying to do is transform the data into a linearly separable form before reaching the output layer.

Shuffle Data Before Training

Deep learning models may fail to converge due to various reasons. Some causes are obvious and common, and therefore, quickly rectifiable, like too high/low learning rate, no data normalization, no batch normalization, etc.

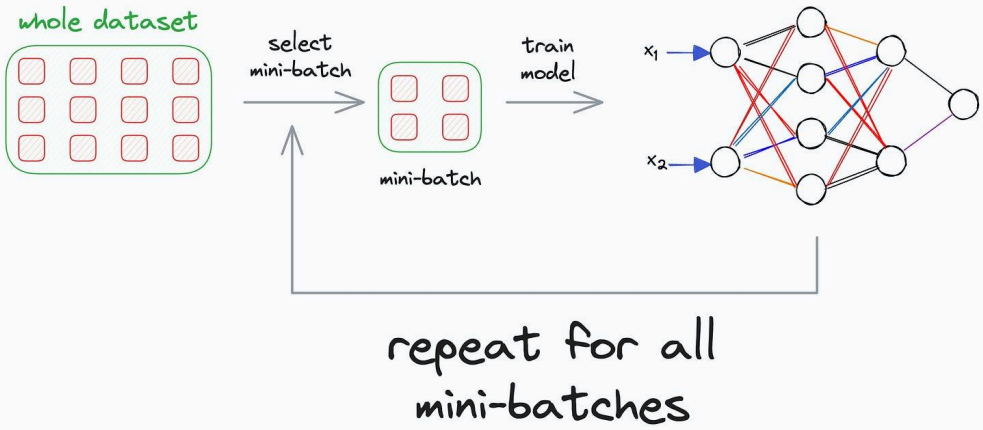


But the problem arises when the cause isn't that apparent. Therefore, it may take some serious time to debug if you are unaware of them. In this chapter, I want to talk about one such data-related mistake, which I once committed during my early days in machine learning. Admittedly, it took me quite some time to figure it out back then because I had no idea about the issue.

Experiment

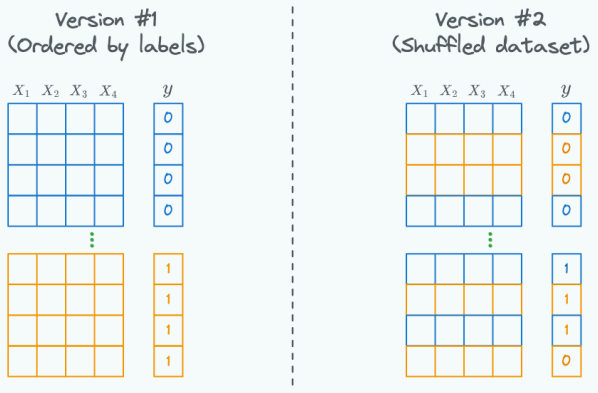
Consider a classification neural network trained using mini-batch gradient descent.

Mini-batch gradient descent: Update network weights using a few data points at a time.



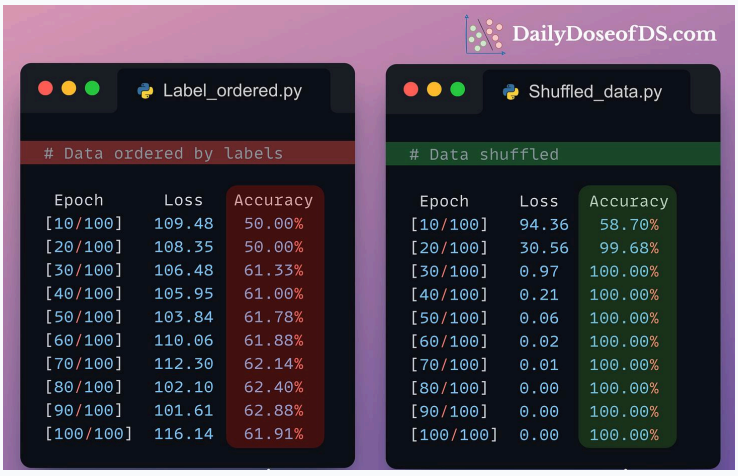
Here, we train two different neural networks:

- Version 1: The dataset is ordered by labels.
- Version 2: The dataset is properly shuffled by labels.



And, of course, before training, we ensure that both networks had the same initial weights, learning rate, and other settings.

The image depicts the epoch-by-epoch performance of the two models. On the left, we have the model trained on label-ordered data, and the one on the right was trained on the shuffled dataset.



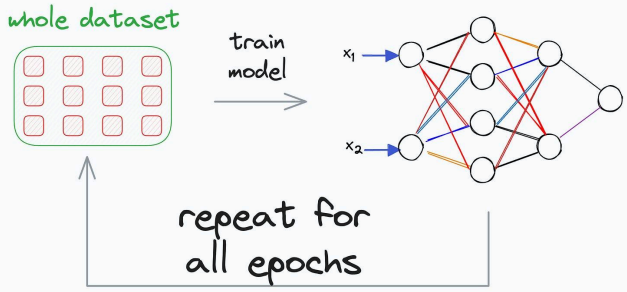
It is clear that the model receiving a label-ordered dataset miserably fails to converge while the other model, although overfits, shows that model has been learn effectively.

Why does that happen?

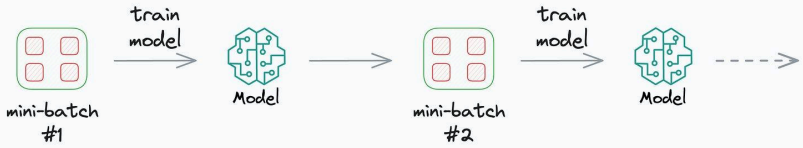
Now, if you think about it for a second, overall, both models received the same data, didn't they? Yet, the order in which the data was fed to these models totally determined their performance. I vividly remember that when I faced this issue, I knew that my data was ordered by labels.

Yet, it never occurred to me that ordering may influence the model performance because the data will always be the same regardless of the ordering.

But later, I realized that this point will only be valid when the model sees the entire data and updates the model weights in one go, i.e., in batch gradient descent, as depicted in this image.



But in the case of mini-batch gradient descent, the weights are updated after every mini-batch. Thus, the prediction and weight update on a subsequent mini-batch is influenced by the previous mini-batches.



In the context of label-ordered data, where samples of the same class are grouped together, mini-batch gradient descent will lead the model to learn patterns specific to the class it excessively saw early on in training. In contrast, randomly ordered data ensures that each mini-batch contains a balanced representation of classes. This allows the model to learn a more comprehensive set of features throughout the training process.

Of course, the idea of shuffling is not valid for time-series datasets as their temporal structure is important. The good thing is that if you happen to use, say, PyTorch DataLoader, you are safe. This is because it already implements shuffling. But if you have a custom implementation, ensure that you are not making any such error.

Before I end, one thing that you must ALWAYS remember when training neural networks is that these models can proficiently learn entirely non-existing patterns about your dataset. So never give them any chance to do so.

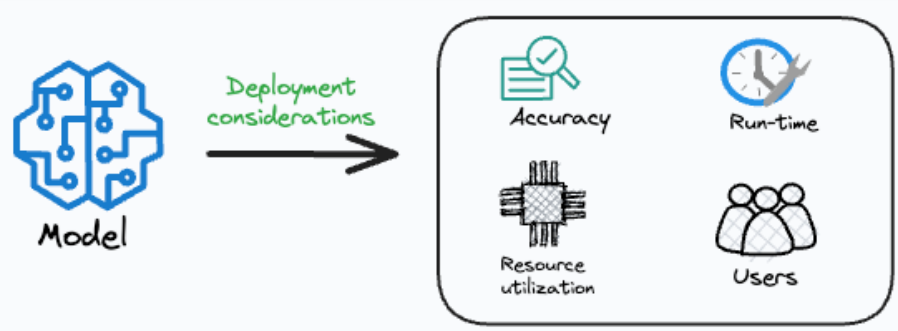
Model compression

Knowledge Distillation for Model Compression

Model accuracy alone (or an equivalent performance metric) rarely determines which model will be deployed.

This is because we also consider several operational metrics, such as:

- Inference Latency: Time taken by the model to return a prediction.
- Model size: The memory occupied by the model.
- Ease of scalability, etc.



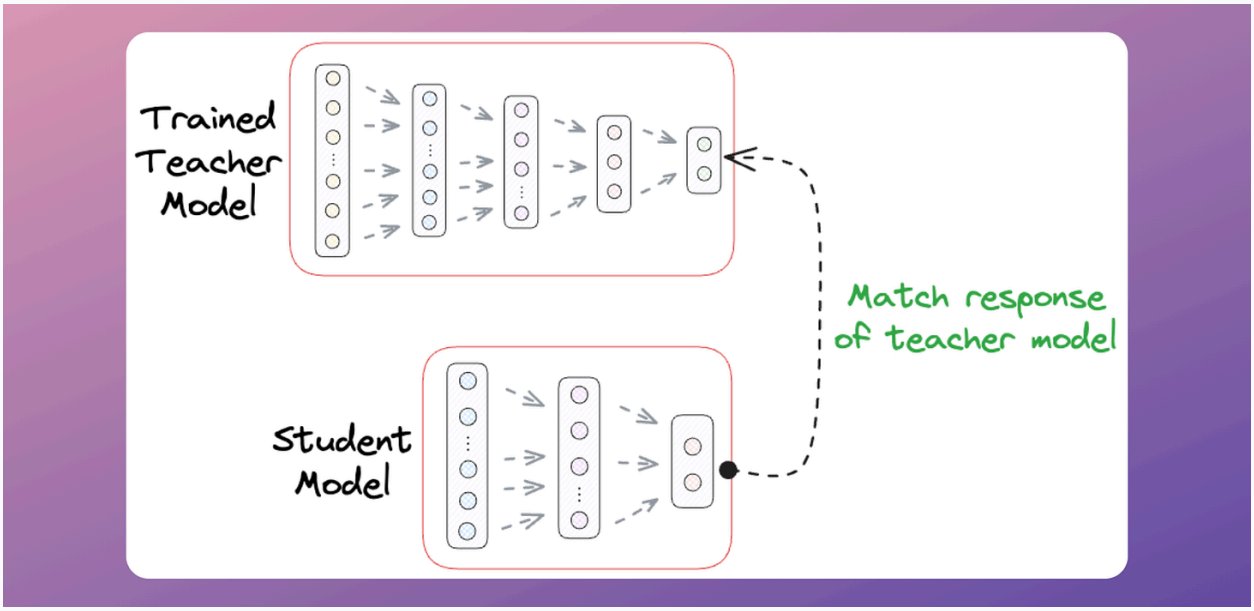
In this chapter, let me share a technique (with a demo) called knowledge distillation, which is commonly used to compress ML models and contribute to the above operational metrics.

What is knowledge distillation?

In a gist, the idea is to train a smaller/simpler model (called the "student" model) that mimics the behavior of a larger/complex model (called the "teacher" model).

This involves two steps:

- Train the teacher model as we typically would.
- Train a student model that matches the output of the teacher model.



If we compare it to an academic teacher-student scenario, the student may not be as performant as the teacher.

But with consistent training, a smaller model may get (almost) as good as the larger one.

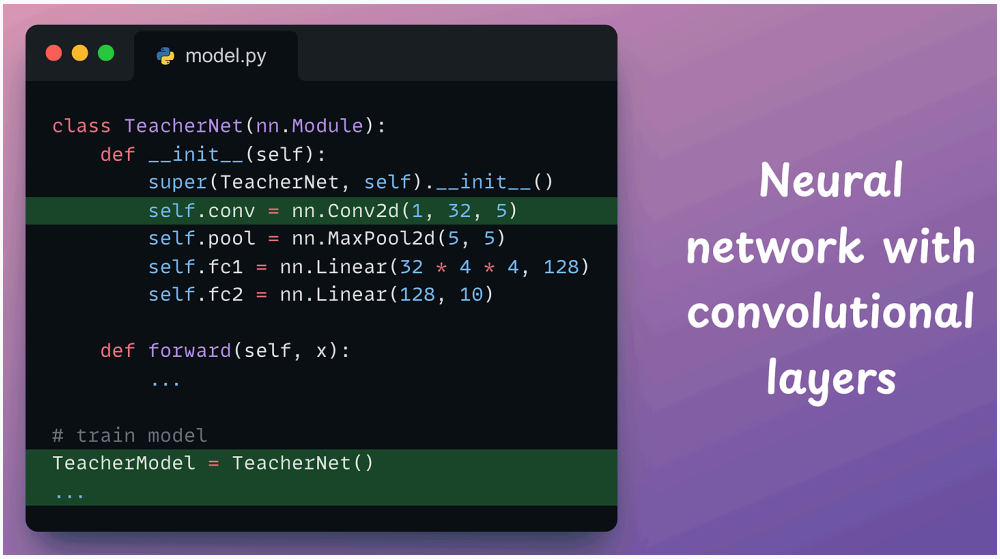
A classic example of a model developed in this way is DistillBERT. It is a student model of BERT.

- DistilBERT is approximately 40% smaller than BERT, which is a massive difference in size.
- Still, it retains approximately 97% of the BERT's capabilities.

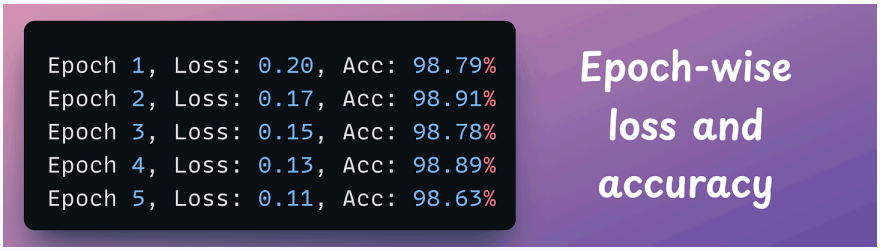
Next, let's look at a demo.

Knowledge distillation demo

In the interest of time, let's say we have already trained the following CNN model on the MNIST dataset (I have provided the full Jupyter notebook towards the end, don't worry):

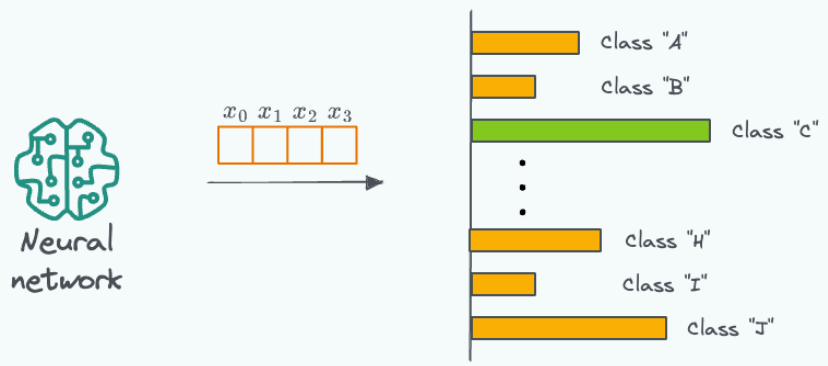


The epoch-by-epoch training loss and validation accuracy is depicted below:

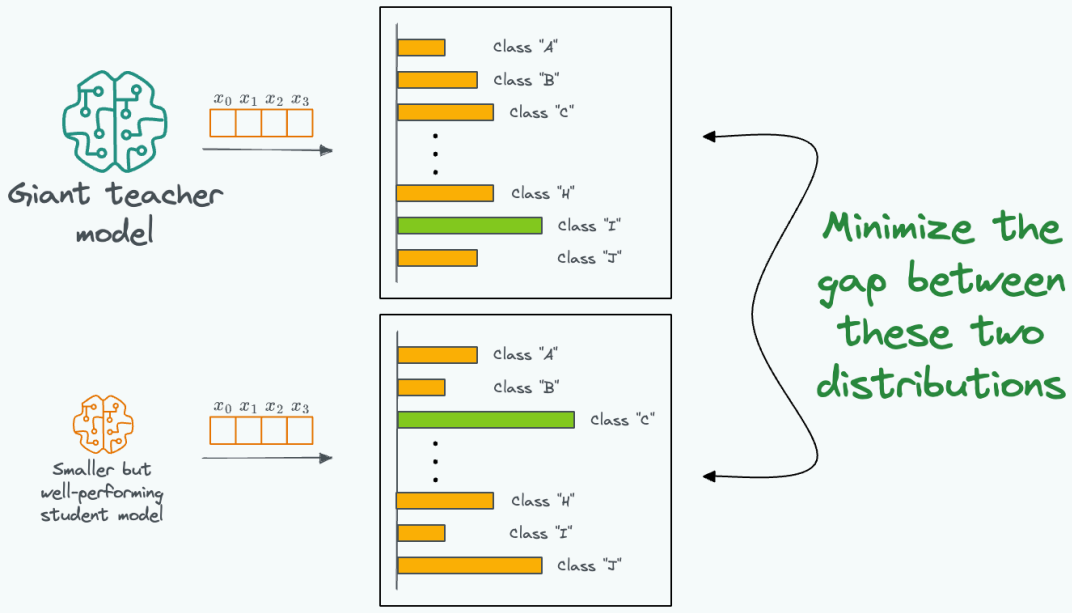


Next, let's define a simpler model without any convolutional layers:

Being a classification model, the output will be a probability distribution over the <N> classes:



Thus, we can train the student model such that its probability distribution matches that of the teacher model.



One way to do this is to use KL divergence as a loss function.

$$D_{KL}(P \parallel Q) = \sum_{x} P(x) \cdot \log\left(\frac{P(x)}{Q(x)}\right)$$

It measures how much information is lost when we use distribution Q to approximate distribution P.

A question for you: What will be the KL divergence if P=Q?

Thus, in our case:

- P will be the probability distribution from the teacher model.
- Q will be the probability distribution from the student model.

The loss function is implemented below:

KL divergence loss function

```python
def KL_loss(student_logits, teacher_logits):

    # convert teacher model outputs to probabilities
    p_teacher = F.softmax(teacher_logits, dim=1)

    # convert student model outputs to probabilities
    p_student = F.log_softmax(student_logits, dim=1)

    # compute KL divergence loss (PyTorch's method)
    return F.kl_div(p_student, p_teacher, reduction='batchmean')
```

Finally, we train the student model as follows:

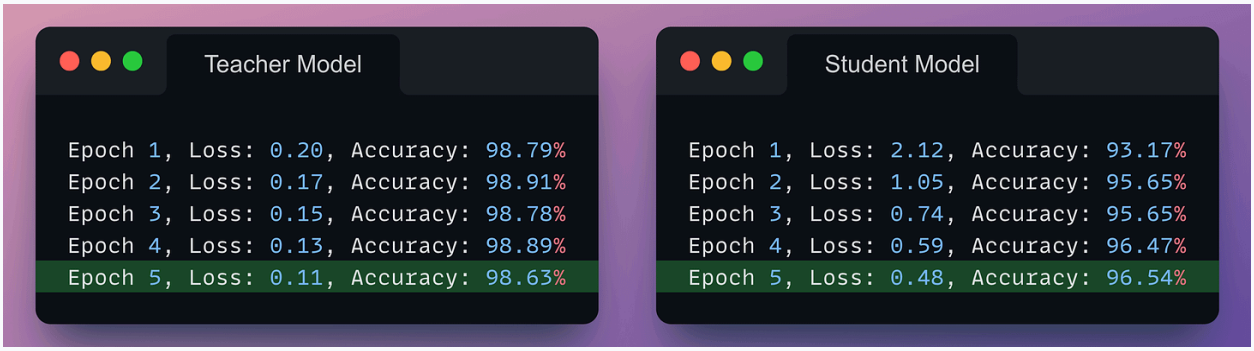
Student model training loop

model.py

```python
for epoch in range(5):
    ...

    student_optimizer.zero_grad()

    # student outputs
    student_logits = student_model(inputs)

    # teacher outputs (detach to avoid backpropagation)
    teacher_logits = teacher_model(inputs).detach()

    # KL Divergence loss
    loss = KL_loss(student_logits, teacher_logits)

    # backpropagation step
    loss.backward()
    student_optimizer.step()
```

Done!

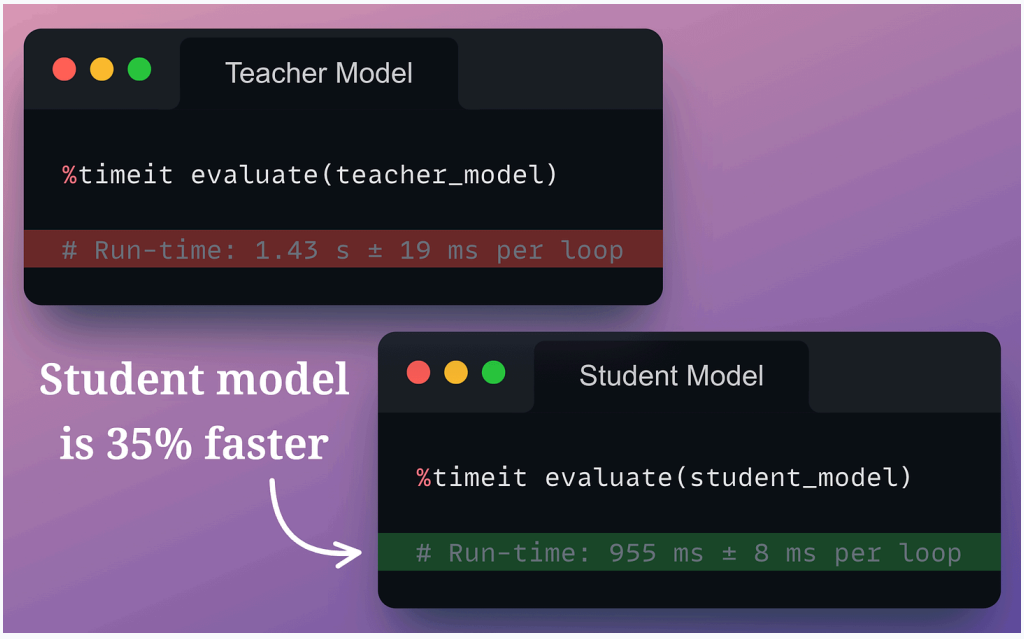
The following image compares the training loss and validation accuracy of the two models:



Of course, as shown in the highlighted lines above, the performance of the student model is not as good as the teacher model, which is expected.

However, it is still pretty promising, given that it was only composed of simple feed-forward layers.

Also, as depicted below, the student model is approximately 35% faster than the teacher model, which is a significant increase in the inference run-time of the model for about a 1-2% drop in the test accuracy.



That said, one of the biggest downsides of knowledge distillation is that one must still train a larger teacher model first to train the student model.

But in a resource-constrained environment, it may not be feasible to train a large teacher model.

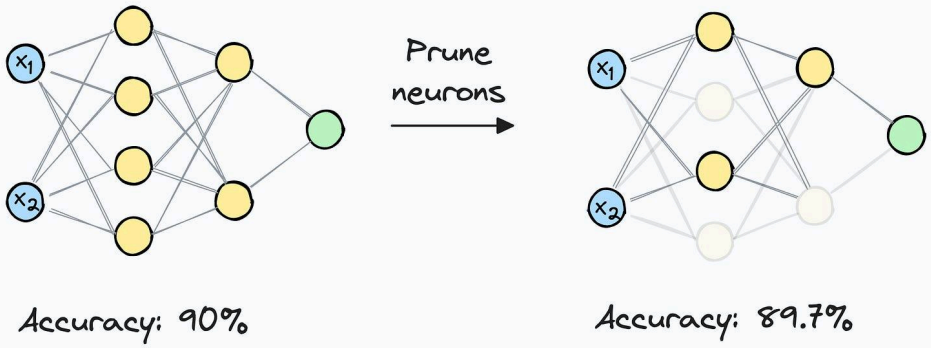
So this technique assumes that we are not resource-constrained at least in the development environment.

In the next chapter, let's discuss one more technique to compress ML models and reduce their memory footprint.

Activation Pruning

Once we complete network training, we are almost always left with plenty of useless neurons — ones that make nearly zero contribution to the network's performance, but they still consume memory.

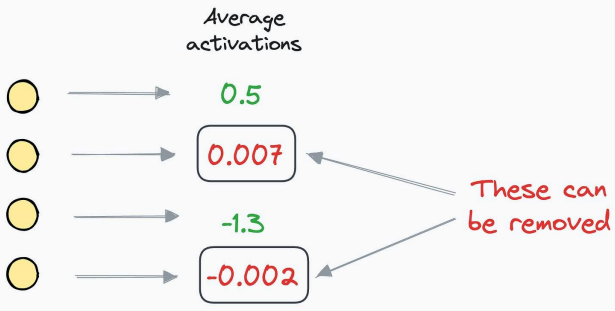
In other words, there is a high percentage of neurons, which, if removed from the trained network, will not affect the performance remarkably:



And, of course, I am not saying this as a random and uninformed thought. I have experimentally verified this over and over across my projects.

Here's the core idea.

After training is complete, we run the dataset through the model (no backpropagation this time) and analyze the average activation of individual neurons. Here, we often observe that many neuron activations are always close to near-zero values.



Thus, they can be pruned from the network, as they will have very little impact on the model's output.

For pruning, we can decide on a pruning threshold (λ) and prune all neurons whose activations are less than this threshold.

This makes intuitive sense as well.

More specifically, if a neuron rarely possesses a high activation value, then it is fair to assume that it isn't contributing to the model's output, and we can safely prune it.

The following table compares the accuracy of the pruned model with the original (full) model across a range of pruning thresholds (λ):

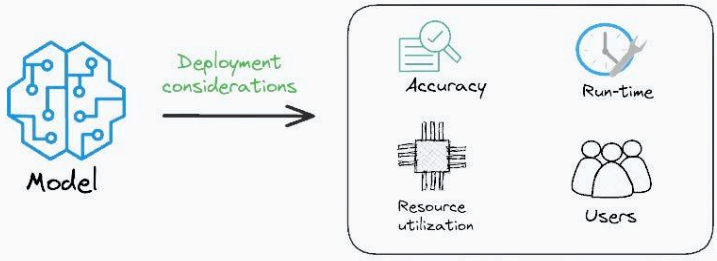
	Pruning Threshold (λ)	Validation Accuracy	Parameter Reduction
Original Model	-	**96.7**	-
Pruned Models	0.1	96.62	62%
	0.2	96.41	66%
	0.3	96.31	69%
	0.4	96.08	72%
	0.5	94.88	74%
	0.6	94.49	76%
	0.7	92.19	79%
	0.8	90.65	81%
	0.9	88.39	83%
	1	87.87	84%

At a pruning threshold λ=0.4, the validation accuracy of the model drops by just 0.62%, but the number of parameters drops by 72%.

That is a huge reduction, while both models being almost equally good! Of course, there is a trade-off because we are not doing as well as the original model. But in many cases, especially when deploying ML models, accuracy is not the only primary metric that decides these.

Instead, several operational metrics like efficiency, speed, memory consumption, etc., are also a key deciding factor.

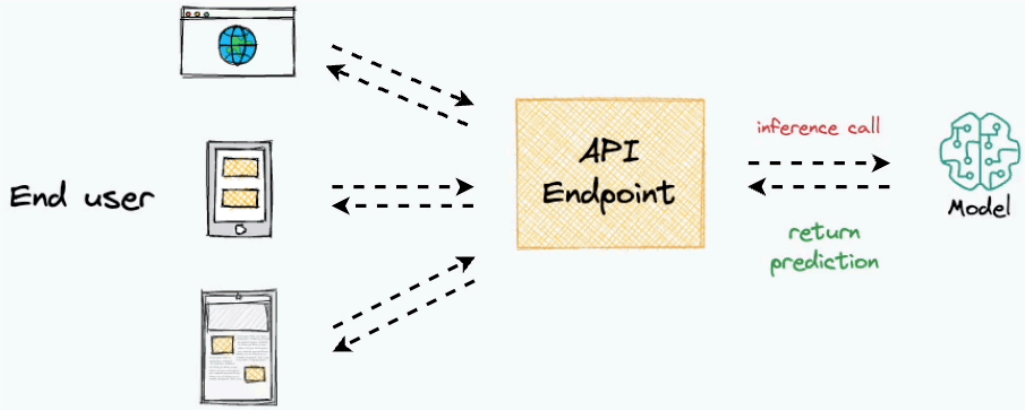
That is why model compression techniques are so crucial in such cases.



Deployment

Deploy ML Models from Jupyter Notebook

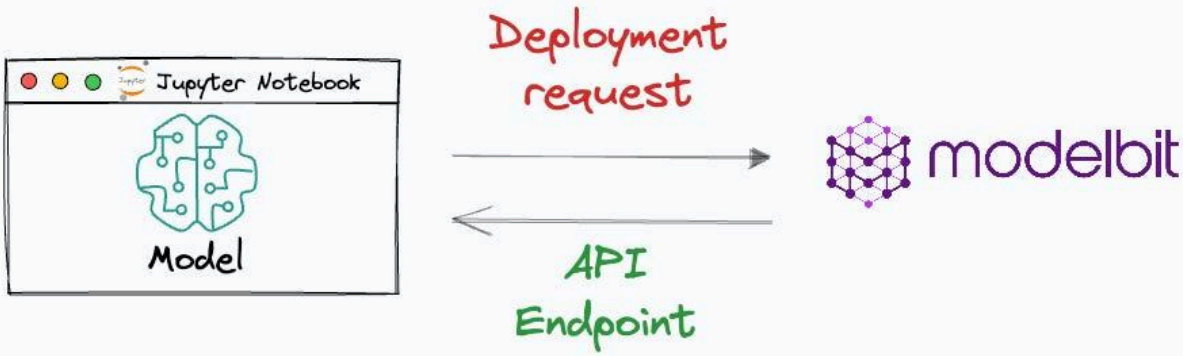
The core objective of model deployment is to obtain an API endpoint that can be used for inference purposes:



While this sounds simple, deployment is typically quite a tedious and time-consuming process. One must maintain environment files, configure various settings, ensure all dependencies are correctly installed, and many more.

So, in this chapter, I want to help you simplify this process. More specifically, we shall learn how to deploy any ML model right from a Jupyter Notebook in just three simple steps using the Modelbit API.

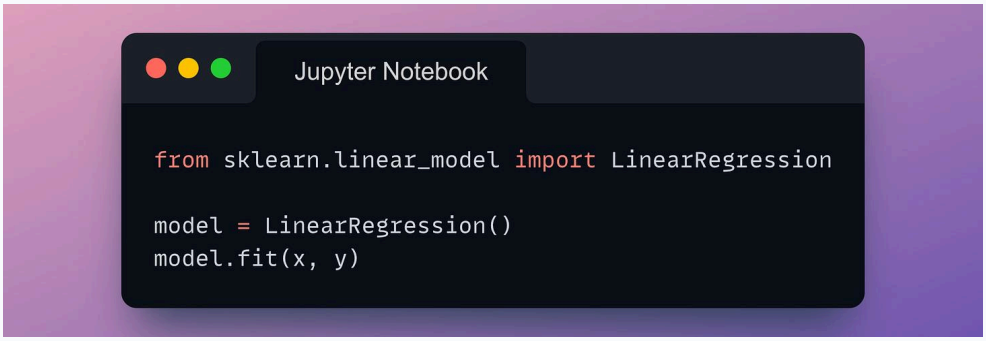
Modelbit lets us seamlessly deploy ML models directly from our Python notebooks (or git) to Snowflake, Redshift, and REST.



Deployment with Modelbit

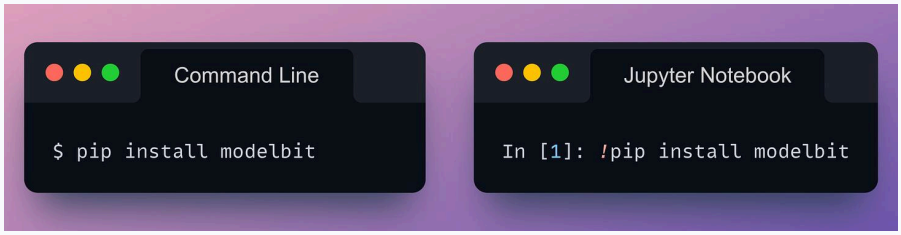
Assume we have already trained our model.

For simplicity, let's assume it to be a linear regression model trained using sklearn, but it can be any other model as well:

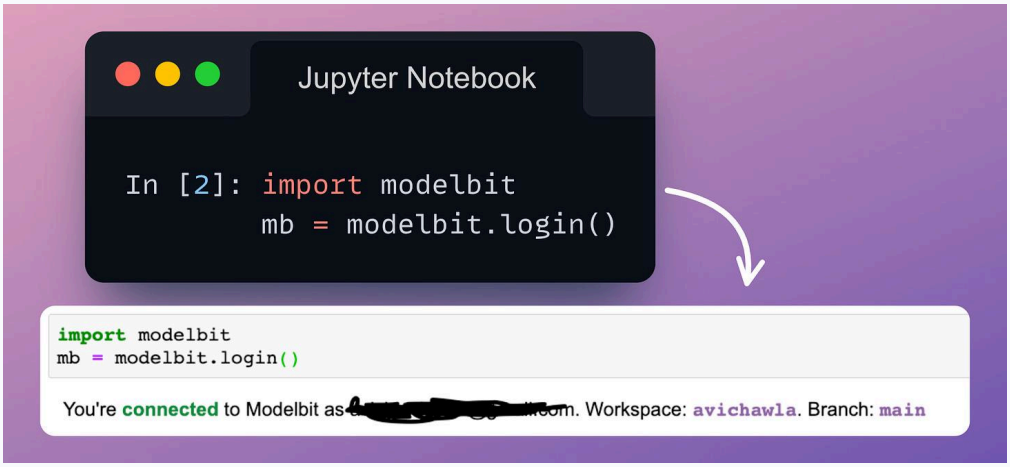


Let's see how we can deploy this model with Modelbit!

- First, we install the Modelbit package via pip:



- Next, we log in to Modelbit from our Jupyter Notebook (make sure you have created an account here: Modelbit)



- Finally, we deploy it, but here's an important point to note:

To deploy a model using Modelbit, we must define an inference function.

Simply put, this function contains the code that will be executed at inference. Thus, it will be responsible for returning the prediction.

```
def inference_function():

    # prediction code
```

A function that returns model predictions

We must specify the input parameters required by the model in this method. Also, we can name it anything we want.

For our linear regression case, the inference function can be as follows:

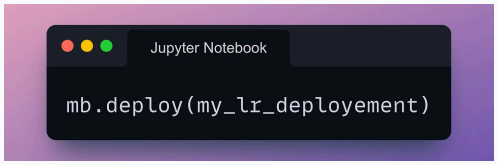
Jupyter Notebook

```
def my_lr_deployement(input_x):

    if isinstance(input_x, (int, float)):    #  check input type
        return model.predict([[input_x]])[0] # prediction

    else:
        return None
```

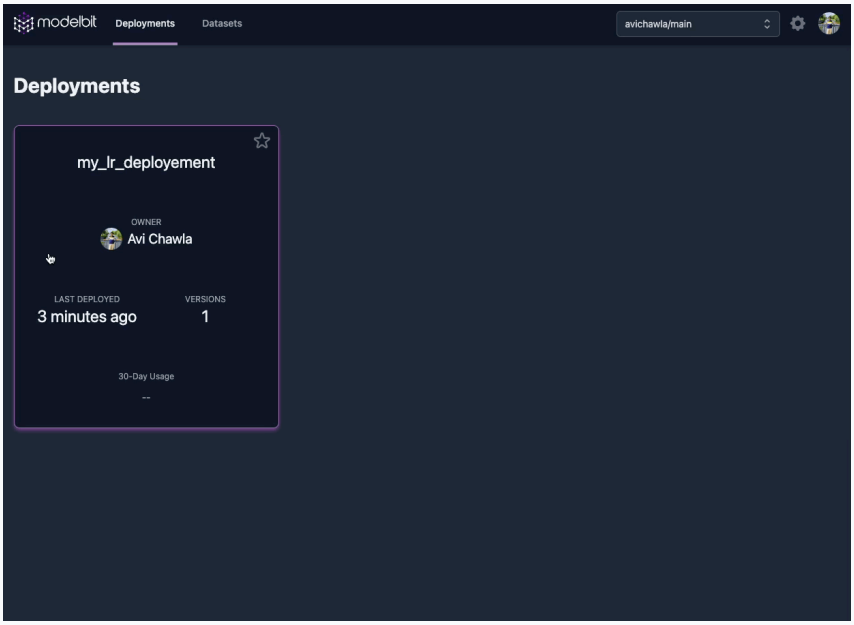
- We define a function `my_lr_deployment()`.
- Next, we specify the input of the model as a parameter of this method.
- We validate the input for its data type.
- Finally, we return the prediction.

One good thing about Modelbit is that every dependency of the function (the model object in this case) is pickled and sent to production automatically along

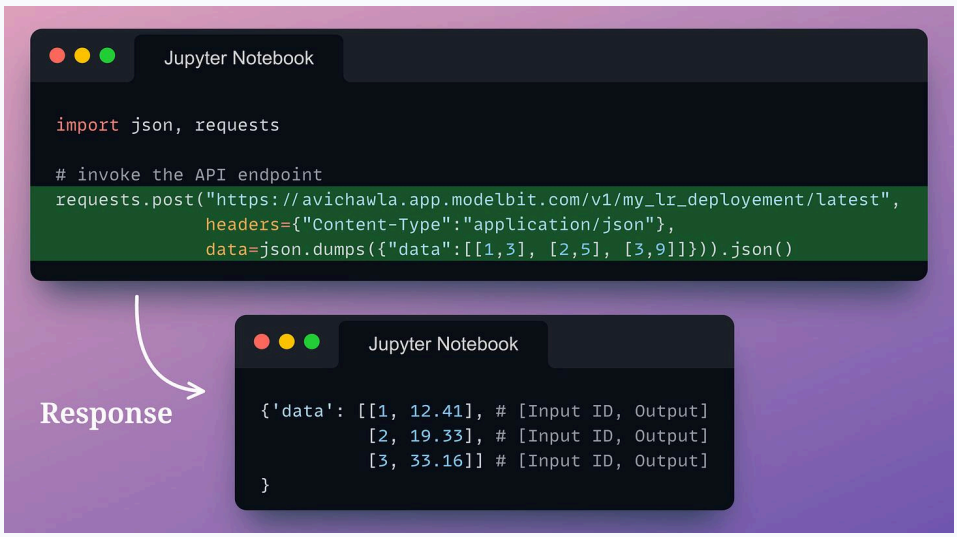
with the function. Thus, we can reference any object in this method. Once we have defined the function, we can proceed with deployment as follows:



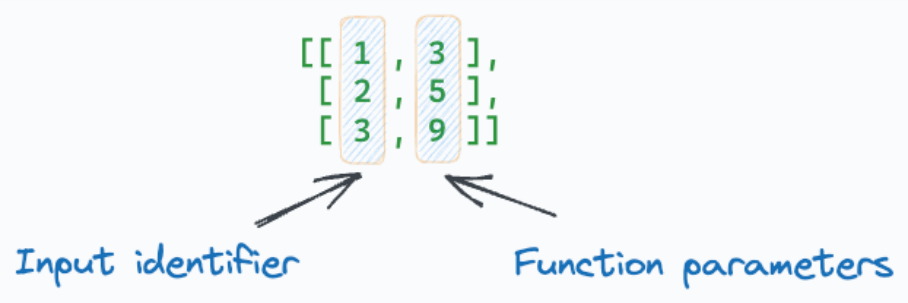
We have successfully deployed the model in three simple steps, that too, right from the Jupyter Notebook! Once our model has been successfully deployed, it will appear in our Modelbit dashboard.



As shown above, Modelbit provides an API endpoint. We can use it for inference purposes as follows:



In the above request, data passed to the endpoint is a list of lists.



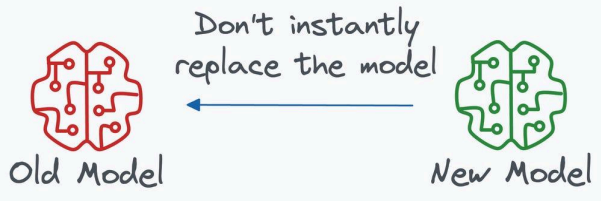
The first number in the list is the input ID. All entries following the ID in a list are the function parameters.

Lastly, we can also specify specific versions of the libraries or Python used while deploying our model. This is depicted below:

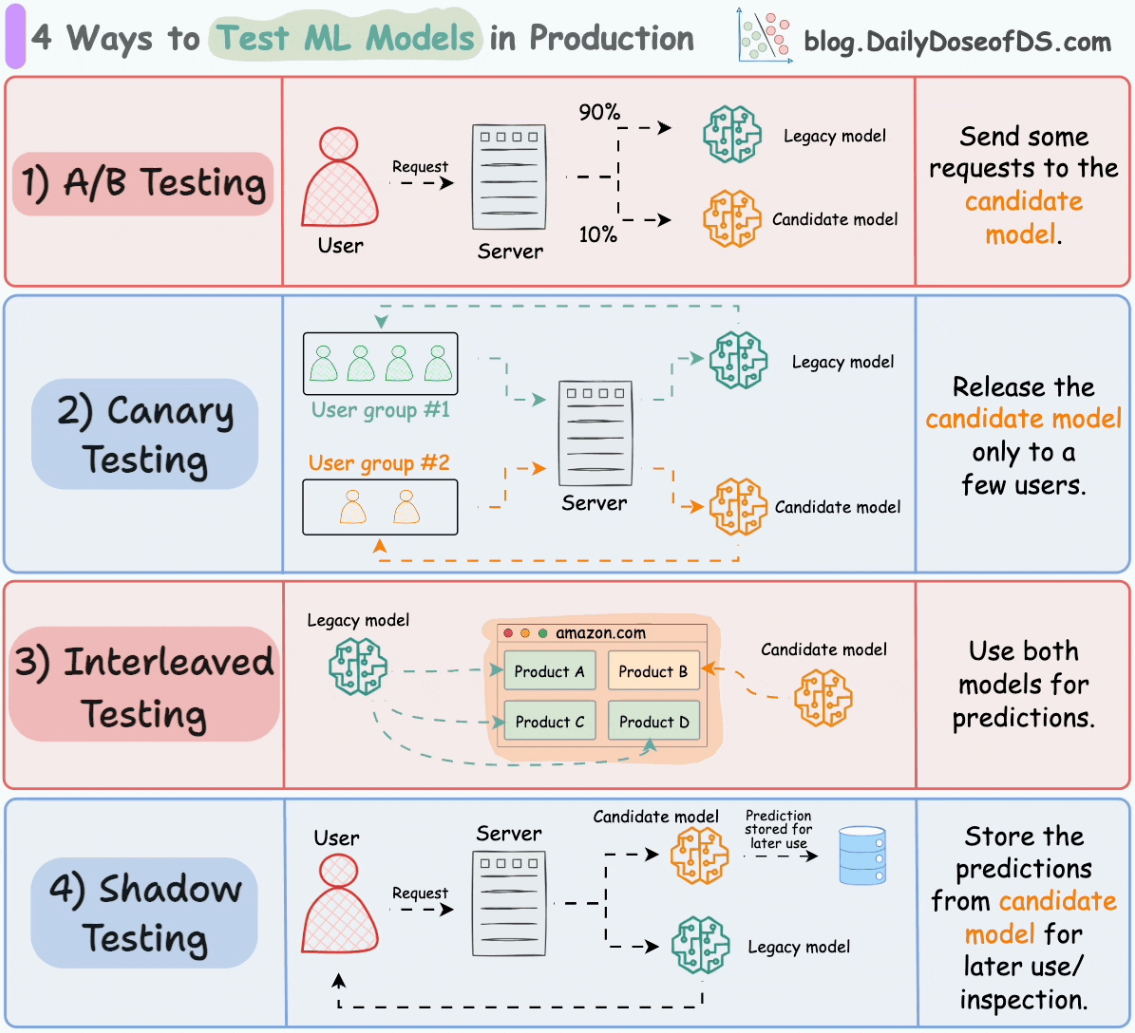
Isn't that cool, simple, and elegant over traditional deployment approaches?

4 Ways to Test ML Models in Production

Despite rigorously testing an ML model locally (on validation and test sets), it could be a terrible idea to instantly replace the previous model with a new model.

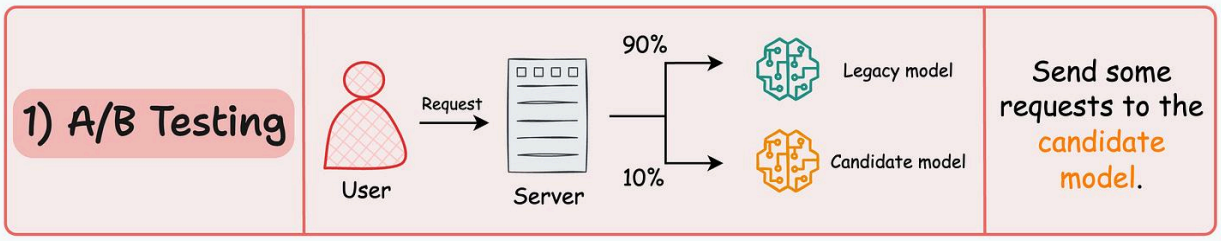


A more reliable strategy is to test the model in production (yes, on real-world incoming data). While this might sound risky, ML teams do it all the time, and it isn't that complicated. The following visual depicts 4 common strategies to do so:



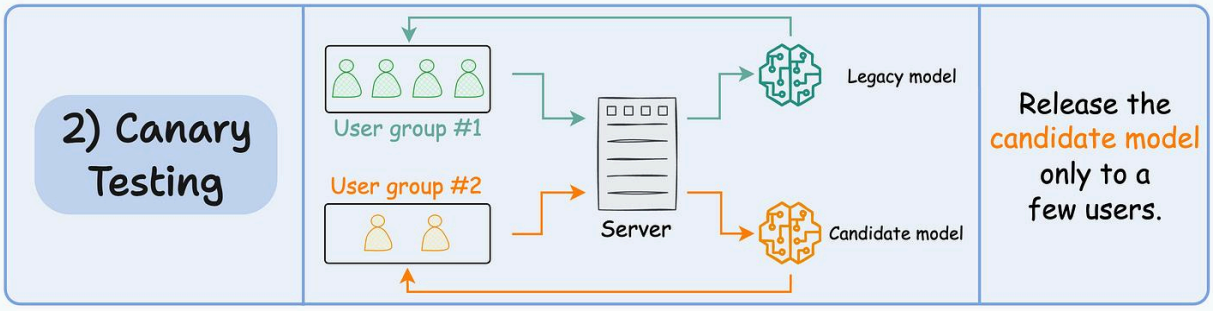
- The current model is called the legacy model.
- The new model is called the candidate model.

#1) A/B testing



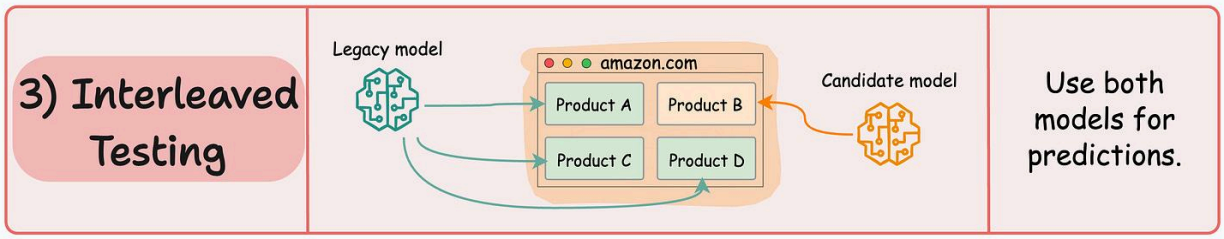
- Distribute the incoming requests non-uniformly between the legacy model and the candidate model.
- Intentionally limit the exposure of the candidate model to avoid any potential risks. Thus, the number of requests sent to the candidate model must be low.

#2) Canary testing



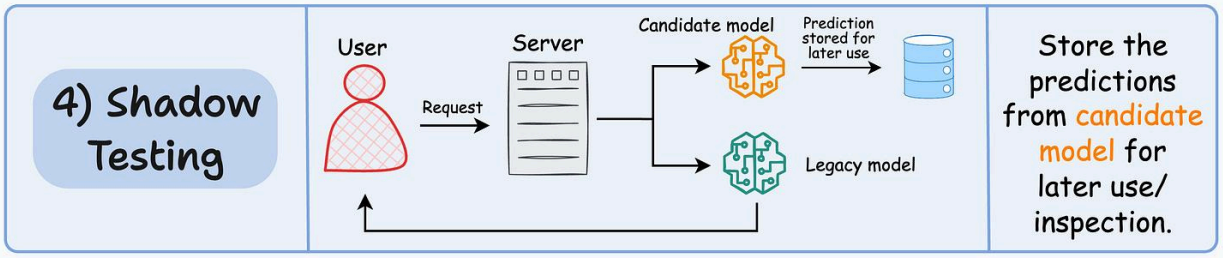
- In A/B testing, since traffic is randomly redirected to either model irrespective of the user, it can potentially affect all users.
- In canary testing, the candidate model is released to a small subset of users in production and gradually rolled out to more users.

#3) Interleaved testing



- This involves mixing the predictions of multiple models in the response.
- Consider Amazon's recommendation engine. In interleaved deployments, some product recommendations displayed on the homepage can come from the legacy model, while some can come from the candidate model.

#4) Shadow testing



- All of the above techniques affect some (or all) users.
- Shadow testing (or dark launches) lets us test a new model in a production environment without affecting the user experience.
- The candidate model is deployed alongside the existing legacy model and serves requests like the legacy model. However, the output is not sent back to the user. Instead, the output is logged for later use to benchmark its performance against the legacy model.
- We explicitly deploy the candidate model instead of testing offline because the production environment is difficult to replicate offline.

Shadow testing offers risk-free testing of the candidate model in a production environment.

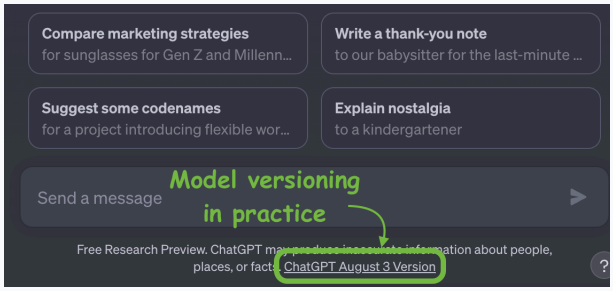
Version Controlling and Model Registry

Real-world ML deployment is never just about "deployment" — host the model somewhere, obtain an API endpoint, integrate it into the application, and you are done!

This is because, in reality, plenty of things must be done post-deployment to ensure the model's reliability and performance.

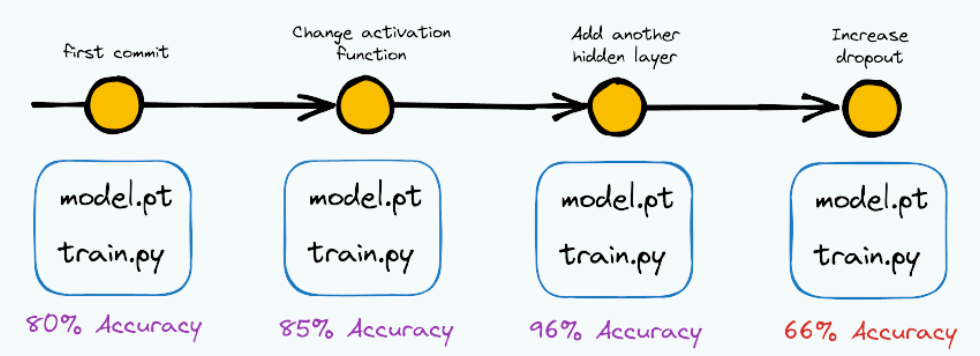
#1) Version control

To begin, it is immensely crucial to version control ML deployments. You may have noticed this while using ChatGPT, for instance.



But updating does not simply mean overwriting the previous version.

Instead, ML models are always version-controlled (using git tools).



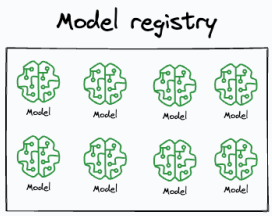
The advantages of version-controlling ML deployments are pretty obvious:

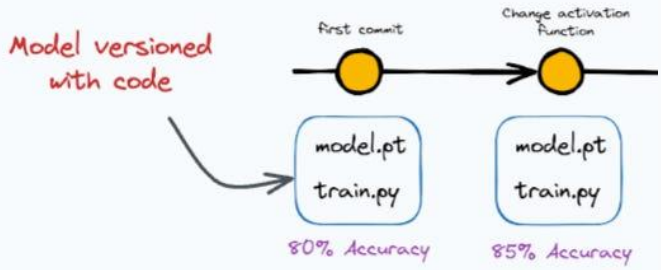
- In case of sudden mishaps post-deployment, we can instantly roll back to an older version.
- We can facilitate parallel development with branching, and many more.

#2) Model registry

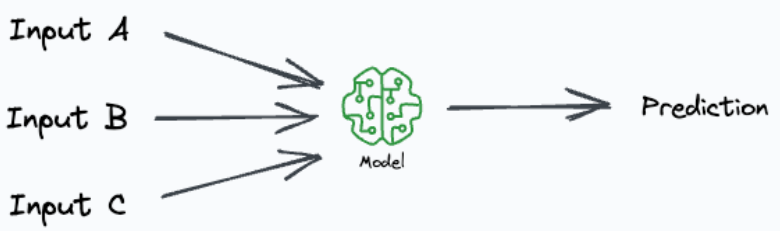
Another practical idea is to maintain a model registry for deployments. Let's understand what it is.

Simply put, a model registry can be considered repository of models. See, typically, we might be inclined to version our code and the ML model together:

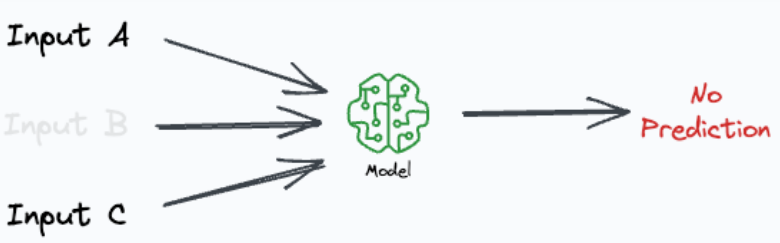




However, when we use a model registry, we version models separately from the code. Let me give you an intuitive example to understand this better. Imagine our deployed model takes three inputs to generate a prediction:



While writing the inference code, we overlooked that, at times, one of the inputs might be missing. We realized this by analyzing the model's logs.



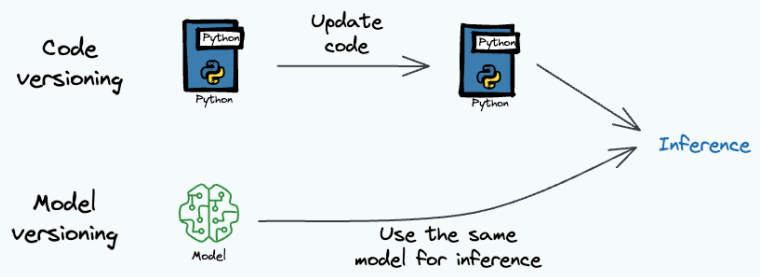
We may want to fix this quickly (at least for a while) before we decide on the next steps more concretely. Thus, we may decide to update the inference code by assigning a dummy value for the missing input.

This will allow the model to still process the incoming request.

Let me ask you a question: "Did we update the model?"

No, right?

Here, we only need to update the inference code. The model will remain the same.



But if we were to version the model and code together, it would lead to a redundant model and take up extra space.

However, by maintaining a model registry:

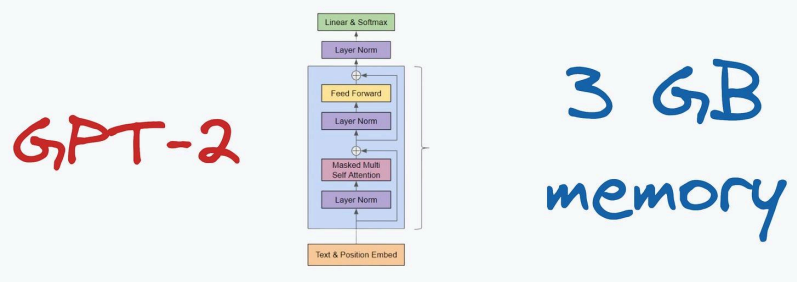
- We can only update the inference code.
- Avoid pushing a new (yet unwanted) model to deployment.

This makes intuitive sense as well, doesn't it?

LLMs

Where Did the GPU Memory Go?

GPT-2 (XL) has 1.5 Billion parameters, and its parameters consume ~3GB of memory in 16-bit precision.



Under 16-bit precision, one parameter takes up 2 bytes of memory, so 1.5B parameters will consume 3GB of memory.

What's your estimate for the minimum memory needed to train GPT-2 on a single GPU?

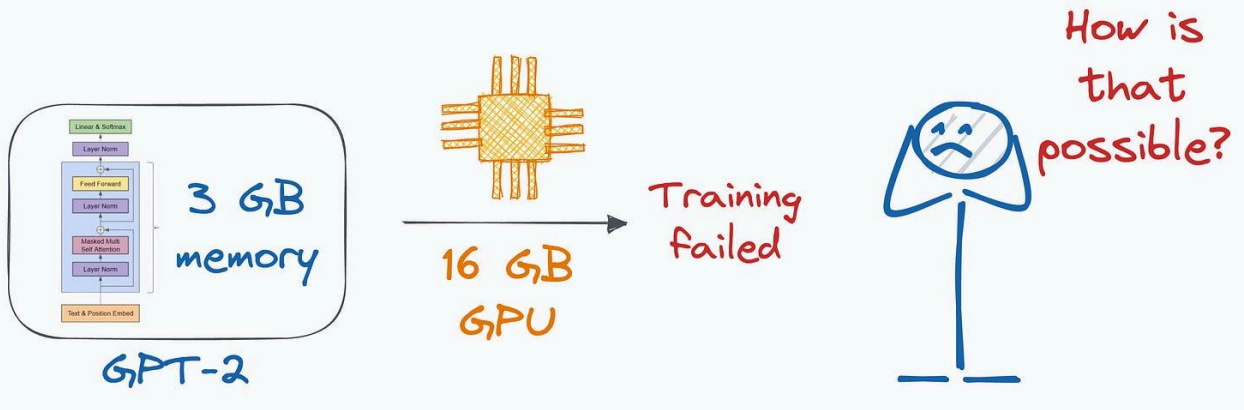
- Optimizer → Adam
- Batch size → 32
- Number of transformer layers → 48
- Sequence length → 1000

What's your estimate for the minimum memory needed to train a GPT-2 model (size 3GBs) on a single GPU?

- 4-6 GB
- 8-10 GB
- 12-15 GB
- 32-35 GB
- 50+ GB

The answer might surprise you.

One can barely train a 3GB GPT-2 model on a single GPU with 32GB of memory.



But how could that be even possible? Where does all the memory go?

Let's understand.

There are so many fronts on which the model consistently takes up memory during training.

#1) Optimizer states, gradients, and parameter memory

Mixed precision training is widely used to speed up model training.

As the name suggests, the idea is to utilize lower-precision `float16` (wherever feasible, like in convolutions and matrix multiplications) along with `float32` — that is why the name "mixed precision."

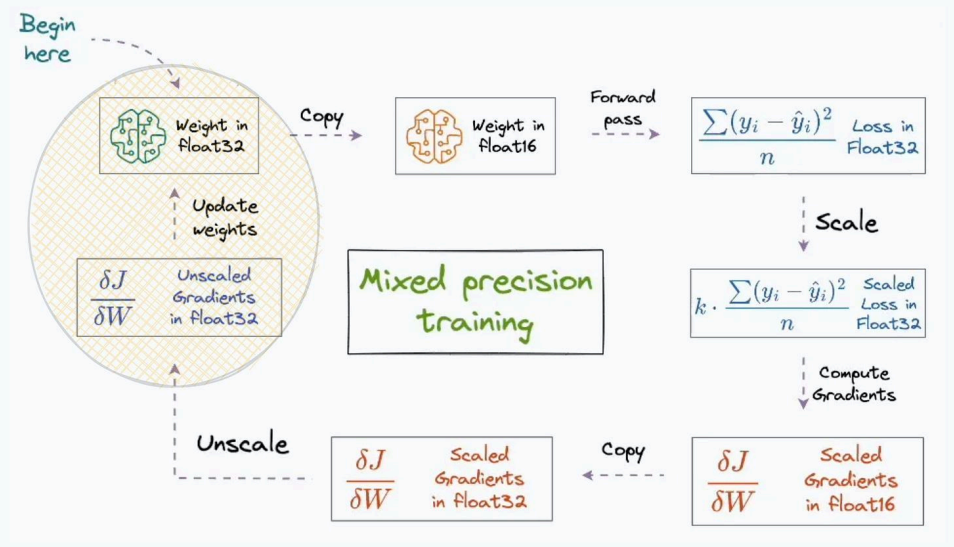
Both the forward and backward propagation are performed using the 16-bit representations of weights and gradients.

Thus, if the model has Φ parameters, then:

- Weights will consume $2*\Phi$ bytes.
- Gradients will consume $2*\Phi$ bytes.

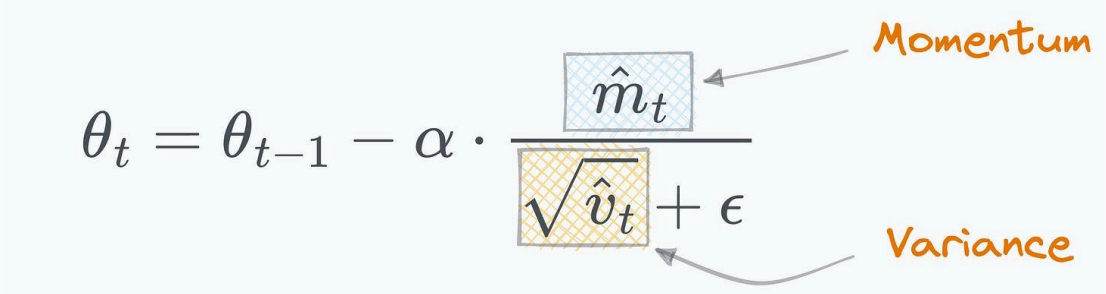
Here, the figure "2" represents a memory consumption of 2 bytes/paramter (16-bit).

Moreover, the updates at the end of the backward propagation are still performed under 32-bit for effective computation. I am talking about the circled step in the image below:



Adam is one of the most popular optimizers for model training.

While many practitioners use it just because it is popular, they don't realize that during training, Adam stores two optimizer states to compute the updates — momentum and variance of the gradients:



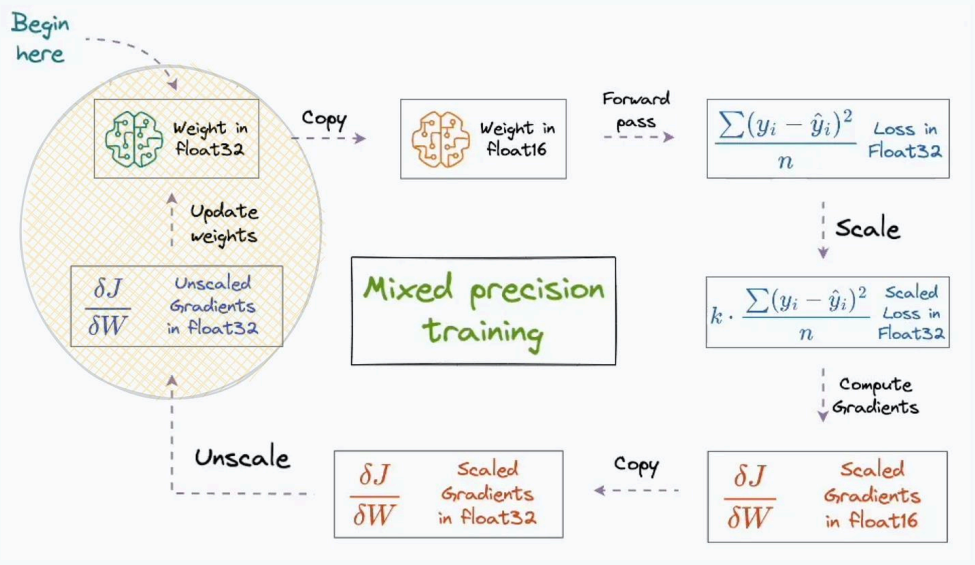
$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Thus, if the model has Φ parameters, then these two optimizer states will consume:

- 4*Φ bytes for momentum.

- Another 4*Φ bytes for variance.

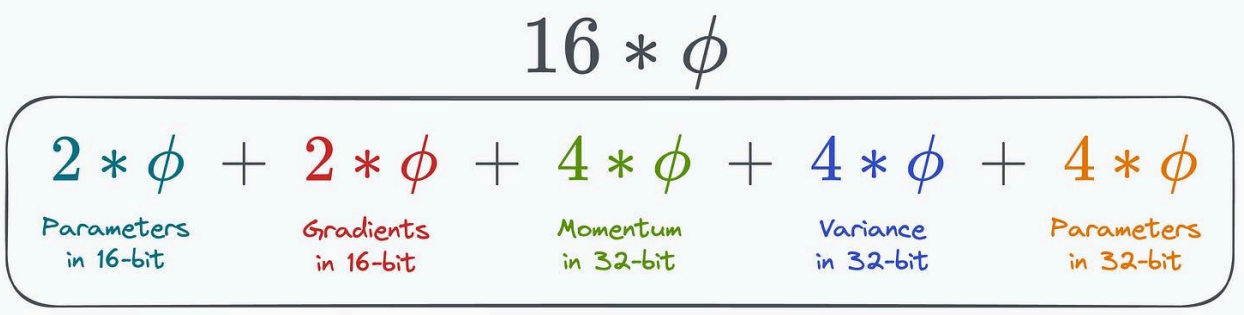
Here, the figure "4" represents a memory consumption of 4 bytes/paramter (32-bit).



Lastly, as shown in the figure above, the final updates are always adjusted in the 32-bit representation of the model weights. This leads to:

- Another 4*Φ bytes for model parameters.

Let's sum them up:



That's 16*Φ, or 24GB of memory, which is ridiculously higher than the 3GB memory utilized by 16-bit parameters.

And we haven't considered everything yet.

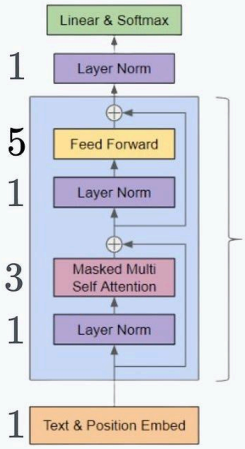
#2) Activations

For big deep learning models, like LLMs, Activations take up significant memory during training.

More formally, the total number of activations computed in one transform block of GPT-2 are:

$$12 * H * S * B$$

Hidden size · Sequence length · Batch size



Thus, across all transformer blocks, this comes out to be:

$$12 * H * S * B * N$$

Hidden size · Sequence length · Batch size · Transformer layers

This is the configuration for GPT2-XL:

$$12 * 1600 * 1000 * 32 * 48$$

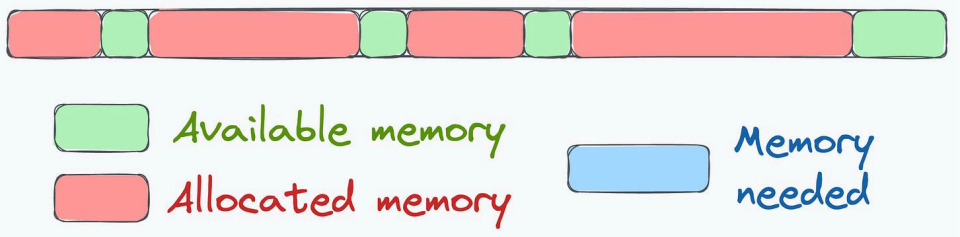
Hidden size · Sequence length · Batch size · Transformer layers

This comes out to be ~30B activations. As each activation is represented in 16-bit, all activations collectively consume 60GB of memory.

With techniques like gradient checkpointing (discussed in the previous chapter), this could be brought down to about 8-9GB at the expense of 25-30% more run-time.

This technique takes complete memory consumption to about 32-35GB range, which I mentioned earlier, for a meager 3GB model, and that too with a pretty small batch size of just 32. On top of this, there are also some more overheads involved, like memory fragmentation.

It occurs when there are small, unused gaps between allocated memory blocks, leading to inefficient use of the available memory.



Memory allocation requests fail because of the unavailability of contiguous memory blocks.

Conclusion

In the above discussion, we considered a relatively small model — GPT-2 (XL) with 1.5 Billion parameters, which is tiny compared to the scale of models being trained these days.

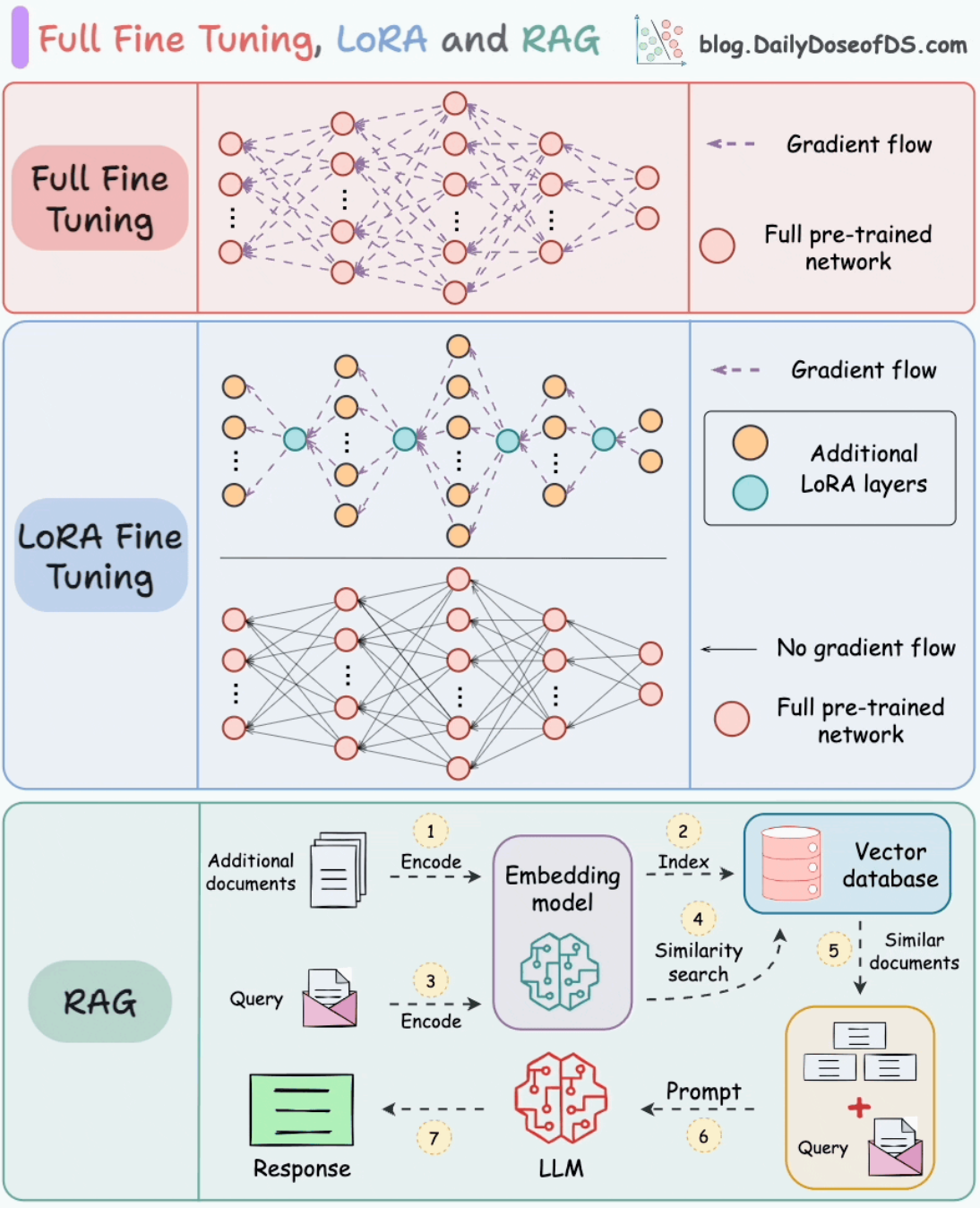
However, the discussion may have helped you reflect on the inherent challenges of building LLMs. Many people often say that GPTs are only about stacking more and more layers in the model and making the network bigger.

If it was that easy, everybody would have been doing it. From this discussion, you may have understood that it's not as simple as appending more layers.

Even one additional layer can lead to multiple GBs of additional memory requirement. Multi-GPU training is at the forefront of these models, which we covered in an earlier chapter in this book.

Full-model Fine-tuning vs. LoRA vs. RAG

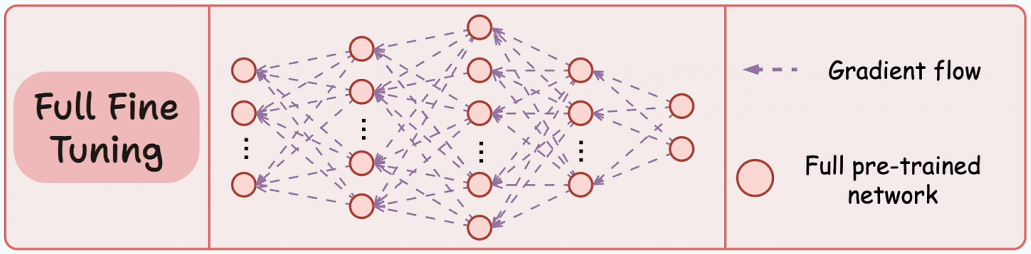
Here's a visual which illustrates "full-model fine-tuning," "fine-tuning with LoRA," and "retrieval augmented generation (RAG)."



All three techniques are used to augment the knowledge of an existing model with additional data.

#1) Full fine-tuning

Fine-tuning means adjusting the weights of a pre-trained model on a new dataset for better performance.

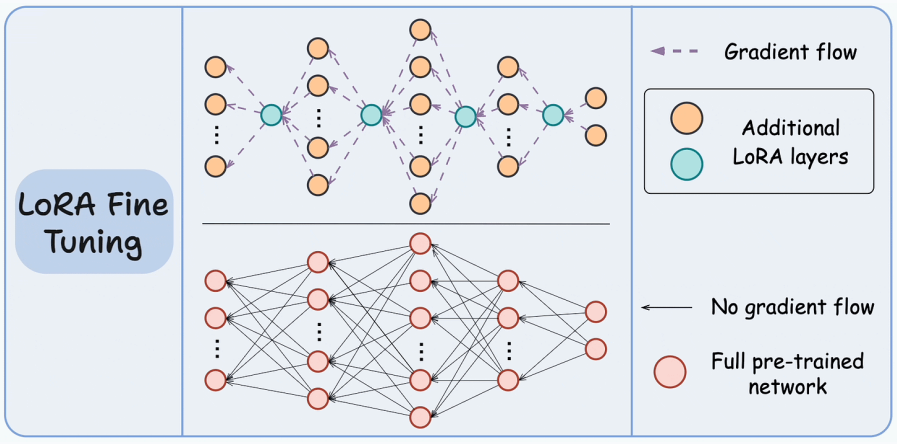


While this fine-tuning technique has been successfully used for a long time, problems arise when we use it on much larger models — LLMs, for instance, primarily because of:

- Their size.
- The cost involved in fine-tuning all weights.
- The cost involved in maintaining all large fine-tuned models.

#2) LoRA fine-tuning

LoRA fine-tuning addresses the limitations of traditional fine-tuning. The core idea is to decompose the weight matrices (some or all) of the original model into low-rank matrices and train them instead. For instance, in the graphic below, the bottom network represents the large pre-trained model, and the top network represents the model with LoRA layers.



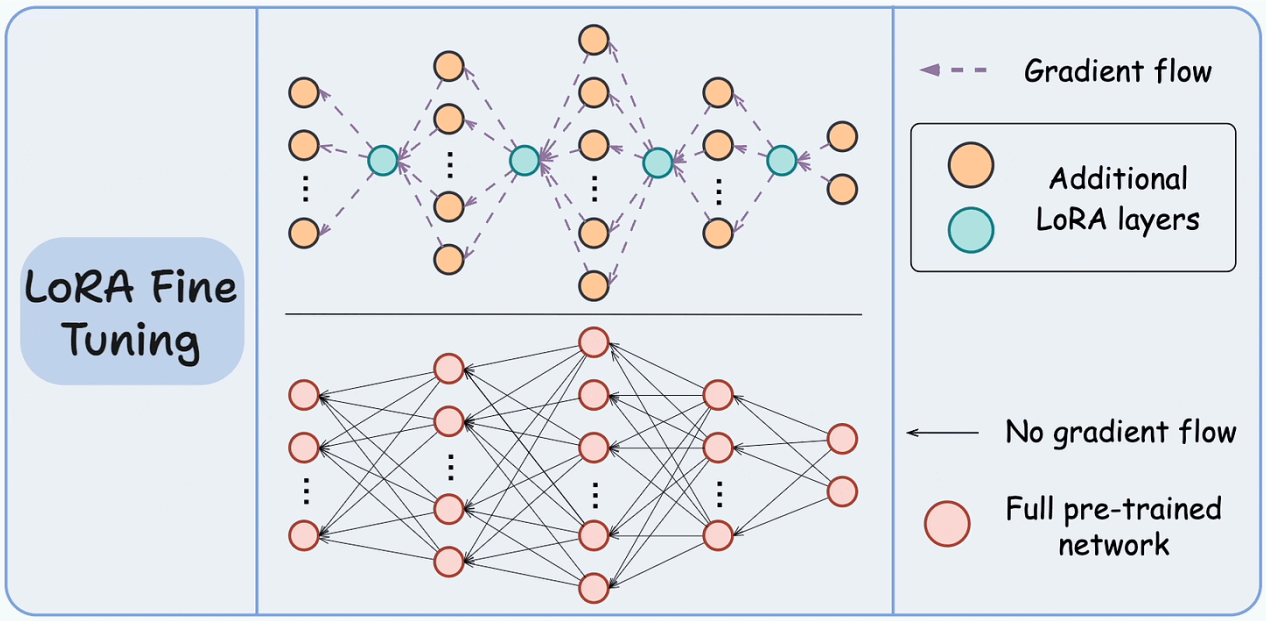
The idea is to train only the LoRA network and freeze the large model.

Looking at the above visual, you might think:

But the LoRA model has more neurons than the original model. How does that help? To understand this, you must make it clear that neurons don't have anything to do with the memory of the network.

They are just used to illustrate the dimensionality transformation from one layer to another.

It is the weight matrices (or the connections between two layers) that take up memory. Thus, we must be comparing these connections instead:

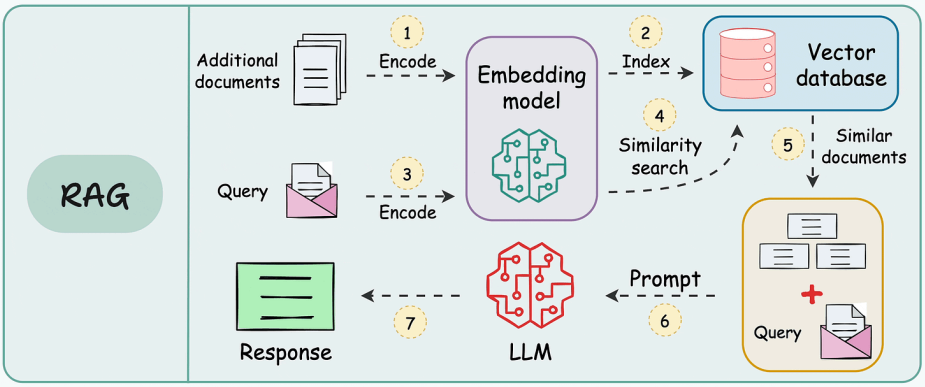


Looking at the above visual, it is pretty clear that the LoRA network has relatively very few connections.

#3) RAG

Retrieval augmented generation (RAG) is another pretty cool way to augment neural networks with additional information, without having to fine-tune the model.

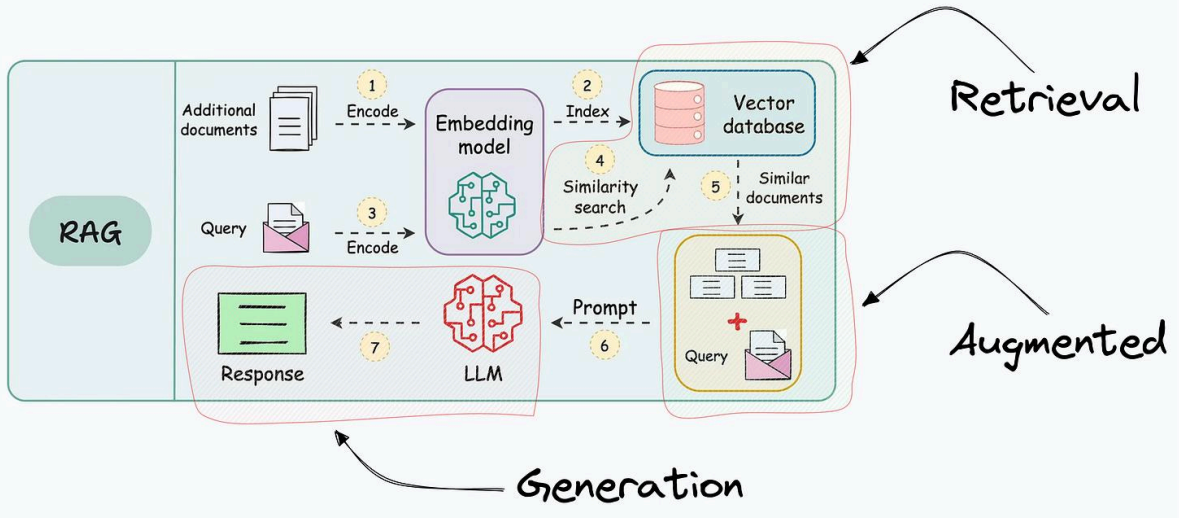
This is illustrated below:



There are 7 steps, which are also marked in the above visual:

- Step 1-2: Take additional data, and dump it in a vector database after embedding. (This is only done once. If the data is evolving, just keep dumping the embeddings into the vector database. There's no need to repeat this again for the entire data)
- Step 3: Use the same embedding model to embed the user query.
- Step 4-5: Find the nearest neighbors in the vector database to the embedded query.
- Step 6-7: Provide the original query and the retrieved documents (for more context) to the LLM to get a response.

In fact, even its name entirely justifies what we do with this technique:



- Retrieval: Accessing and retrieving information from a knowledge source, such as a database or memory.
- Augmented: Enhancing or enriching something, in this case, the text generation process, with additional information or context.
- Generation: The process of creating or producing something, in this context, generating text or language.

Of course, there are many problems with RAG too, such as:

- RAGs involve similarity matching between the query vector and the vectors of the additional documents. However, questions are structurally very different from answers.
- Typical RAG systems are well-suited only for lookup-based question-answering systems. For instance, we cannot build a RAG pipeline to summarize the additional data. The LLM never gets info about all the documents in its prompt because the similarity matching step only retrieves top matches.

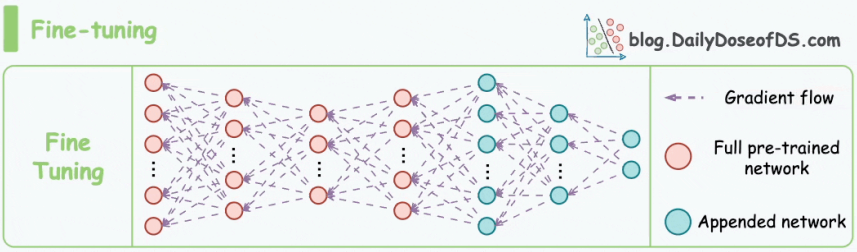
So, it's pretty clear that RAG has both pros and cons.

- We never have to fine-tune the model, which saves a lot of computing power.
- But this also limits the applicability to specific types of systems.

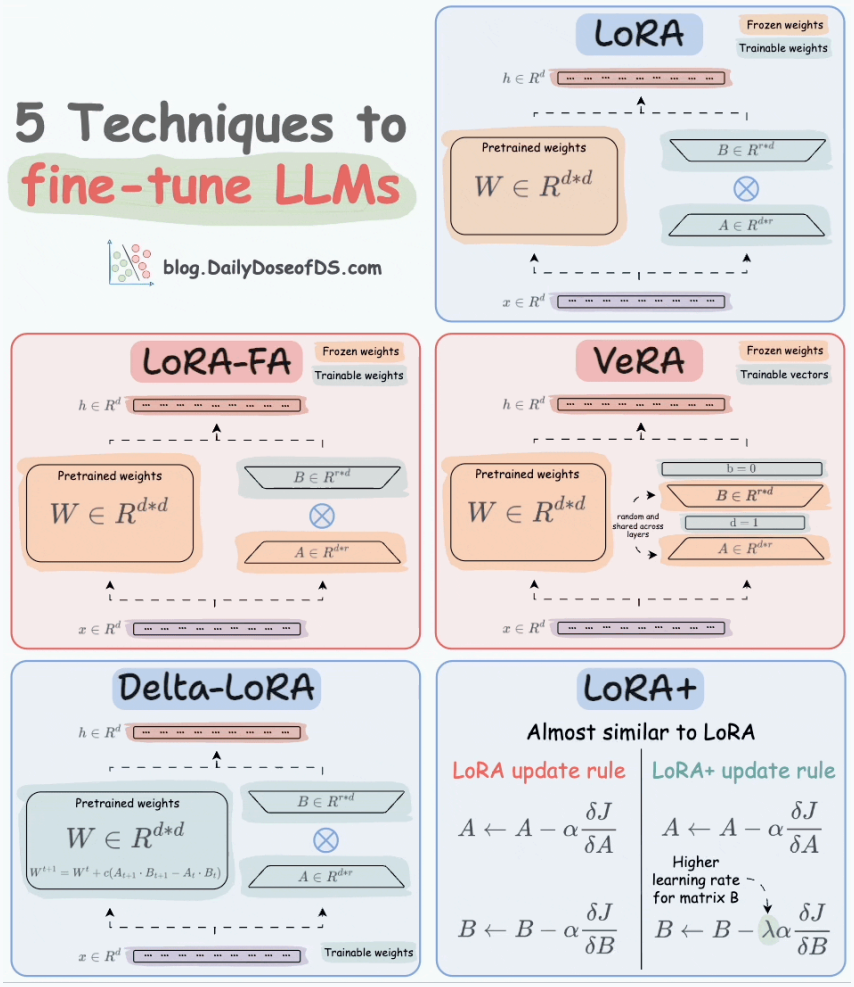
Let's continue the discussion on LLM fine-tuning in the next chapter.

5 LLM Fine-tuning Techniques

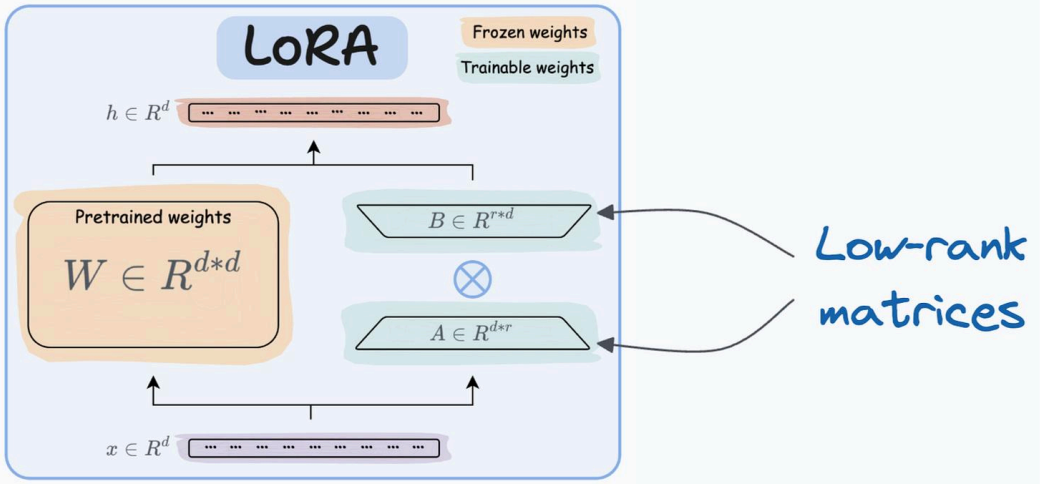
Traditional fine-tuning (depicted below) is infeasible with LLMs because these models have billions of parameters and are hundreds of GBs in size, and not everyone has access to such computing infrastructure.



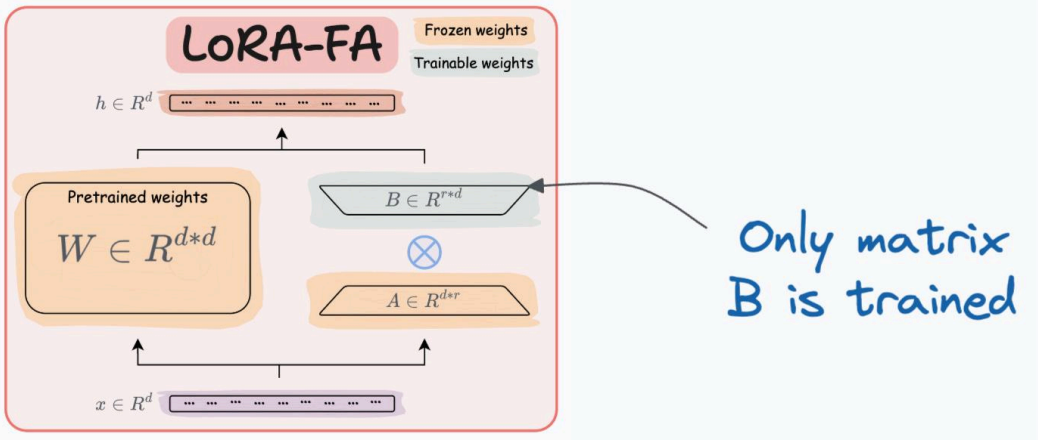
But today, we have many optimal ways to fine-tune LLMs, and five popular techniques are depicted below:



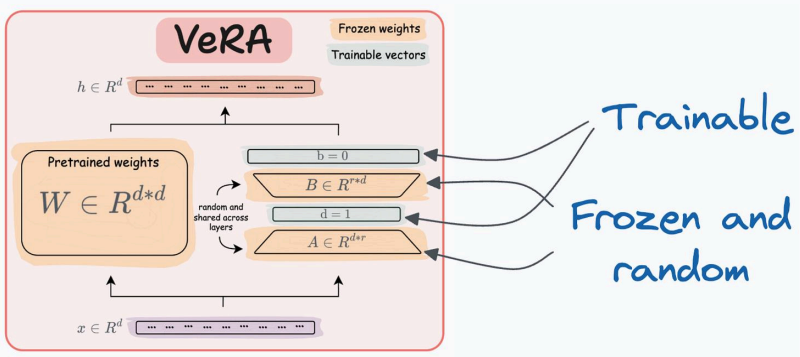
- LoRA: Add two low-rank matrices A and B alongside weight matrices, which contain the trainable parameters. Instead of fine-tuning W, adjust the updates in these low-rank matrices.



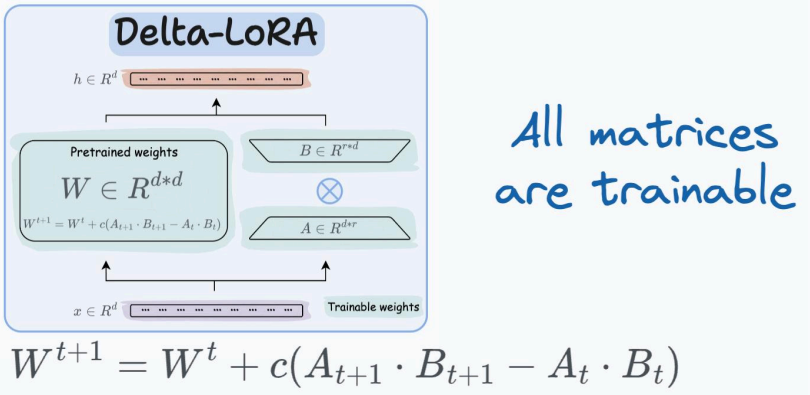
- LoRA-FA: While LoRA considerably decreases the total trainable parameters, it still requires substantial activation memory to update the low-rank weights. LoRA-FA (FA stands for Frozen-A) freezes the matrix A and only updates matrix B.



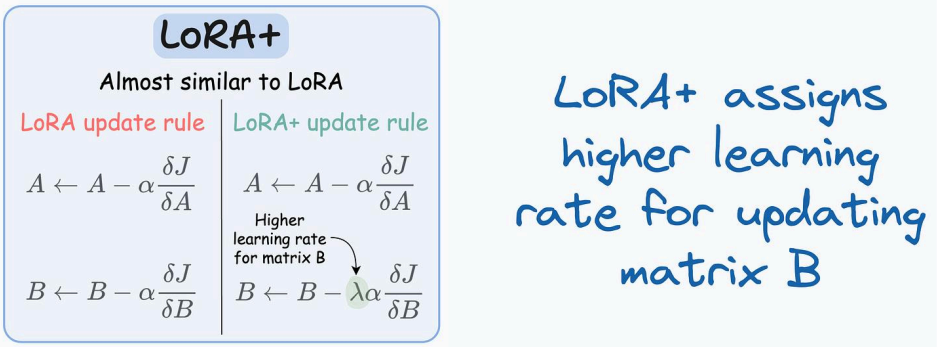
- VeRA: In LoRA, every layer has a different pair of low-rank matrices A and B, and both matrices are trained. In VeRA, however, matrices A and B are frozen, random, and shared across all model layers. VeRA focuses on learning small, layer-specific scaling vectors, denoted as b and d, which are the only trainable parameters in this setup.



- Delta-LoRA: Here, in addition to training low-rank matrices, the matrix W is also adjusted but not in the traditional way. Instead, the difference (or delta) between the product of the low-rank matrices A and B in two consecutive training steps is added to W:



- LoRA+: In LoRA, both matrices A and B are updated with the same learning rate. Authors found that setting a higher learning rate for matrix B results in more optimal convergence.

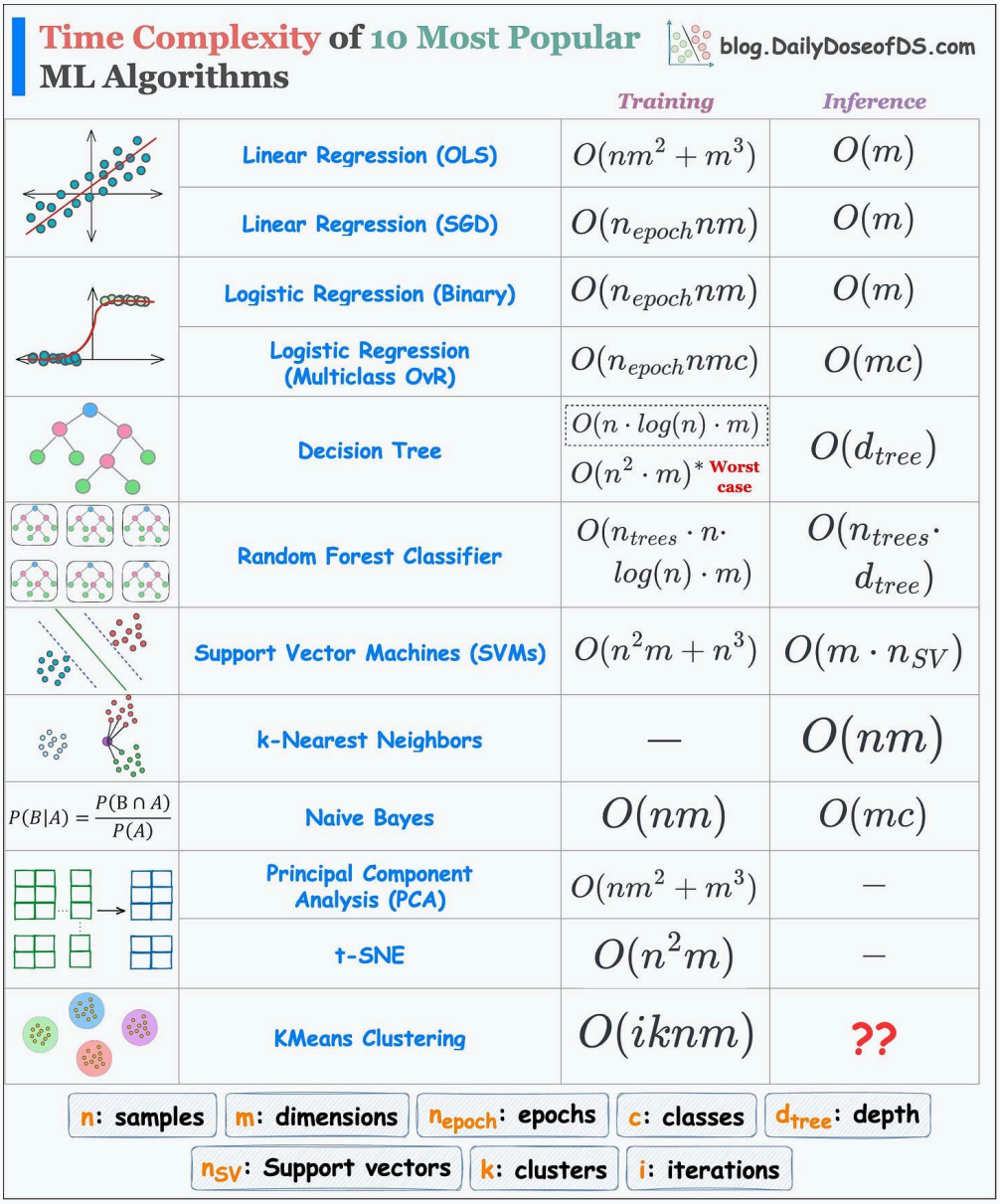


Classical ML

ML Fundamentals

Training and Inference Time Complexity of 10 ML Algorithms

Here's the run-time complexity of the 10 most popular ML algorithms.



	Training	Inference
Linear Regression (OLS)	$O(nm^2 + m^3)$	$O(m)$
Linear Regression (SGD)	$O(n_{epoch}nm)$	$O(m)$
Logistic Regression (Binary)	$O(n_{epoch}nm)$	$O(m)$
Logistic Regression (Multiclass OvR)	$O(n_{epoch}nmc)$	$O(mc)$
Decision Tree	$O(n \cdot log(n) \cdot m)$ $O(n^2 \cdot m)$* Worst case	$O(d_{tree})$
Random Forest Classifier	$O(n_{trees} \cdot n \cdot log(n) \cdot m)$	$O(n_{trees} \cdot d_{tree})$
Support Vector Machines (SVMs)	$O(n^2m + n^3)$	$O(m \cdot n_{SV})$
k-Nearest Neighbors	—	$O(nm)$
Naive Bayes	$O(nm)$	$O(mc)$
Principal Component Analysis (PCA)	$O(nm^2 + m^3)$	—
t-SNE	$O(n^2m)$	—
KMeans Clustering	$O(iknm)$	??

n: samples | m: dimensions | n_{epoch}: epochs | c: classes | d_{tree}: depth | n_{SV}: Support vectors | k: clusters | i: iterations

But why even care about run time? There are multiple reasons why I always care about run time and why you should too.

To begin, we know that everyone is a big fan of sklearn implementations. It literally takes just two (max three) lines of code to run any ML algorithm with sklearn. Yet, in my experience, due to this simplicity, most users often overlook:

- The core understanding of an algorithm.
- The data-specific conditions that allow us to use an algorithm.

For instance, you'll be up for a big surprise if you use SVM or t-SNE on a dataset with plenty of samples.

- SVM's run-time grows cubically with the total number of samples.
- t-SNE's run-time grows quadratically with the total number of samples.

Another advantage of figuring out the run-time is that it helps us understand how an algorithm works end-to-end. Of course, in the above table, I have made some assumptions here and there. For instance:

- In a random forest, all decision trees may have different depths. But here, I have assumed that they are equal.
- During inference in kNN, we first find the distance to all data points. This gives a list of distances of size n (total samples).
    - Then, we find the k-smallest distances from this list.
    - The run-time to determine the k-smallest values may depend on the implementation.
        - Sorting and selecting the k-smallest values will be $O(nlogn)$.
        - But if we use a priority queue, it will take $O(nlog(k))$.
- In t-SNE, there's a learning step. Since the major run-time comes from computing the pairwise similarities in the high-dimensional space, we have ignored that step.

Nonetheless, the table still accurately reflects the general run-time of each of these algorithms.

As an exercise, I would encourage you to derive these run-time complexities yourself. This activity will provide you so much confidence in algorithmic understanding.

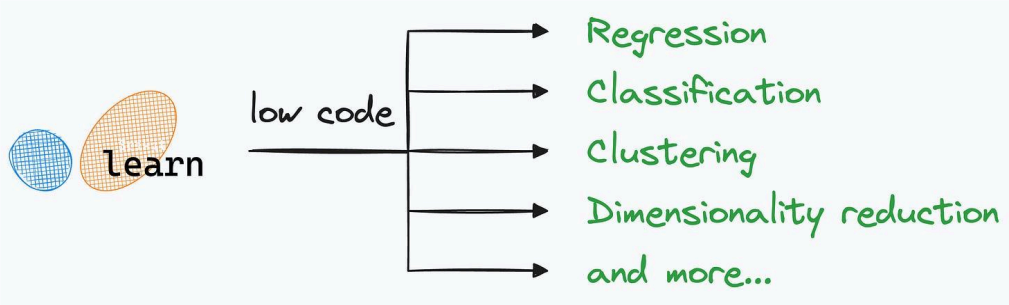
25 Most Important Mathematical Definitions in Data Science

"Is mathematical knowledge important in data science and machine learning?"

This is a question that so many people have, especially those who are just getting started.

Short answer: Yes, it's important, and here's why I say so.

See...these days, one can do "ML" without understanding any mathematical details of an algorithm. For instance (and thanks to sklearn, by the way):



- One can use any clustering algorithm in 2-3 lines of code.
- One can train classification models in 2-3 lines of code.
- And more.

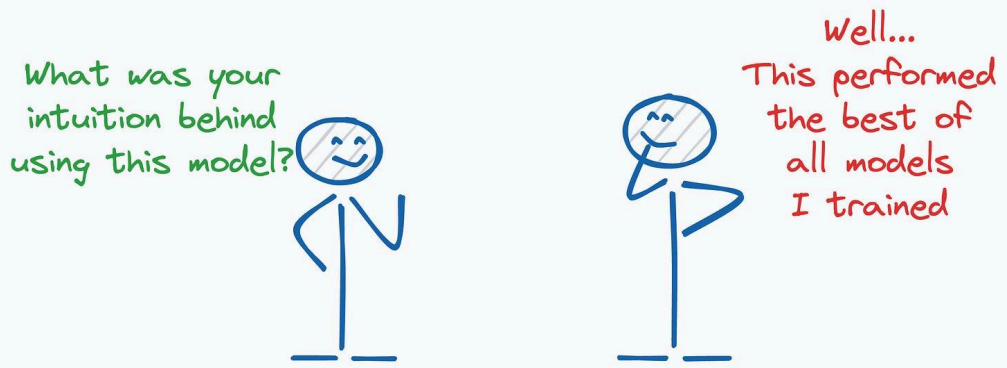
This is both good and bad:

- It's good because it saves us time.
- It's bad because this tempts us to ignore the underlying details.

In fact, I know many data scientists (mainly on the applied side) who do not entirely understand the mathematical details but can still build and deploy models.

Nothing wrong.

However, when I talk to them, I also see some disconnect between "What they are using" and "Why they are using it."



Due to a lack of understanding of the underlying details:

- They find it quite difficult to optimize their models.
- They struggle to identify potential areas of improvement.
- They take a longer time to debug when things don't work well.
- They do not fully understand the role of specific hyperparameters.
- They use any algorithm without estimating their time complexity first.

If it feels like you are one of them, it's okay. This problem can be solved.

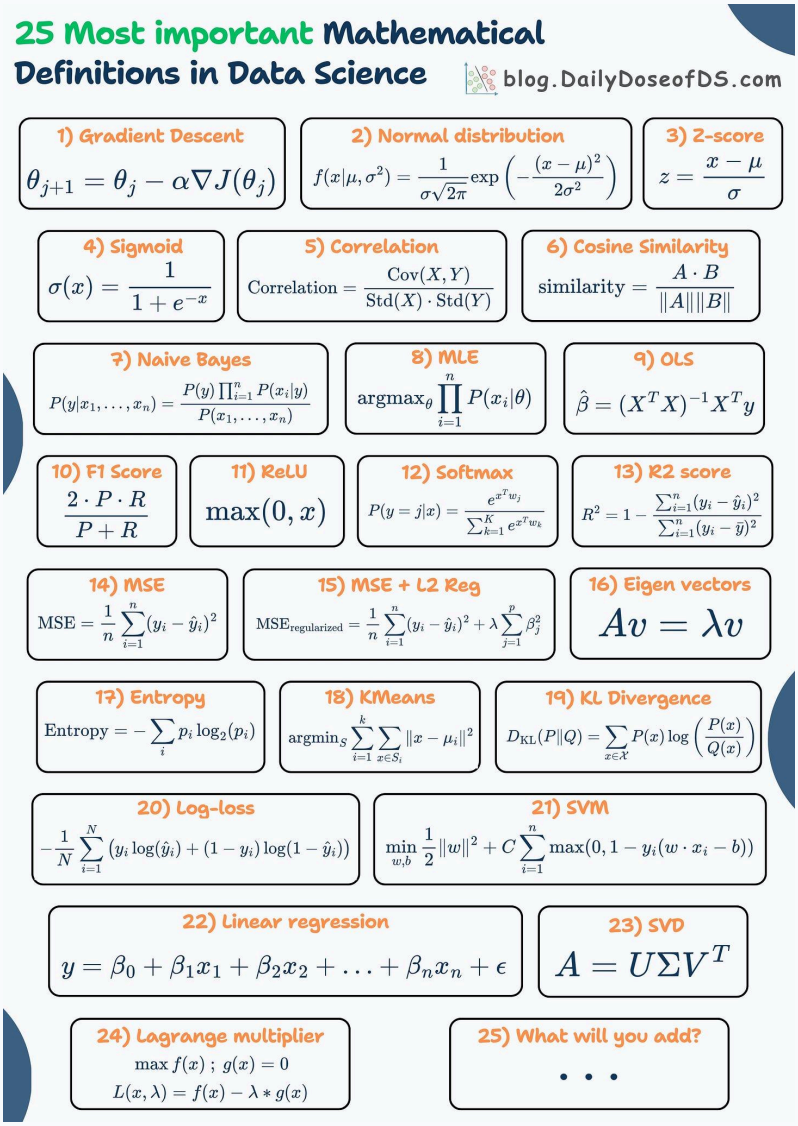
That said, if you genuinely aspire to excel in this field, building a curiosity for the underlying mathematical details holds exponential returns.

- Algorithmic awareness will give you confidence.
- It will decrease your time to build and iterate.

Gradually, you will go from a hit-and-trial approach to "I know what should work."

To help you take that first step, I prepared the following visual, which lists some of the most important mathematical formulations used in Data Science and Statistics (in no specific order).

Before reading ahead, look at them one by one and calculate how many of them do you already know:



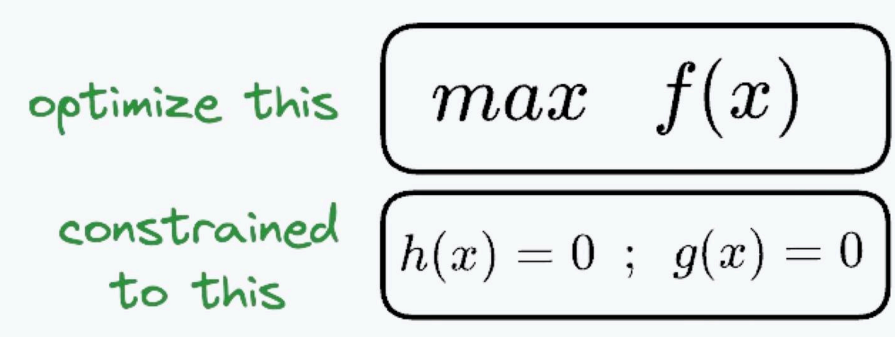
Some of the terms are pretty self-explanatory, so I won't go through each of them, like:

- Gradient Descent, Normal Distribution, Sigmoid, Correlation, Cosine similarity, Naive Bayes, F1 score, ReLU, Softmax, MSE, MSE + L2 regularization, KMeans, Linear regression, SVM, Log loss.

Here are the remaining terms:

- MLE (Maximum Likelihood Estimation): A method for estimating the parameters of a statistical model by maximizing the likelihood of the observed data. We covered it in the previous chapter.

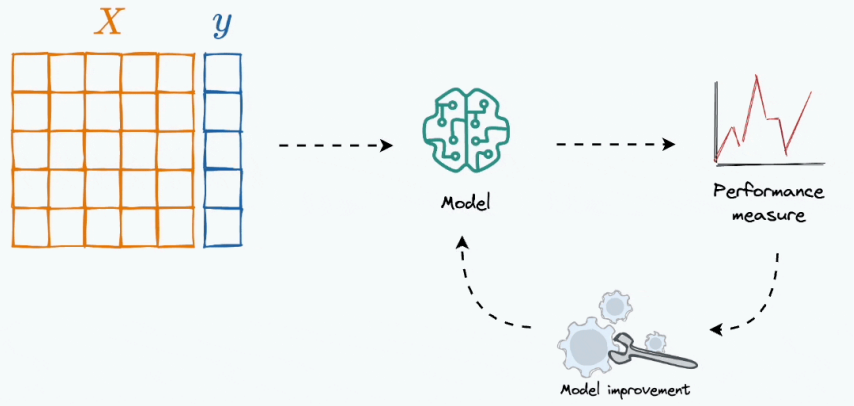
- Z-score: A standardized value that indicates how many standard deviations a data point is from the mean.
- Ordinary Least Squares: A closed-form solution for linear regression obtained using the MLE step mentioned above.
- Entropy: A measure of the uncertainty of a random variable.
- Eigen Vectors: The non-zero vectors that do not change their direction when a linear transformation is applied. It is widely used in dimensionality reduction techniques like PCA.
- R2 (R-squared): A statistical measure that represents the proportion of variance explained by a regression model.
- KL divergence: Assess how much information is lost when one distribution is used to approximate another distribution. It is used as a loss function in the t-SNE algorithm.
- SVD: A factorization technique that decomposes a matrix into three other matrices, often noted as U, Σ, and V. It is fundamental in linear algebra for applications like dimensionality reduction, noise reduction, and data compression.
- Lagrange multipliers: They are commonly used mathematical techniques to solve constrained optimization problems. For instance, consider an optimization problem with an objective function $f(x)$ and assume that the constraints are $g(x)=0$ and $h(x)=0$. Lagrange multipliers solve this.



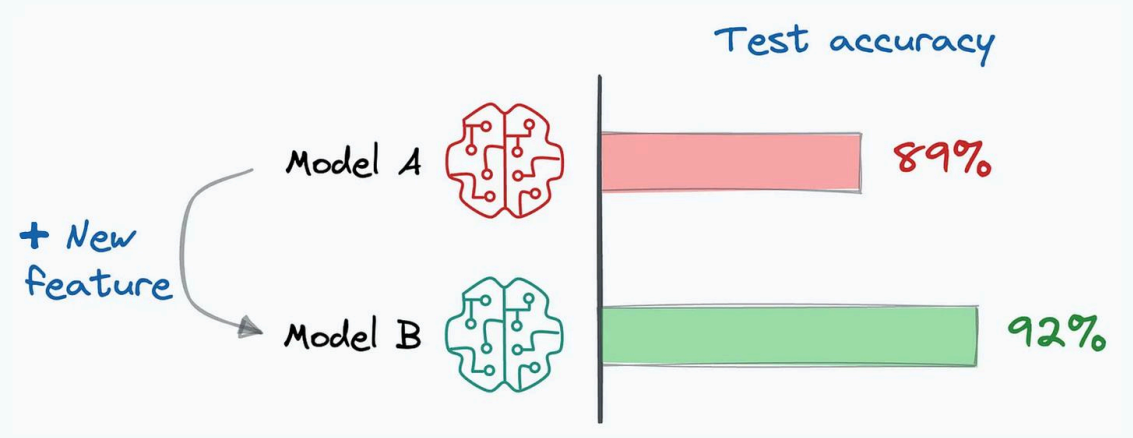
How to Reliably Improve Probabilistic Multiclass-classification Models

ML model building is typically an iterative process. Given some dataset:

- We train a model.
- We evaluate it.
- And we continue to improve it until we are satisfied with the performance.

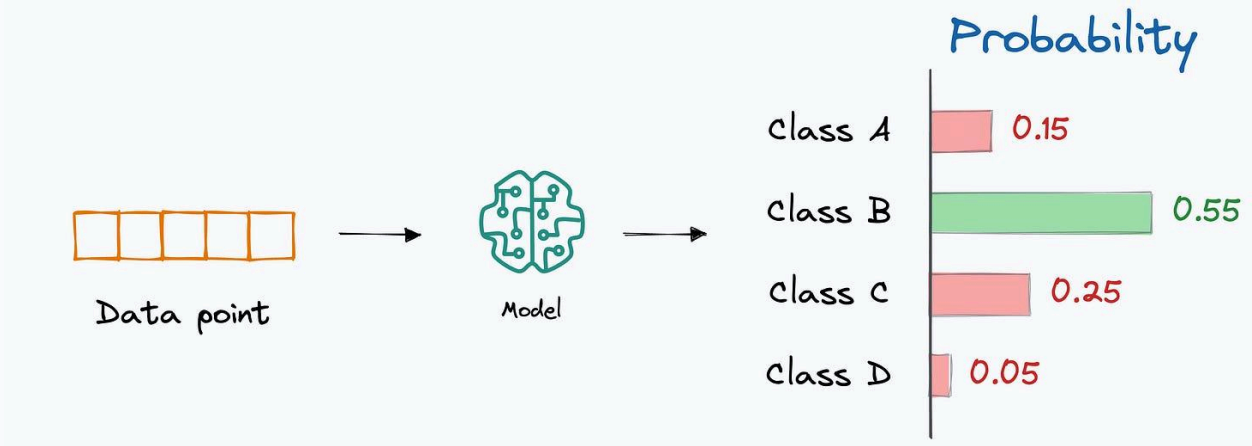


Here, the efficacy of any model improvement strategy (say, introducing a new feature) is determined using some sort of performance metric.



However, I have often observed that when improving probabilistic multiclass-classification models, this technique can be a bit deceptive when the efficacy is determined using "Accuracy."

Probabilistic multiclass-classification models are those models that output probabilities corresponding to each class, like neural networks.

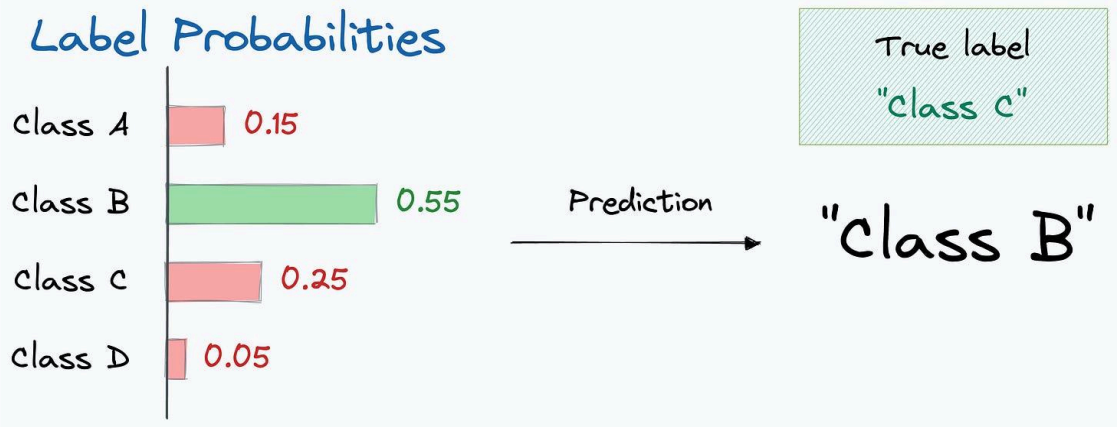


In other words, it is possible that we are actually making good progress in improving the model, but "Accuracy" is not reflecting that (yet).

Let's understand.

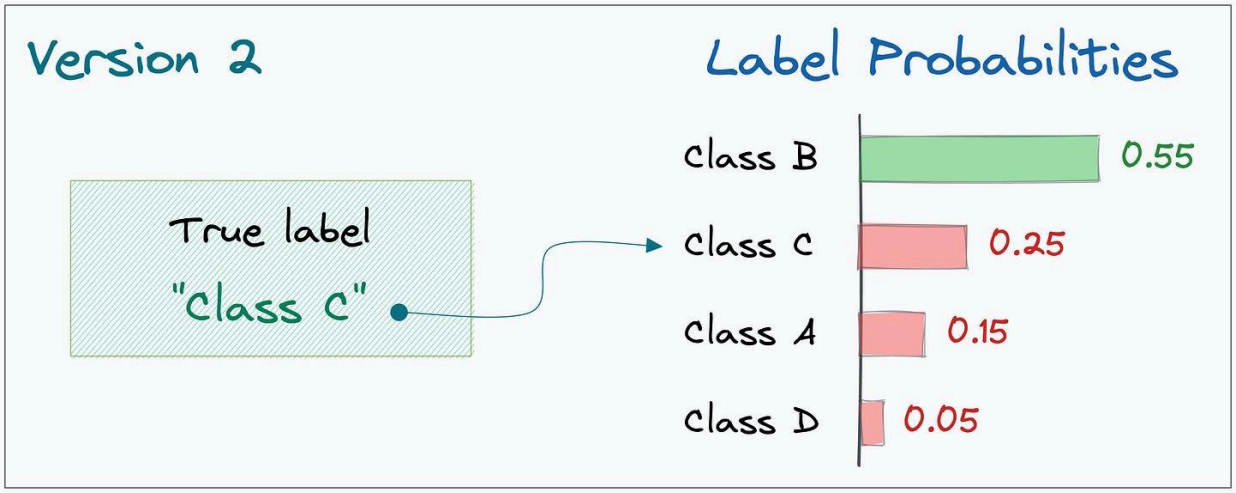
Pitfall of Accuracy

In probabilistic multiclass-classification models, Accuracy is determined using the output label that has the highest probability:

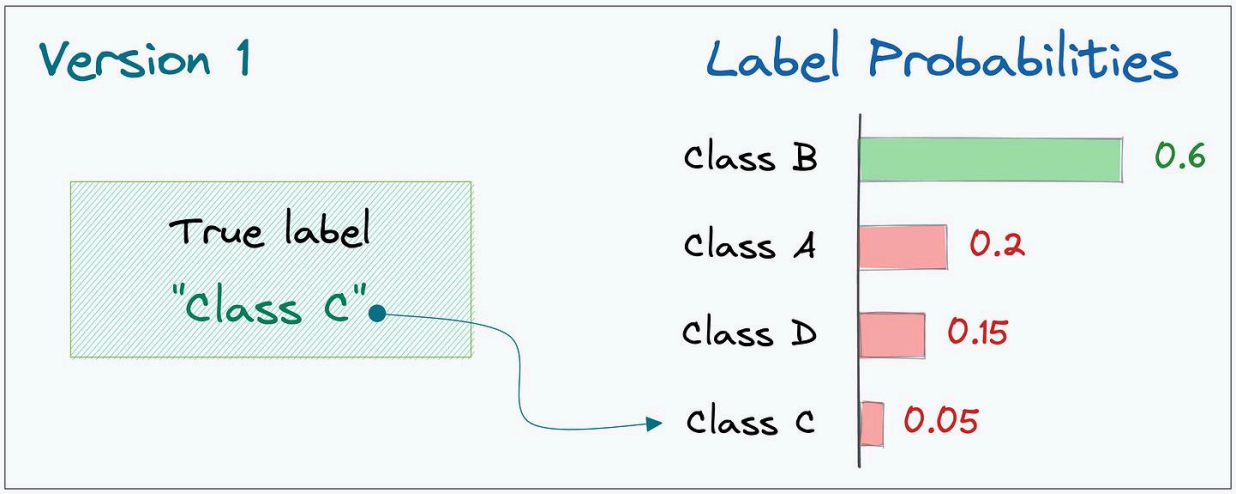


Now, it's possible that the actual label is not predicted with the highest probability by the model, but it's in the top "k" output labels.

For instance, in the image below, the actual label (Class C) is not the highest probability label, but it's at least in the top 2 predicted probabilities (Class B and Class C):



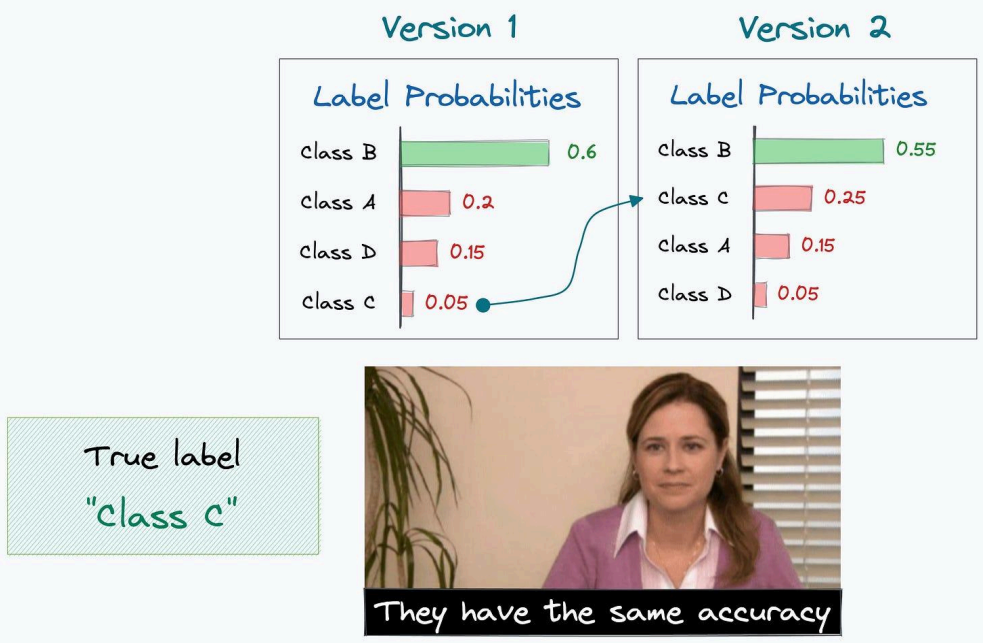
And what if in an earlier version of our model, the output probability of Class C was the lowest, as depicted below:



Now, of course, in both cases, the final prediction is incorrect.

However, while iterating from "Version 1" to "Version 2" using some model improvement techniques, we genuinely made good progress.

Nonetheless, Accuracy entirely discards this as it only cares about the highest probability label.



I hope you understand the problem here.

Solution

Whenever I am building and iteratively improving any probabilistic multiclass classification model, I always use the top-k accuracy score. As the name suggests, it computes whether the correct label is among the top k labels predicted probabilities or not.

As you may have already guessed, top-1 accuracy score is the traditional Accuracy score. This is a much better indicator to assess whether my model improvement efforts are translating into meaningful enhancements in predictive performance or not.

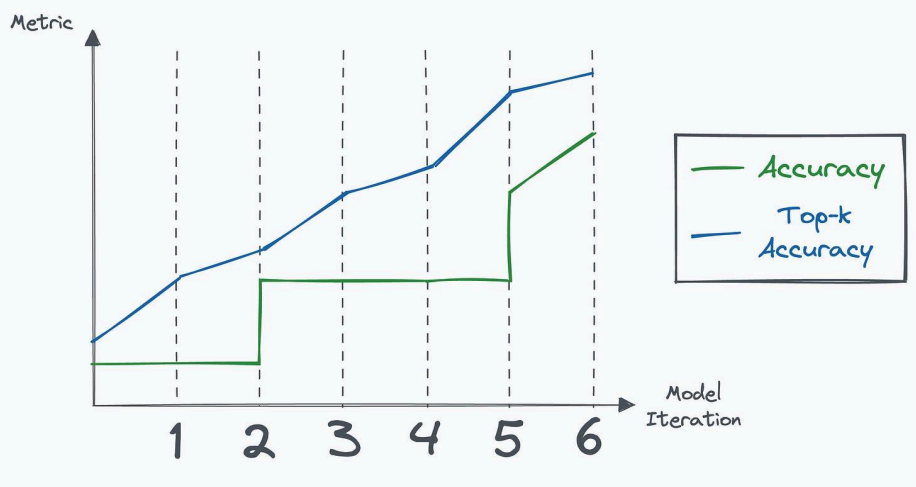
For instance, if the top-3 accuracy score goes from 75% to 90%, this totally suggests that whatever we did to improve the model was effective:

- Earlier, the correct prediction was in the top 3 labels only 75% of the time.
- But now, the correct prediction is in the top 3 labels 90% of the time.

As a result, one can effectively redirect their engineering efforts in the right direction. Of course, what I am saying should only be used to assess the model improvement efforts.

This is because true predictive power will inevitably be determined using traditional model accuracy. So make sure you are gradually progressing on the Accuracy front too.

Ideally, it is expected that "Top-k Accuracy" may continue to increase during model iterations, which reflects improvement in performance. Accuracy, however, may stay the same for a while, as depicted below:



Top-k accuracy score is also available in Sklearn:

sklearn.metrics.top_k_accuracy_score

sklearn.metrics.**top_k_accuracy_score**(*y_true*, *y_score*, ***, *k=2*, *normalize=True*, *sample_weight=None*, *labels=None*) [source]

Top-k Accuracy classification score.

This metric computes the number of times where the correct label is among the top `k` labels predicted (ranked by predicted scores). Note that the multilabel case isn't covered here.

Read more in the User Guide

Parameters:

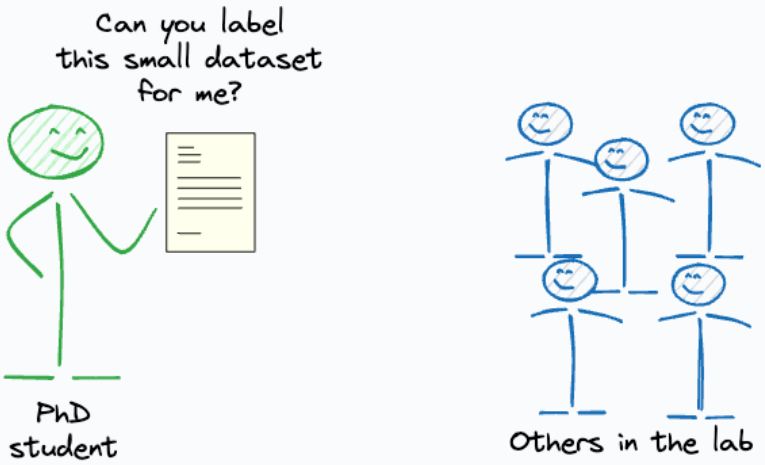
y_true : *array-like of shape (n_samples,)*
True labels.

y_score : *array-like of shape (n_samples,) or (n_samples, n_classes)*
Target scores. These can be either probability estimates or non-thresholded decision values (as returned

Your Entire Model Improvement Efforts Might Be Going in Vain

Back in 2019, I was working with an ML research group in Germany.

One day, a Ph.D. student came up to me (and others in the lab), handed over a small sample of the dataset he was working with, and requested us to label it, despite having true labels.

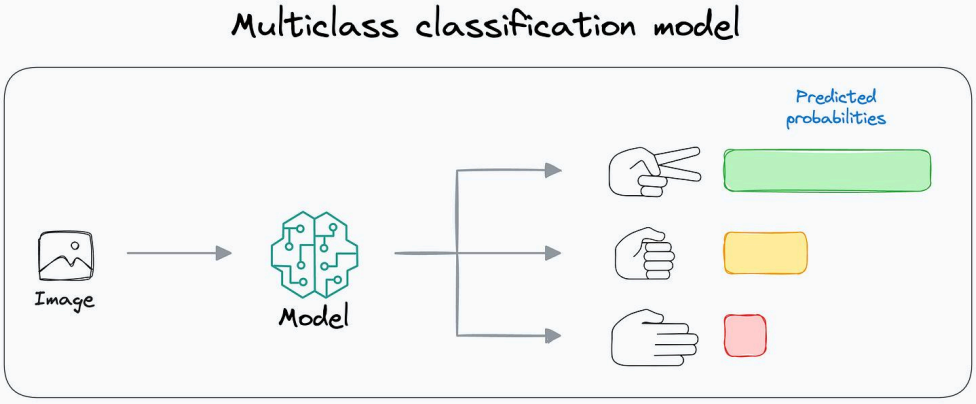


This made me curious about why gathering human labels was necessary for him when he already had ground truth labels available. So I asked.

What I learned that day changed my approach to incremental model improvement, and I am sure you will find this idea fascinating too.

Let me explain what I learned.

Consider we are building a multiclass classification model. Say it's a model that classifies an input image as a rock, paper, or scissors:



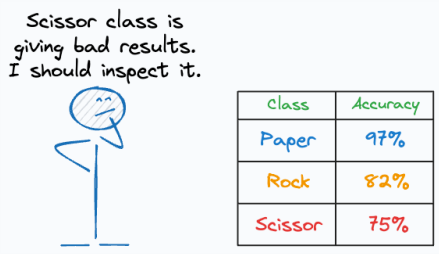
For simplicity, let's assume there's no class imbalance. Calculating the class-wise validation accuracies gives us the following results:

Class-wise accuracies

Class	Accuracy
Paper	97%
Rock	82%
Scissor	75%

Question: Which class would you most intuitively proceed to inspect further and improve the model on?

After looking at these results, most people believe that "Scissor" is the worst-performing class and should be inspected further.



But this might not be true. And this is precisely what that Ph.D. student wanted to verify by collecting human labels. Let's say that the human labels give us the following results:

Class-wise human accuracies

Class	Human Accuracy		Model Accuracy
Paper	98%	-1% →	97%
Rock	95%	-13% →	82%
Scissor	77%	-2% →	75%

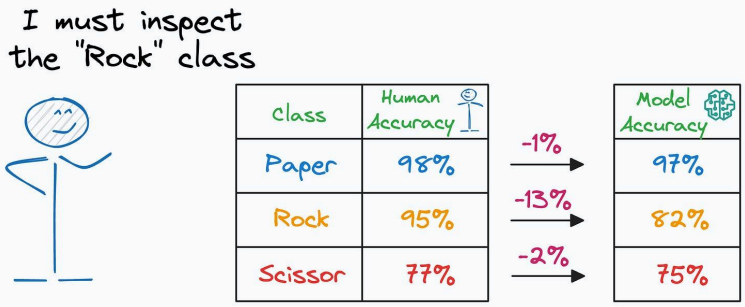
Based on this, do you still think the model performs the worst on the "Scissor" class?

No, right?

I mean, of course, the model has the least accuracy on the "Scissor" class, and I am not denying it. However, with more context, we notice that the model is doing a pretty good job classifying the "Scissor" class. This is because an average human is achieving just 2% higher accuracy in comparison to what our model is able to achieve.

However, the above results astonishingly reveal that it is the "Rock" class instead that demands more attention. The accuracy difference between an average human and the model is way too high (13%). Had we not known this, we would have continued to improve the "Scissor" class, when in reality, "Rock" requires more improvement.

Ever since I learned this technique, I have found it super helpful to determine my next steps for model improvement, if possible. I say "if possible" because I understand that many datasets are hard for humans to interpret and label. Nonetheless, if it is feasible to set up such a "human baseline," one can get so much clarity into how the model is performing.



As a result, one can effectively redirect their engineering efforts in the right direction.

Of course, I am not claiming that this will be universally useful in all use cases.

For instance, if the model is already performing better than the baseline, the model improvements from there on will have to be guided based on past results.

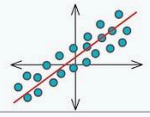
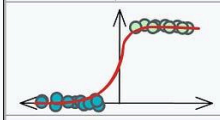
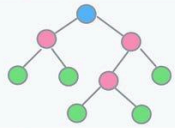
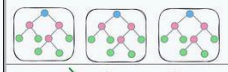
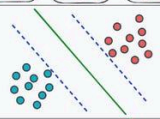
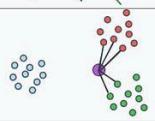
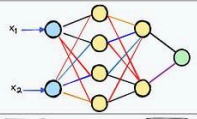
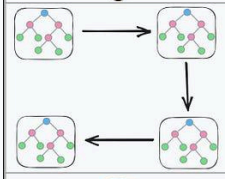
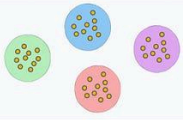
Yet, in such cases, surpassing a human baseline at least helps us validate that the model is doing better than what a human can do.

Loss Function of 16 ML Algos

The below visual depicts the most commonly used loss functions by various ML algorithms.

Loss functions of 16 Most Popular ML Algorithms

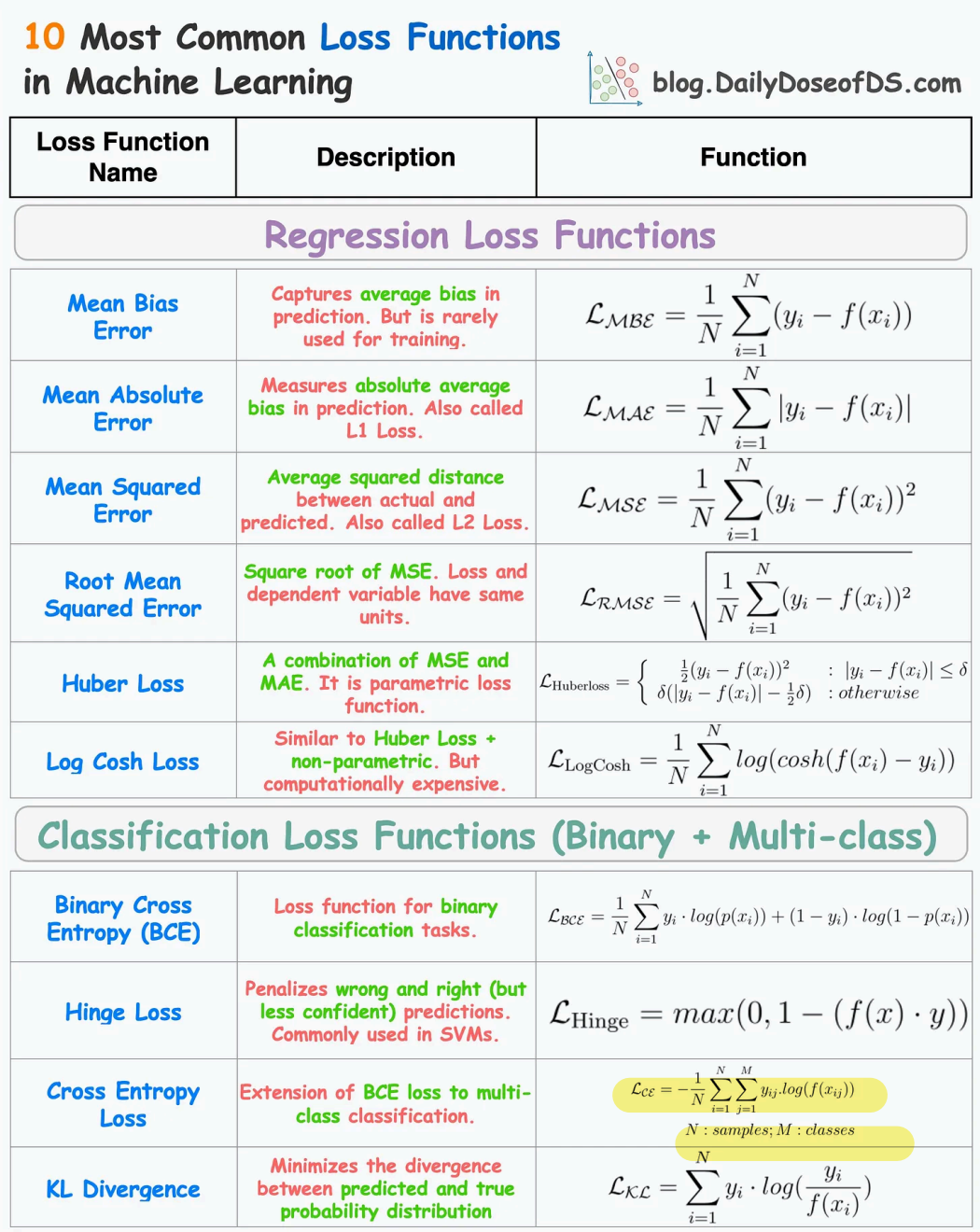
blog.DailyDoseofDS.com

	Linear Regression	Mean Squared Error
	Logistic Regression	Cross-Entropy Loss
	Decision Tree Classifier	Information Gain or Gini impurity
	Decision Tree Regressor	Mean Squared Error
	Random Forest Classifier	Information Gain or Gini impurity
	Random Forest Regressor	Mean Squared Error
	Support Vector Machines (SVMs)	Hinge Loss
	k-Nearest Neighbors	No loss function
$P(B\|A) = \frac{P(B \cap A)}{P(A)}$	Naive Bayes	No loss function
	Neural Networks	Regression: Mean Squared Error Classification: Cross-Entropy Loss
	AdaBoost	Exponential loss
	Gradient Boosting | LightGBM | CatBoost | XGBoost	Regression: Mean Squared Error Classification: Cross-Entropy Loss
	KMeans Clustering	??

*These are typically used loss functions. It does not mean they are used always in these algorithms.

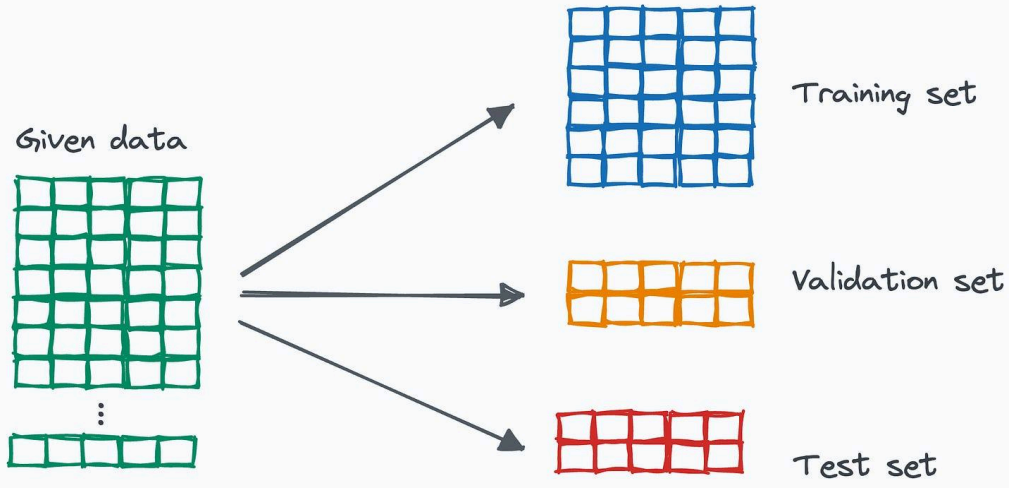
10 Most Common Loss Function

The below visual depicts some commonly used loss functions in regression and classification tasks.



How to Actually Use Train, Validation and Test Set

It is pretty conventional to split the given data into train, test, and validation sets.



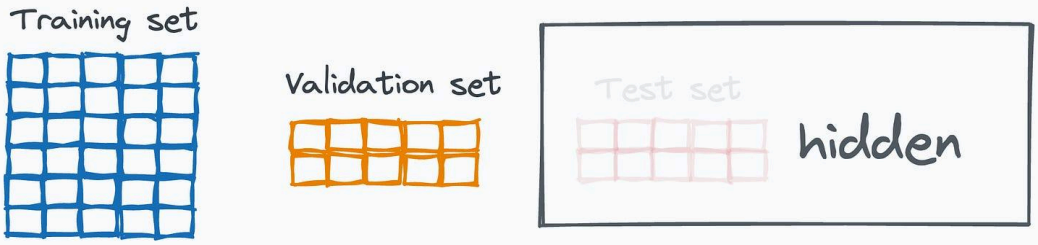
However, there are quite a few misconceptions about how they are meant to be used, especially the validation and test sets.

In this chapter, let's clear them up and see how to truly use train, validation, and test sets.

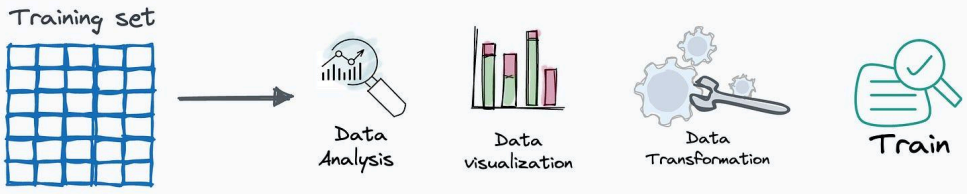
We begin by splitting the data into:

- Train
- Validation
- Test

At this point, just assume that the test data does not even exist. Forget about it instantly.



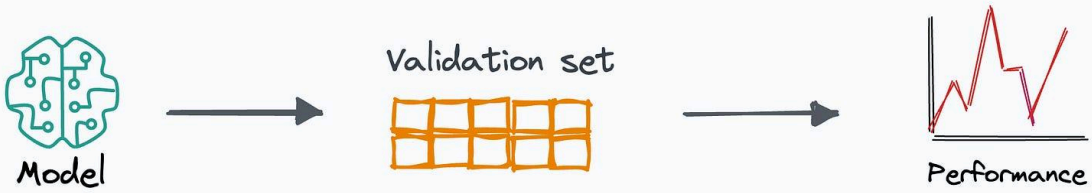
Begin with the train set. This is your whole world now.



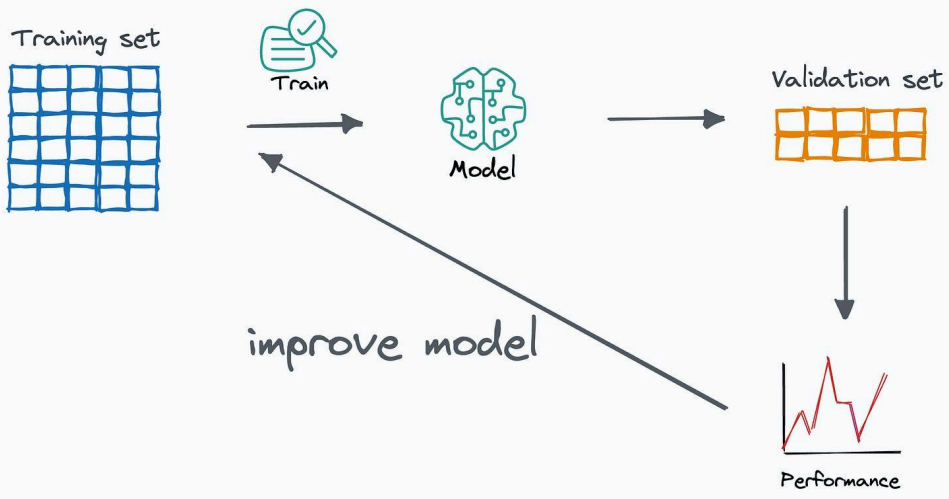
- You analyze it
- You transform it
- You use it to determine features
- You fit a model on it

After modeling, you would want to measure the model's performance on unseen data, wouldn't you?

Bring in the validation set now.



Based on validation performance, improve the model. Here's how you iteratively build your model:



- Train using a train set
- Evaluate it using the validation set
- Improve the model
- Evaluate again using the validation set
- Improve the model again
- and so on.

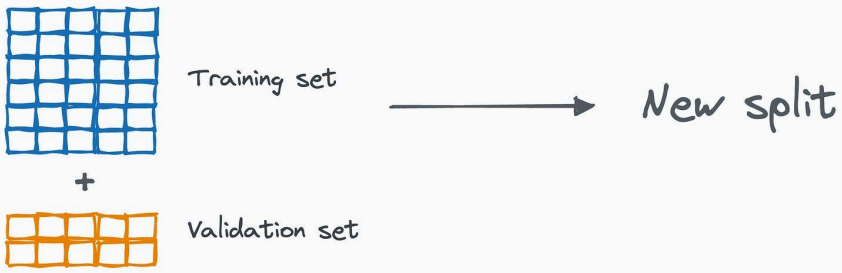
Until...

You reach a point where you start overfitting the validation set.

This indicates that you have exploited (or polluted) the validation set.

No worries.

Merge it with the train set and generate a new split of train and validation.



Note: Rely on cross-validation if needed, especially when you don't have much data. You may still use cross-validation if you have enough data. But it can be computationally intensive.

Now, if you are happy with the model's performance, evaluate it on test data.

✅ What you use a test set for:

- Get a final and unbiased review of the model.

❌ What you DON'T use a test set for:

- Analysis, decision-making, etc.

If the model is underperforming on the test set, no problem. Go back to the modeling stage and improve it.

BUT (and here's what most people do wrong)!

They use the same test set again. This is not allowed!

Think of it this way.

Your professor taught you in the classroom. All in-class lessons and examples are the train set.

The professor gave you take-home assignments, which acted like validation sets.

You got some wrong and some right. Based on this, you adjusted your topic fundamentals, i.e., improved the model.

Now, if you keep solving the same take-home assignment repeatedly, you will eventually overfit it, won't you?

That is why we bring in a new validation set after some iterations.

The final exam day paper is your test set. If you do well, awesome!

But if you fail, the professor cannot give you the exact exam paper next time, can they? This is because you know what's inside.

Of course, by evaluating a model on the test set, the model never gets to "know" the precise examples inside that set. But the issue is that the test set has been exposed now.

Your previous evaluation will inevitably influence any further evaluations on that specific test set. That is why you must always use a specific test set only ONCE.

Once you do, merge it with the train and validation set and generate an entirely new split.

Repeat.

And that is how you use train, validation, and test sets in machine learning.

5 Cross Validation Techniques

Tuning and validating machine learning models on a single validation set can be misleading and sometimes yield overly optimistic results.

This can occur due to a lucky random split of data, which results in a model that performs exceptionally well on the validation set but poorly on new, unseen data.

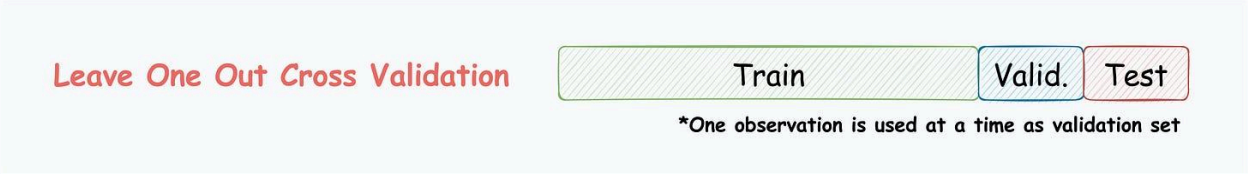
That is why we often use cross validation instead of simple single-set validation.

Cross validation involves repeatedly partitioning the available data into subsets, training the model on a few subsets, and validating on the remaining subsets.

The main advantage of cross validation is that it provides a more robust and unbiased estimate of model performance compared to the traditional validation method.

Below are five of the most commonly used and must-know cross validation techniques.

Leave-One-Out Cross Validation

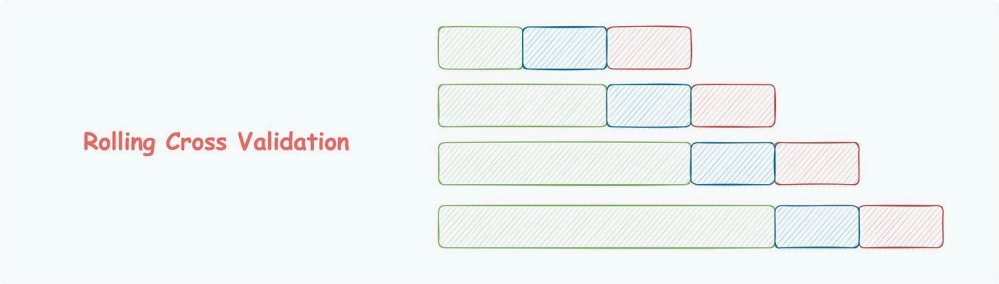


- Leave one data point for validation.
- Train the model on the remaining data points.
- Repeat for all points.
- Of course, as you may have guessed, this is practically infeasible when you have many data points. This is because number of models is equal to number of data points.
- We can extend this to Leave-p-Out Cross Validation, where, in each iteration, p observations are reserved for validation, and the rest are used for training.

K-Fold Cross Validation

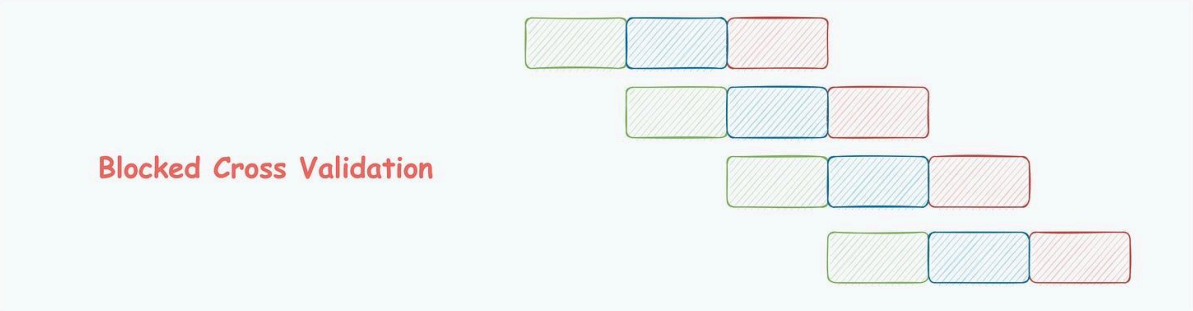
- Split data into k equally-sized subsets.
- Select one subset for validation.
- Train the model on the remaining subsets.
- Repeat for all subsets.

Rolling Cross Validation



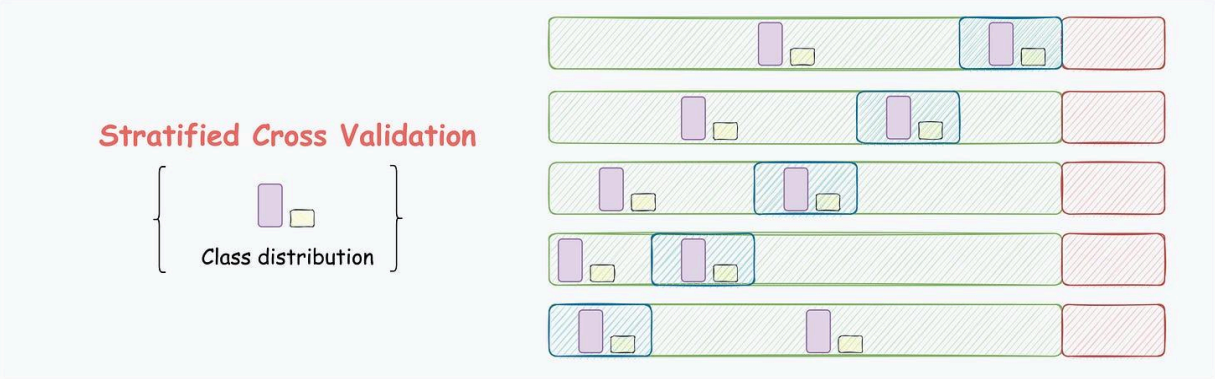
- Mostly used for data with temporal structure.
- Data splitting respects the temporal order, using a fixed-size training window.
- The model is evaluated on the subsequent window.

Blocked Cross Validation



- Another common technique for time-series data.
- In contrast to rolling cross validation, the slice of data is intentionally kept short if the variance does not change appreciably from one window to the next.
- This also saves computation over rolling cross validation.

Stratified Cross Validation



- The above-discussed techniques may not work for imbalanced datasets. Stratified cross validation is mainly used for preserving the class distribution.
- The partitioning ensures that the class distribution is preserved.

Let's continue our discussion on cross validation in the next chapter.

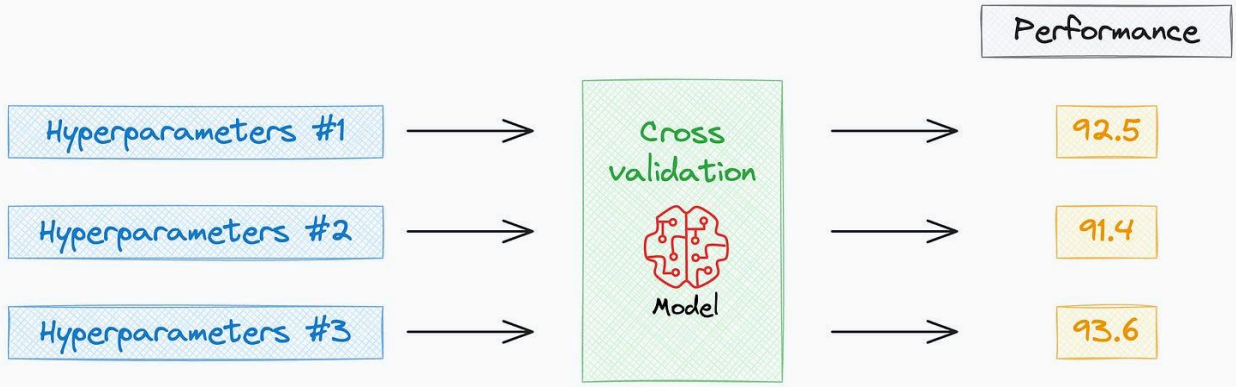
What To Do After Cross Validation?

Let me ask you a question.

But before I do that, I need to borrow your imagination for just a moment.

Imagine you are building some supervised machine learning model. You are using cross-validation to determine an optimal set of hyperparameters.

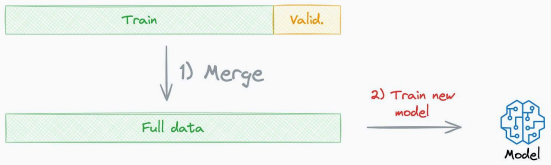
Essentially, every hyperparameter configuration corresponds to a cross-validation performance:



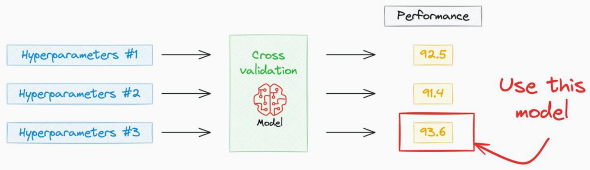
After obtaining the best hyperparameters, we need to finalize a model (say for production); otherwise, what is the point of all this hassle? Now, here's the question:

After obtaining the optimal hyperparameters, what would you be more inclined to do:

1) Retrain the model again on the entire data (train + validation + test) with the optimal hyperparameters? If we do this, remember that we can't reliably validate this new model as there is no unseen data left.



2) Just proceed with the best-performing model based on cross-validation performance itself. If we do this, remember that we are leaving out important data, which we could have trained our model with.



What would you do?

Recommended path

My strong preference has almost always been "retraining a new model with entire data."

There are, of course, some considerations to keep in mind, which I have learned through the models I have built and deployed. That said, in most cases, retraining is the ideal way to proceed.

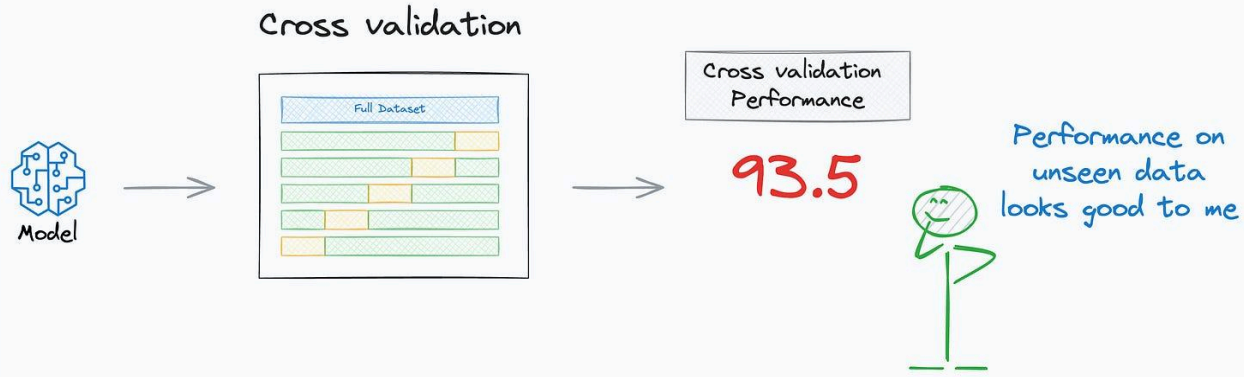
Let me explain.

Why retrain the model?

We would want to retrain a new model because, in a way, we are already satisfied with the cross-validation performance, which, by its very nature, is an out-of-fold metric.

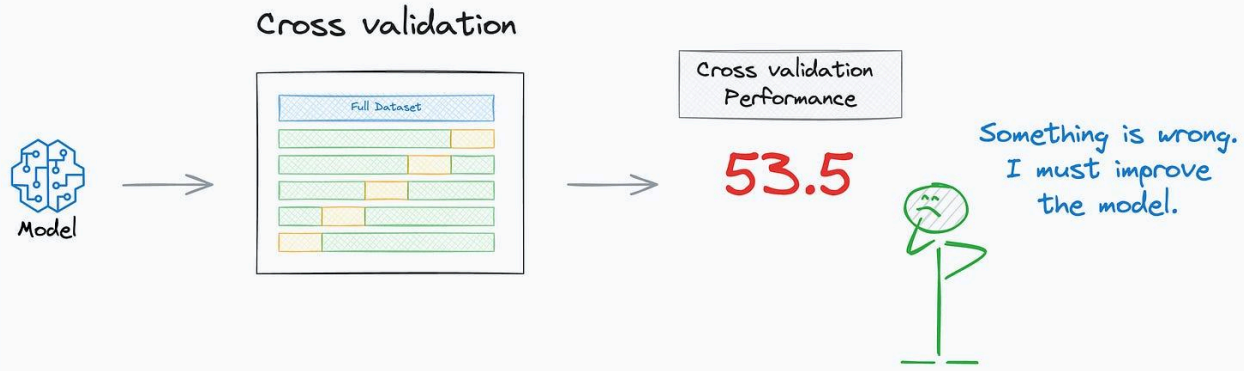
An out-of-fold data is data that has not been seen by the model during the training. An out-of-fold metric is the performance on that data.

In other words, we already believe that the model aligns with how we expect it to perform on unseen data.



Thus, incorporating this unseen validation set in the training data and retraining the model will MOST LIKELY have NO effect on its performance on unseen data after deployment (assuming a sudden covariate shift hasn't kicked in, which is a different issue altogether).

If, however, we were not satisfied with the cross-validation performance itself, we wouldn't even be thinking about finalizing a model in the first place.

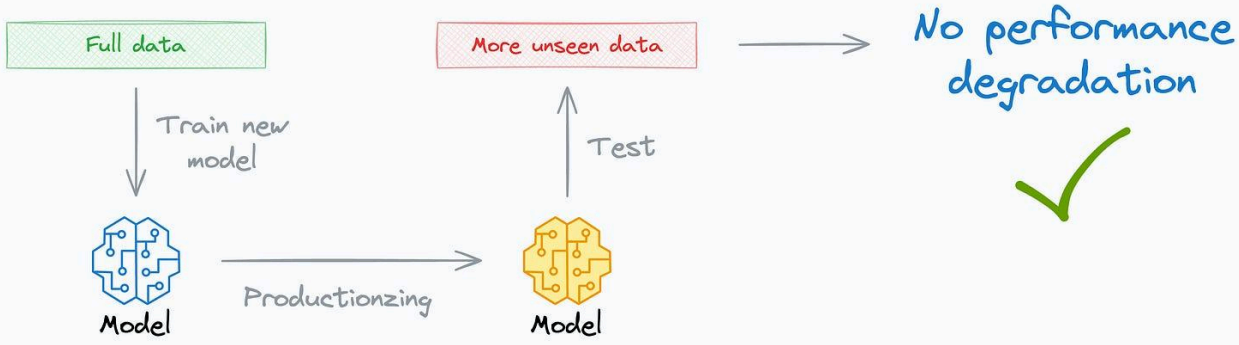


Instead, we would be thinking about ways to improve the model by working on feature engineering, trying new hyperparameters, experimenting with different models, and more.

The reasoning makes intuitive sense as well.

It's hard for me to recall any instance where retraining did something disastrously bad to the overall model.

In fact, I vividly remember one instance wherein, while I was productionizing the model (it took me a couple of days after retraining), the team had gathered some more labeled data.

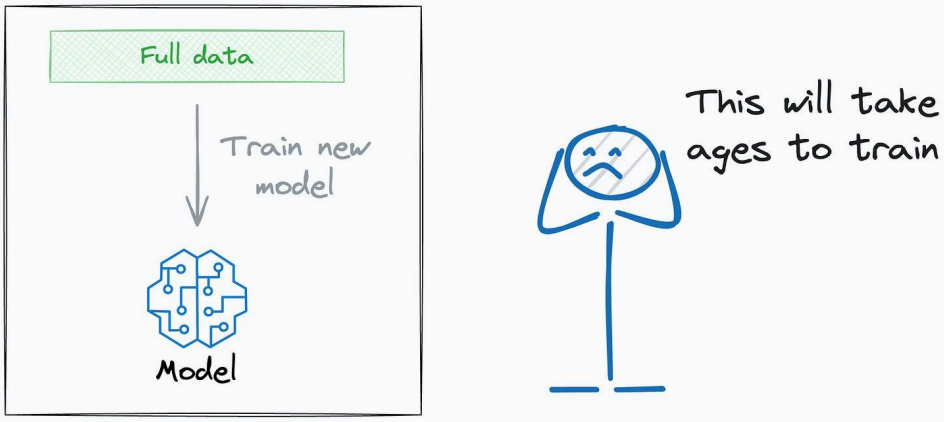


The model didn't show any performance degradation when I evaluated it (just to double-check). As an added benefit, this also helped ensure that I had made no errors while productionizing my model.

Some considerations

Here, please note that it's not a rule that you must always retrain a new model.

The field itself and the tasks one can solve are pretty diverse, so one must be open-minded while solving the problem at hand. One of the reasons I wouldn't want to retrain a new model is that it takes days or weeks to train the model.



In fact, even if we retrain a new model, there are MANY business situations in which stakes are just too high.

Thus, one can never afford to be negligent about deploying a model without re-evaluating it — transactional fraud, for instance.

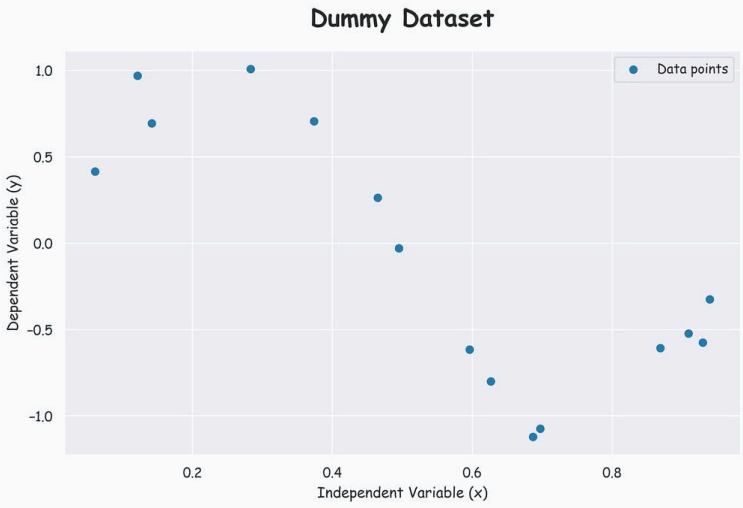
In such cases, I have seen that while a team works on productionizing the model, data engineers gather some more data in the meantime.

Before deploying, the team would do some final checks on that dataset.

The newly gathered data is then considered in the subsequent iterations of model improvements.

Double Descent vs. Bias-Variance Trade-off

It is well-known that as the number of model parameters increases, we typically overfit the data more and more. For instance, consider fitting a polynomial regression model trained on this dummy dataset below:



Dummy regression dataset

In case you don't know, this is called a polynomial regression model:

Polynomial regression model

$$y = \theta_0 + \theta_1 * x^1 + \theta_2 * x^2 + \cdots + \theta_m * x^m$$

It is expected that as we'll increase the degree (m) and train the polynomial regression model:

- The training loss will get closer and closer to zero.
- The test (or validation) loss will first reduce and then get bigger and bigger.

This is because, with a higher degree, the model will find it easier to contort its regression fit through each training data point, which makes sense.

In fact, this is also evident from the following loss plot:

But notice what happens when we continue to increase the degree (m):

That's strange, right?

Why does the test loss increase to a certain point but then decrease?

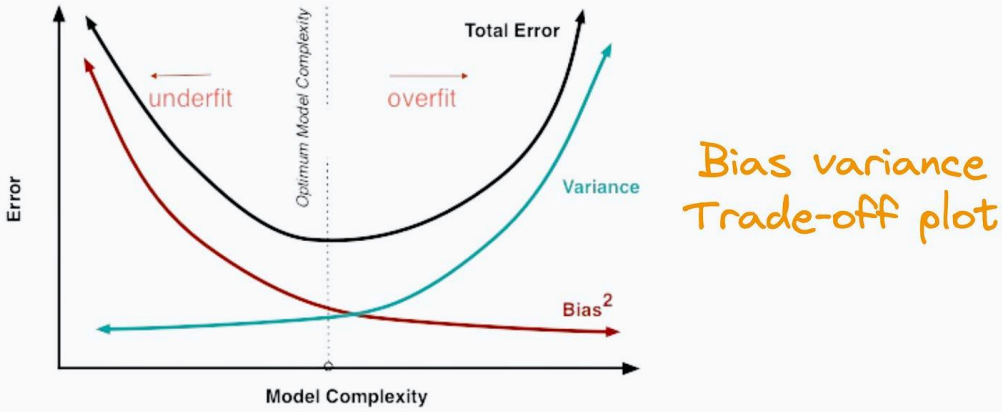
This was not expected, was it?

Well...what you are seeing is called the "double descent phenomenon," which is quite commonly observed in many ML models, especially deep learning models.

It shows that, counterintuitively, increasing the model complexity beyond the point of interpolation can improve generalization performance.

In fact, this whole idea is deeply rooted to why LLMs, although massively big (billions or even trillions of parameters), can still generalize pretty well.

And it's hard to accept it because this phenomenon directly challenges the traditional bias-variance trade-off we learn in any introductory ML class:



Putting it another way, training very large models, even with more parameters than training data points, can still generalize well.

To the best of my knowledge, this is still an open question, and it isn't entirely clear why neural networks exhibit this behavior.

There are some theories around regularization, however, such as this one:

It could be that the model applies some sort of implicit regularization, with which, it can precisely focus on an apt number of parameters for generalization.

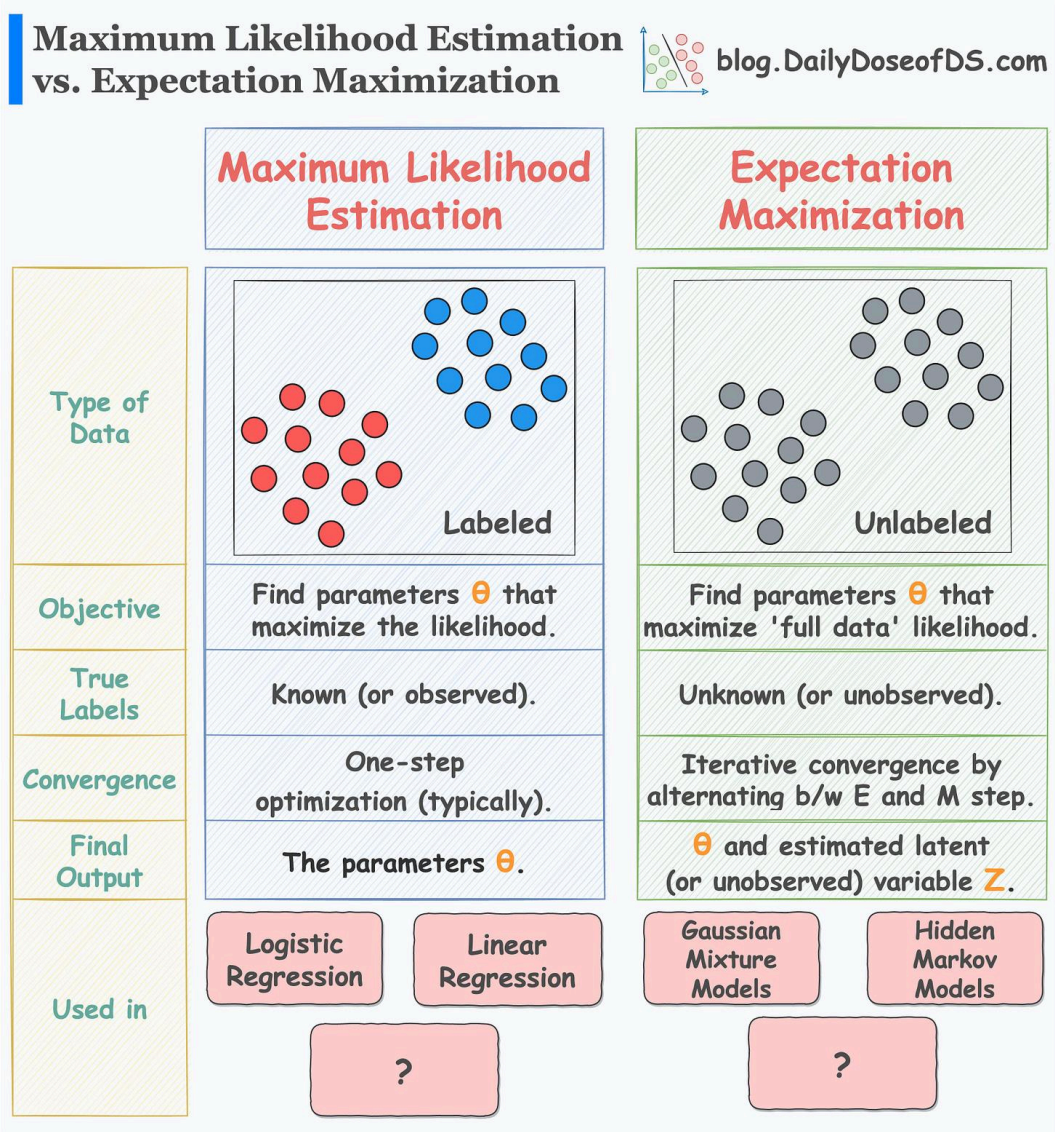
But to be honest, nothing is clear yet.

Statistical Foundations

MLE vs. EM — What's the Difference?

Maximum likelihood estimation (MLE) and expectation maximization (EM) are two popular techniques to determine the parameters of statistical models.

Due to its applicability in MANY statistical models, I have seen it being asked in plenty of data science interviews as well, especially the distinction between the two. The following visual summarizes how they work:

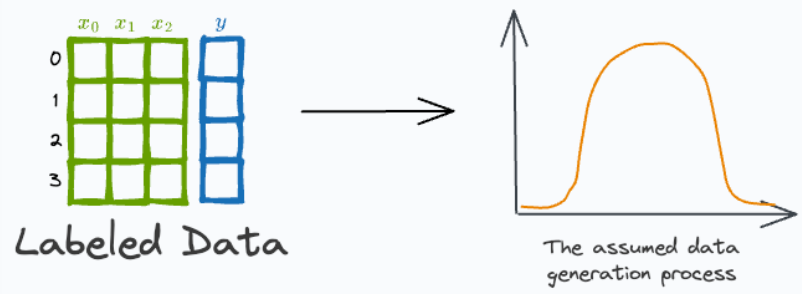


Maximum likelihood estimation (MLE)

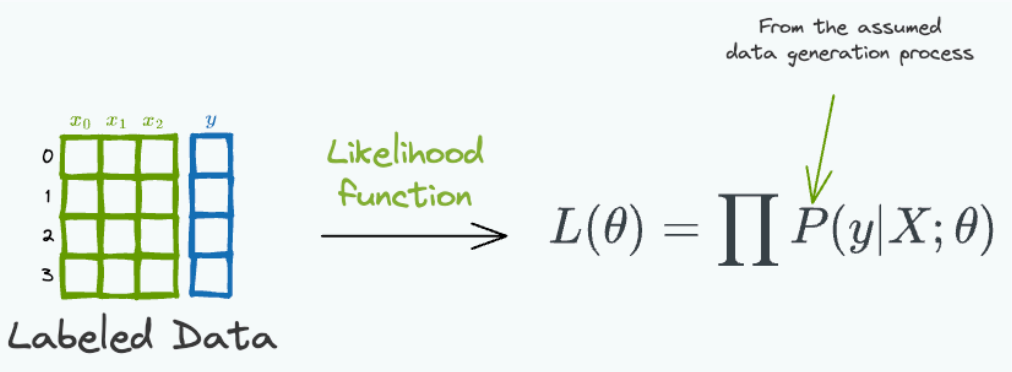
MLE starts with a labeled dataset and aims to determine the parameters of the statistical model we are trying to fit.

The process is pretty simple and straightforward. In MLE, we:

- Start by assuming a data generation process. Simply put, this data generation process reflects our belief about the distribution of the output label (y), given the input (X).



- Next, we define the likelihood of observing the data. As each observation is independent, the likelihood of observing the entire data is the same as the product of observing individual observations:

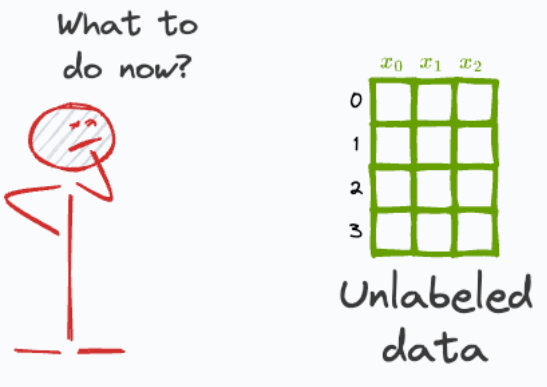


- The likelihood function above depends on parameter values (θ). Our objective is to determine those specific parameter values that maximize the likelihood function. We do this as follows:

This gives our parameter estimates that would have most likely generated the given data.

That was pretty simple, wasn't it?

But what do we do if we don't have true labels? We still want to estimate the parameters, don't we?

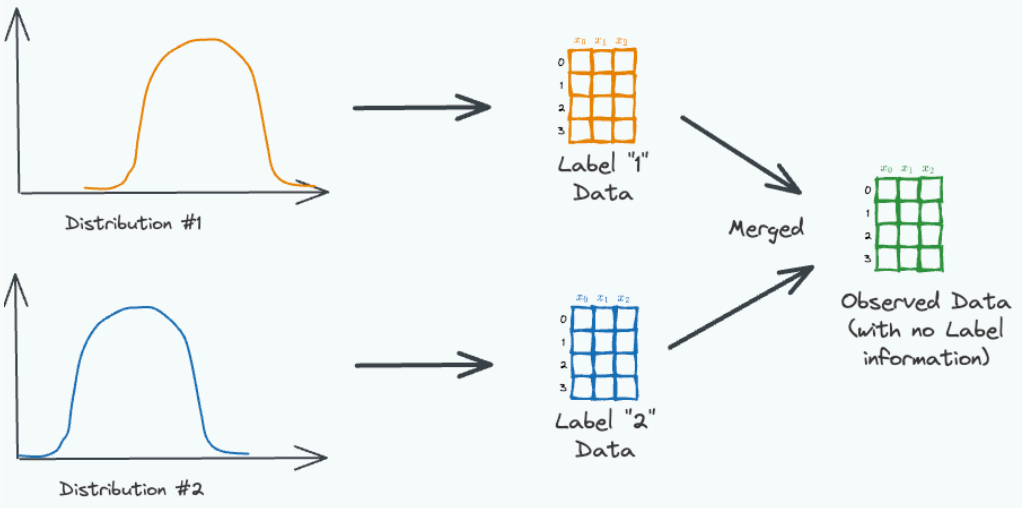


MLE, as you may have guessed, will not be applicable. The true label (y), being unobserved, makes it impossible to define a likelihood function like we did earlier.

In such cases, advanced techniques like expectation maximization are pretty helpful.

Expectation maximization (EM)

EM is an iterative optimization technique to estimate the parameters of statistical models. It is particularly useful when we have an unobserved (or hidden) label. One example situation could be as follows:



As depicted above, we assume that the data was generated from multiple distributions (a mixture). However, the observed/complete data does not contain that information. In other words, the observed dataset does not have information about whether a specific row was generated from distribution 1 or distribution 2.

Had it contained the label (y) information, we would have already used MLE.

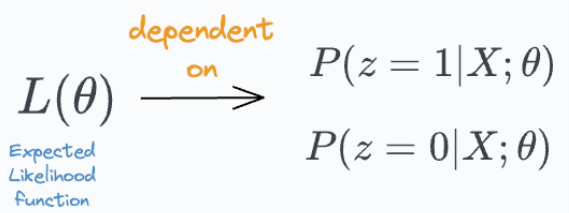
EM helps us with parameter estimates of such datasets. The core idea behind EM is as follows:

- Make a guess about the initial parameters (θ).
- Expectation (E) step: Compute the posterior probabilities of the unobserved label (let's call it 'z') using the above parameters.

$$P(z = 1|X; \theta)$$

$$P(z = 0|X; \theta)$$

    - Here, 'z' is also called a latent variable, which means hidden or unobserved.
    - Relating it to our case, we know that the true label exists in nature. But we don't know what it is.
    - Thus, we replace it with a latent variable 'z' and estimate its posterior probabilities using the guessed parameters.
- Given that we now have a proxy (not precise, though) for the true label, we can define an "expected likelihood" function. Thus, we use the above posterior probabilities to do so:



- Maximization (M) step: So now we have a likelihood function to work with. Maximizing it with respect to the parameters will give us a new estimate for the parameters (θ`).
- Next, we use the updated parameters (θ`) to recompute the posterior probabilities we defined in the expectation step.

$$P(z=1|X;\theta')$$
$$P(z=0|X;\theta')$$

- We will update the likelihood function (L) using the new posterior probabilities.
- Again, maximizing it will give us a new estimate for the parameters (θ).
- And this process goes on and on until convergence.

The point is that in expectation maximization, we repeatedly iterate between the E and the M steps until the parameters converge. A good thing about EM is that it always converges. Yet, at times, it might converge to a local extrema.

MLE vs. EM is a popular question asked in many data science interviews.

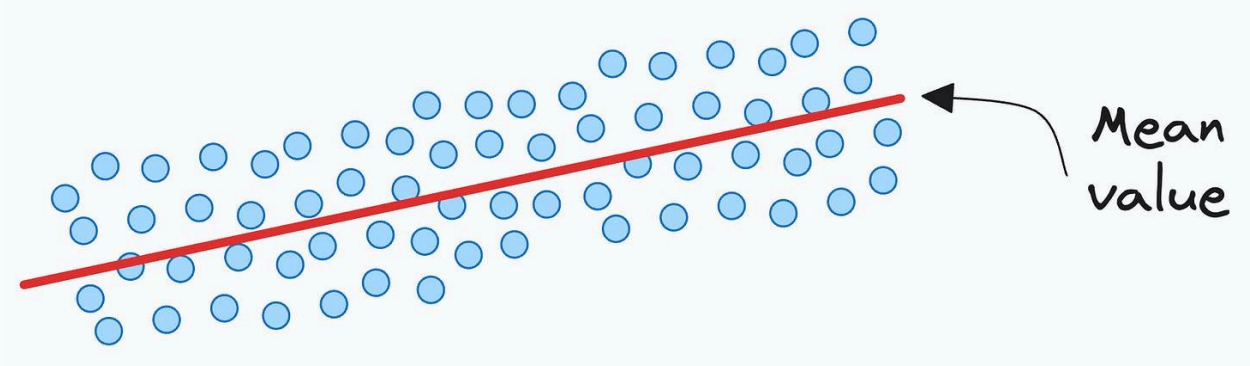
Confidence Interval and Prediction Interval

Statistical estimates always have some uncertainty.

For instance, a linear regression model never predicts an actual value.

Consider a simple example of modeling house prices just based on its area. A prediction wouldn't tell the true value of a house based on its area. This is because different houses of the same size can have different prices.

Instead, what it predicts is the **mean value** related to the outcome at a particular input.



The point is...

There's always some uncertainty involved in statistical estimates, and it is important to communicate it.

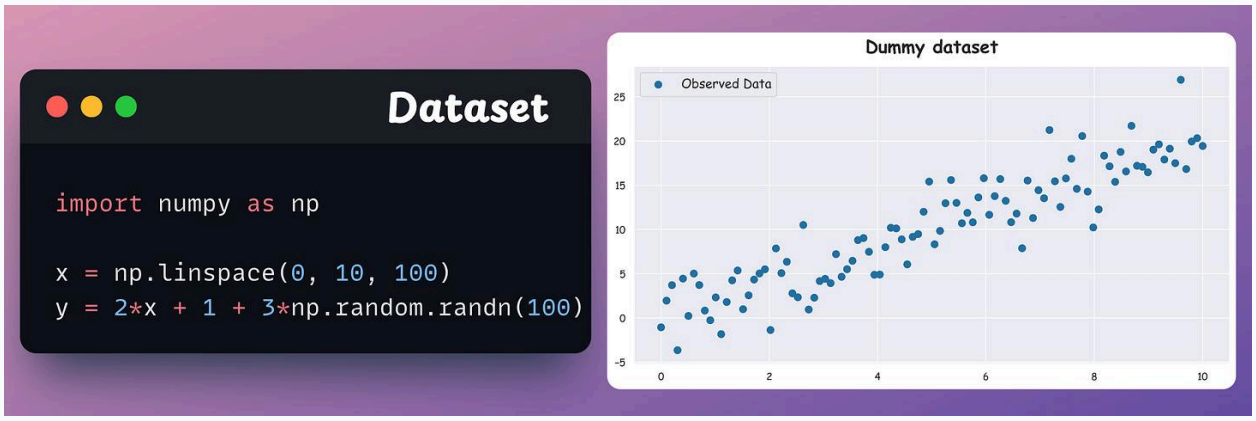
In this specific case, there are two types of uncertainties:

- The uncertainty in estimating the true mean value.
- The uncertainty in estimating the true value.

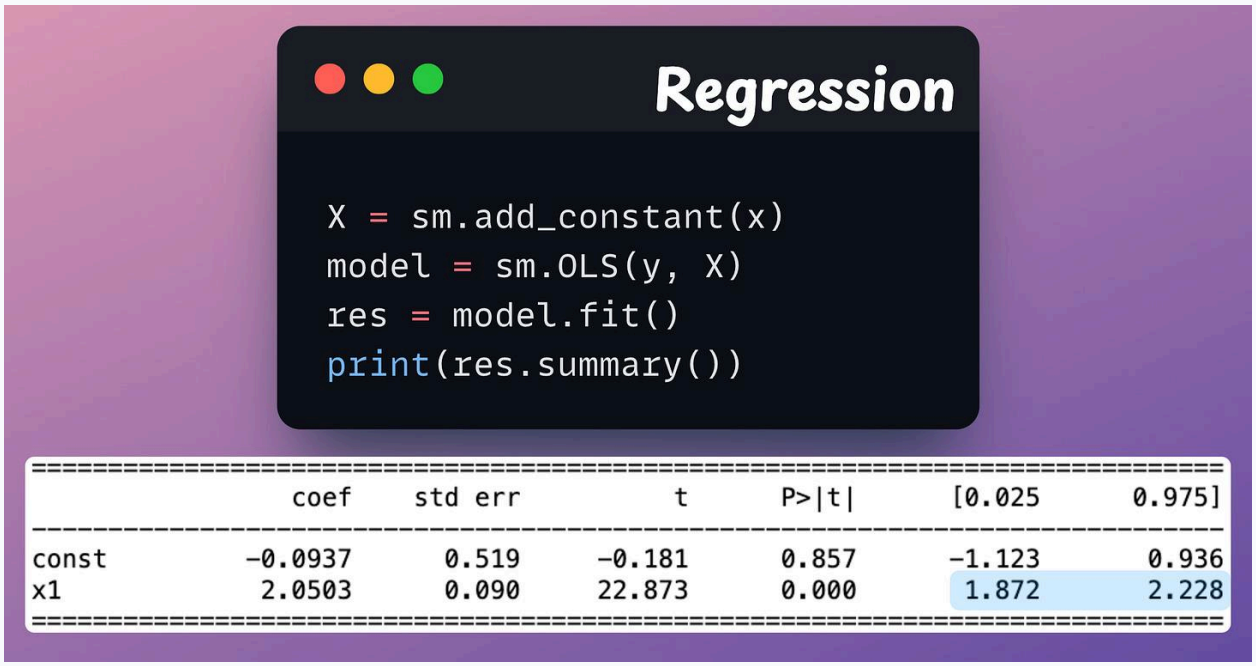
Confidence interval and prediction interval help us capture these uncertainties.

Let's understand.

Consider the following dummy dataset:



Let's fit a linear regression model using statsmodel and print a part of the regression summary:

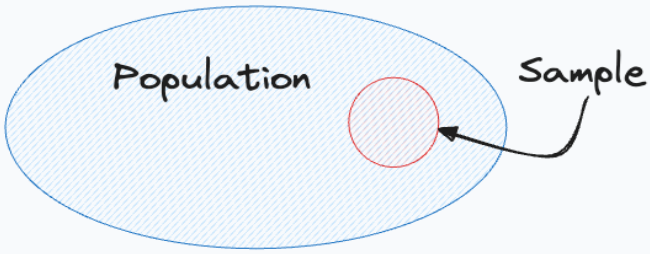


Notice that the coefficient of the predictor "x1" is 2.0503 with a 95% interval of [1.872, 2.228].

It is a 95% interval because 0.975-0.025 = 0.95.

This is known as the confidence interval, which comes from **sampling uncertainty**.

More specifically, this uncertainty arises because the data we used above for modeling is just a **sample** of the population.



So, if we gathered more such samples and fit an OLS to each sample, the true coefficient (which we can only know if we had the data for the entire population) would lie 95% of the time in this confidence interval.

Next, we use this model to make a prediction as follows:

```
>>> res.get_prediction([1,3]).summary_frame(0.05)
   mean  mean_se  mean_ci_lower  mean_ci_upper  obs_ci_lower  obs_ci_upper
0  7.62    0.391          6.832          8.419          1.06         14.19
```

Predicted value | 95% Confidence interval | 95% Prediction interval

- The predicted value is 7.62 (mean).
- The 95% confidence interval is [6.832, 8.419].
- The 95% prediction interval is [1.06, 14.19].

The confidence interval we saw above was for the coefficient, so what does the confidence interval represent in this case?

Similar to what we discussed above, the data is just a **sample** of the population.

The regression fit obtained by this sample produced a prediction (some mean value) for the input x=3.

However, if we gathered more such samples and fit an OLS to each dataset, the **true mean value** (which we can only know if we had the data for the entire

population) for this specific input (x=3) would lie 95% of the time in this confidence interval.

Coming to the prediction interval...

```
>>> res.get_prediction([1,3]).summary_frame(0.05)

   mean   mean_se  mean_ci_lower  mean_ci_upper  obs_ci_lower  obs_ci_upper
0  7.62     0.391          6.832          8.419          1.06         14.19
```

Predicted value

95% Confidence interval

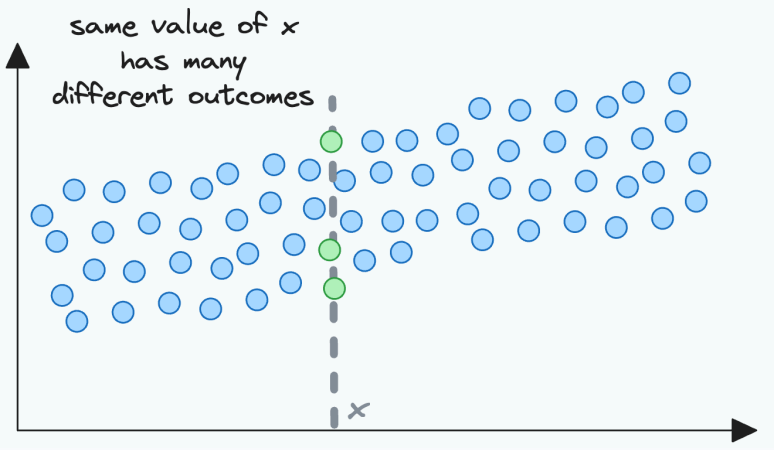
95% Prediction interval

...we notice that it is wider than the confidence interval. Why is it, and what does this interval tell?

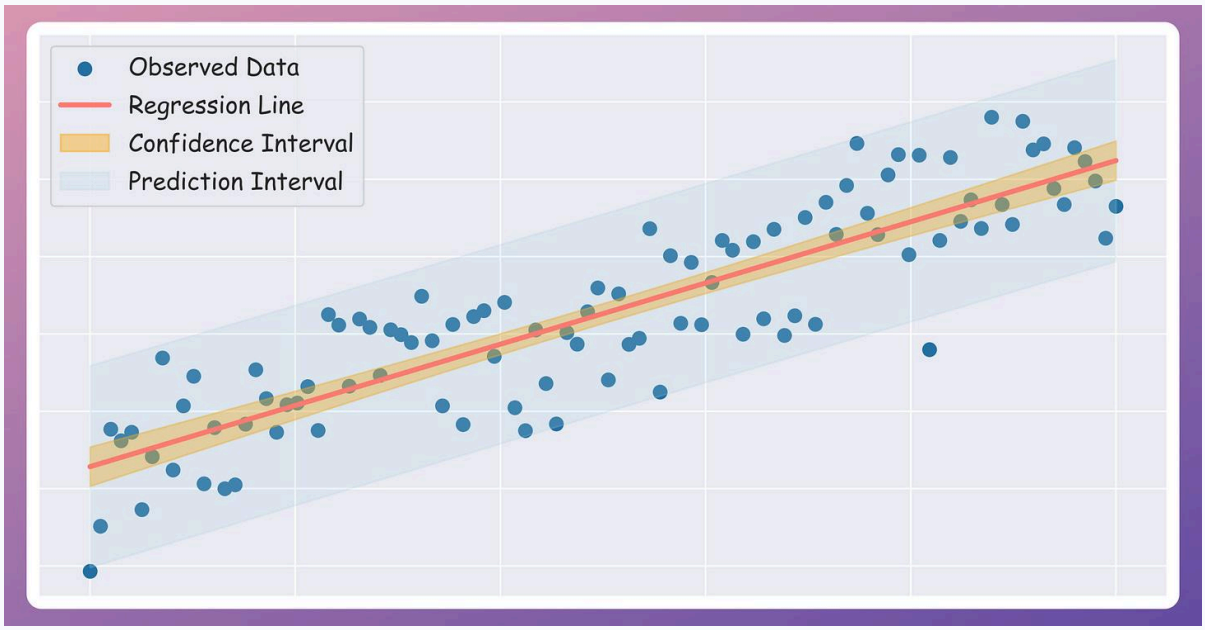
What we saw above with confidence interval was about estimating the true population mean at a specific input.

What we are talking about now is obtaining an interval where the **true value** for an input can lie.

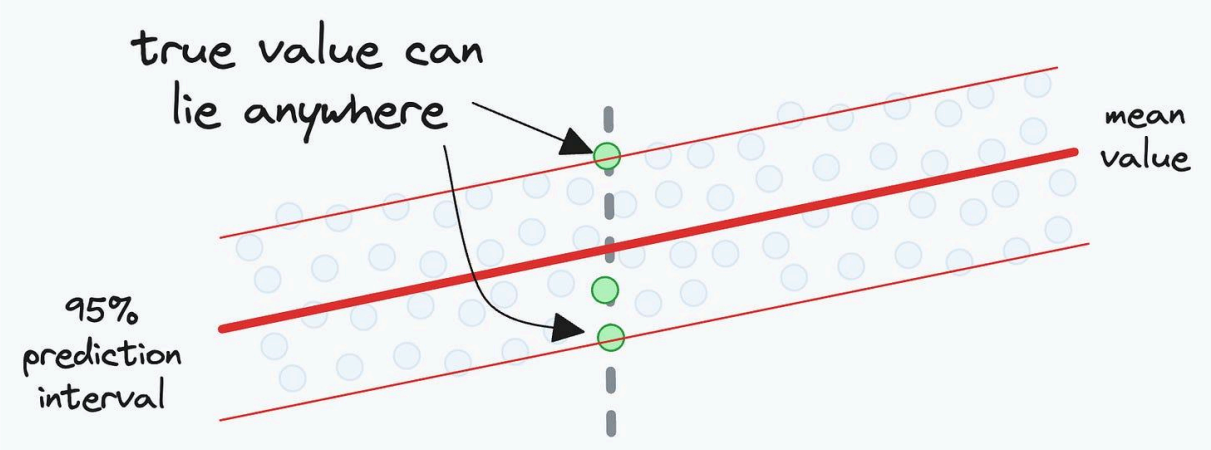
Thus, this additional uncertainty appears because in our dataset, for the same value of input x, there can be multiple different values of the outcome. This is depicted below:



Thus, it is wider than the confidence interval. Plotting it across the entire input range, we get the following plot:



Given that the model is predicting a mean value (as depicted below), we have to represent the prediction uncertainty that the actual value can lie anywhere in the prediction interval:



A 95% prediction interval tells us that we can be 95% sure that the actual value of this observation will fall within this interval.

So to summarize:

- A confidence interval captures the sampling uncertainty. More data means less sampling uncertainty, which in turn leads to a smaller interval.
- In addition to the sampling uncertainty, the prediction interval also represents the uncertainty in estimating the true value of a particular data point. Thus, it is wider than the confidence interval.

Communicating these uncertainties is quite crucial in decision-making because it provides a clearer understanding of the reliability and precision of predictions.

This transparency allows stakeholders to make more informed decisions by considering the range of possible outcomes and the associated risks.

Why is OLS Called an Unbiased Estimator?

The OLS estimator for linear regression (shown below) is known as an unbiased estimator.

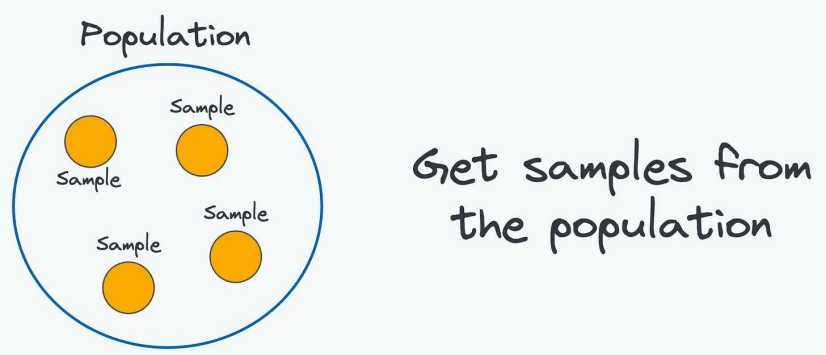
$$\theta = (X^T X)^{-1} X^T y$$

- What do we mean by that?
- Why is OLS called such?

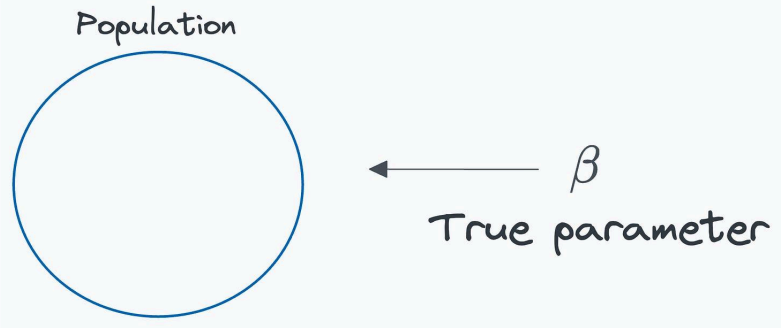
Background

The goal of statistical modeling is to make conclusions about the whole population.

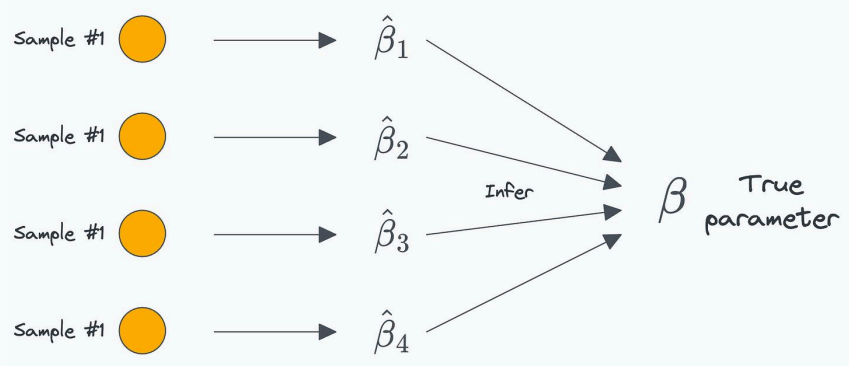
However, it is pretty obvious that observing the entire population is impractical.



In other words, given that we cannot observe (or collect data of) the entire population, we cannot obtain the true parameter (β) for the population:



Thus, we must obtain parameter estimates ($\hat{\beta}$) on samples and infer the true parameter (β) for the population from those estimates:



And, of course, we want these sample estimates ($\hat{\beta}$) to be reliable to determine the actual parameter (β).

The OLS estimator ensures that.

Let's understand how!

True population model

When using a linear regression model, we assume that the response variable (Y) and features (X) for the entire population are related as follows:

$$Y = \beta * X + \epsilon$$

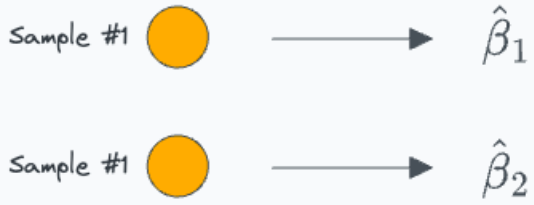
- β is the true parameter that we are not aware of.
- ϵ is the error term.

Expected Value of OLS Estimates

The closed-form solution of OLS is given by:

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

What's more, as discussed above, using OLS on different samples will result in different parameter estimates:



Let's find the expected value of OLS estimates $E[\hat{\beta}]$.

Simply put, the expected value is the average value of the parameters if we run OLS on many samples.

This is given by:

$$E[\hat{\beta}] = E[(X^T X)^{-1} X^T Y]$$

Substitute $\hat{\beta}$ as the OLS solution

Here, substitute $Y = \beta X + \varepsilon$:

$$\begin{aligned} E[\hat{\beta}] &= E[(X^T X)^{-1} X^T Y] \\ &= E[(X^T X)^{-1} X^T (\beta X + \epsilon)] \end{aligned}$$

If you are wondering how we can substitute $Y = \beta X + \varepsilon$ when we don't know what β is, then here's the explanation:

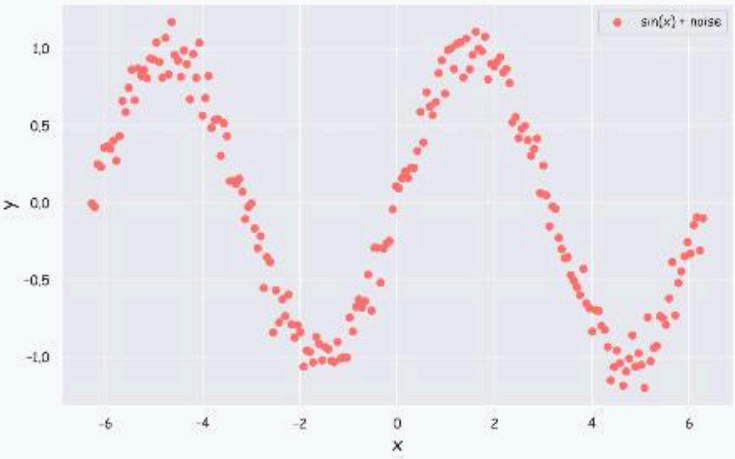
See, we can do that substitution because even if we don't know the parameter β for the whole population, we know that the sample was drawn from the population.

Thus, the equation in terms of the true parameters ($Y = \beta X + \varepsilon$) still holds for the sample.

Let me give you an example.

Say the population data was defined by $y = sin(x) + \varepsilon$. Of course, we wouldn't know this, but just keep that aside for a second.

$$y = sin(x) + \epsilon$$



Now, even if we were to draw samples from this population data, the true equation $y = sin(x) + \varepsilon$ would still be valid on the sampled data points, wouldn't it?

The same idea has been extended for expected value.

Coming back to the following:

$$E[\hat{\beta}] = E[(X^T X)^{-1} X^T Y]$$
$$= E[(X^T X)^{-1} X^T (\beta X + \epsilon)]$$

Let's open the inner parenthesis:

$$E[\hat{\beta}] = E[(X^T X)^{-1} X^T X \beta] + E[(X^T X)^{-1} X^T \epsilon]$$

$(X^T X)^{-1} X^T X$ → I

$(X^T X)^{-1} X^T \epsilon$: Some constant

Simplifying, we get:

$$E[\hat{\beta}] = E[\beta] + (X^T X)^{-1} X^T E[\epsilon]$$

$E[\epsilon]$: zero

β is constant so its expected value is β

And finally, what do we get?

$$E[\hat{\beta}] = \beta$$

The expected value of parameter estimates on the samples equals the true parameter value β.

And this is precisely what the definition of an unbiased estimator is.

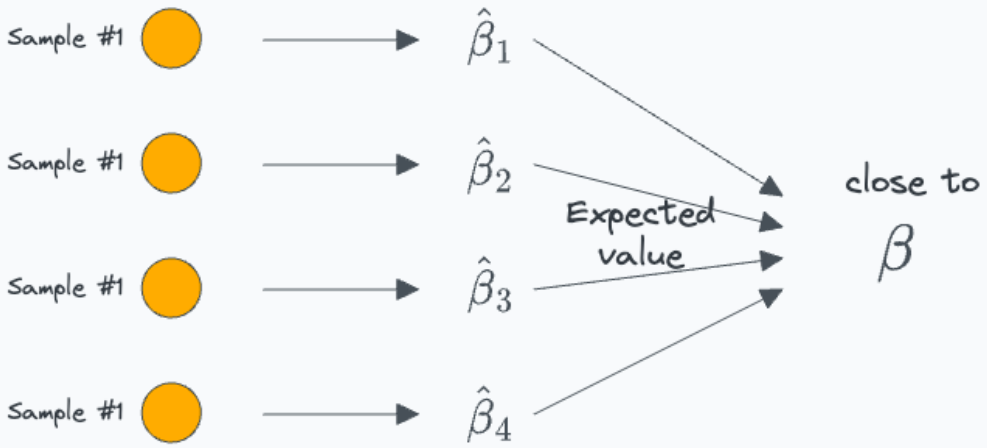
More formally, an estimator is called unbiased if the expected value of the parameters is equal to the actual parameter value.

And that is why we call OLS an unbiased estimator.

An important takeaway

Many people misinterpret unbiasedness with the idea that the parameter estimates from a single run of OLS on a sample are equal to the true parameter values.

Don't make that mistake.



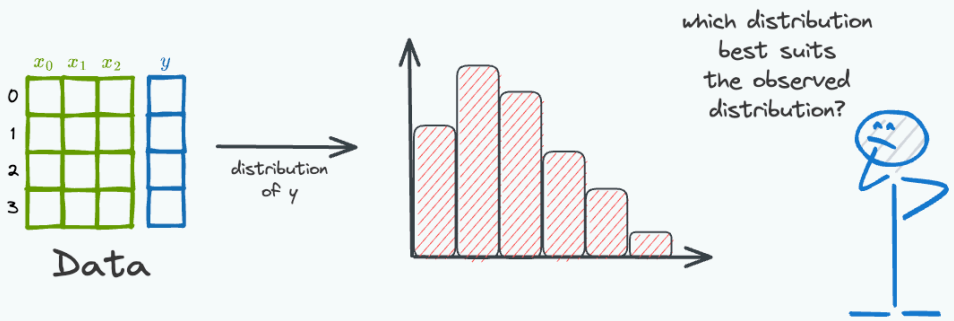
Instead, unbiasedness implies that if we were to generate OLS estimates on many different samples (drawn from the same population), then the expected value of obtained estimates will be equal to the true population parameter.

And, of course, all this is based on the assumption that we have good representative samples and that the assumptions of linear regression are not violated.

Bhattacharyya Distance

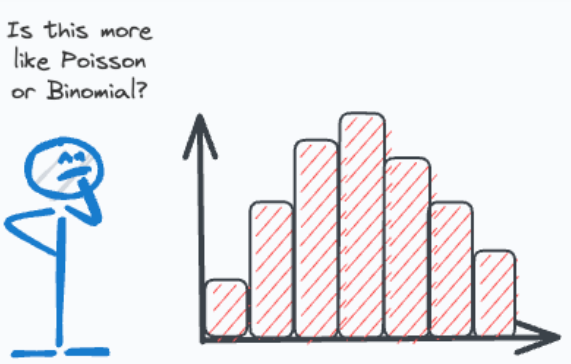
Assessing the similarity between two probability distributions is quite helpful at times. For instance, imagine we have a labeled dataset (X, y).

By analyzing the label (y) distribution, we may hypothesize its distribution before building a statistical model.



We also looked at this in an earlier chapter on generalized linear models (GLMs).

While visual inspection is often helpful, this approach is quite subjective and may lead to misleading conclusions.



Thus, it is essential to be aware of quantitative measures as well. Bhattacharyya distance is one such reliable measure.

It quantifies the similarity between two probability distributions.

The core idea is to approximate the overlap between two distributions, which measures the "closeness" between the two distributions under consideration.

Bhattacharyya distance is measured as follows:

For two discrete probability distributions

$$D_B(P, Q) = -\ln\left(\sum_{x \in X} \sqrt{P(x) \cdot Q(x)}\right)$$

For two continuous probability distributions (replace summation with an integral):

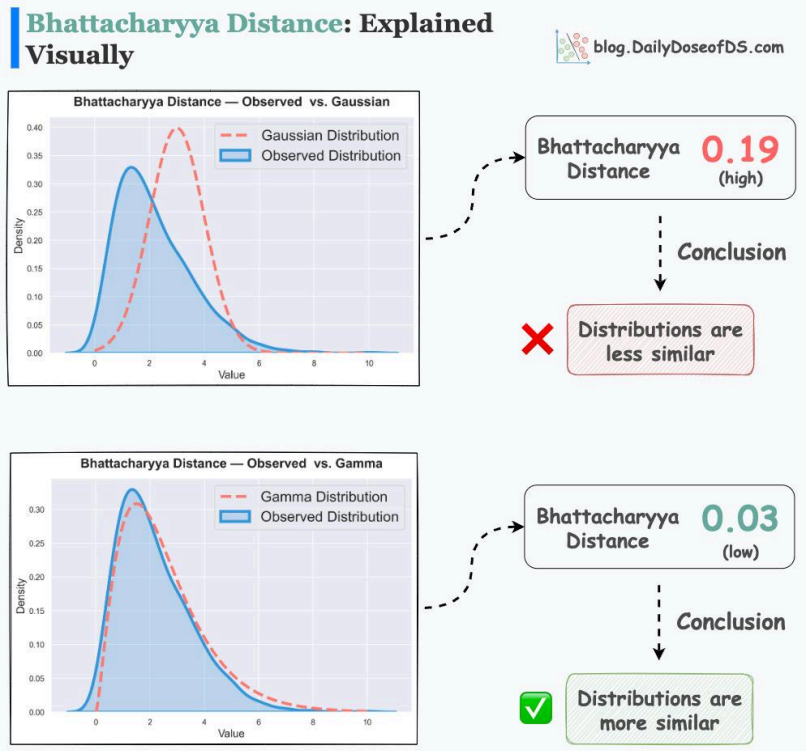
$$D_B(P, Q) = -\ln\left(\int \sqrt{P(x) \cdot Q(x)}\, dx\right)$$

Its effectiveness is evident from the image below.

Here, we have an observed distribution (Blue). Next, we measure its distance from:

- Gaussian → 0.19.
- Gamma → 0.03.

A high Bhattacharyya distance indicates less overlap or more dissimilarity. This lets us conclude that the observed distribution resembles a Gamma distribution.



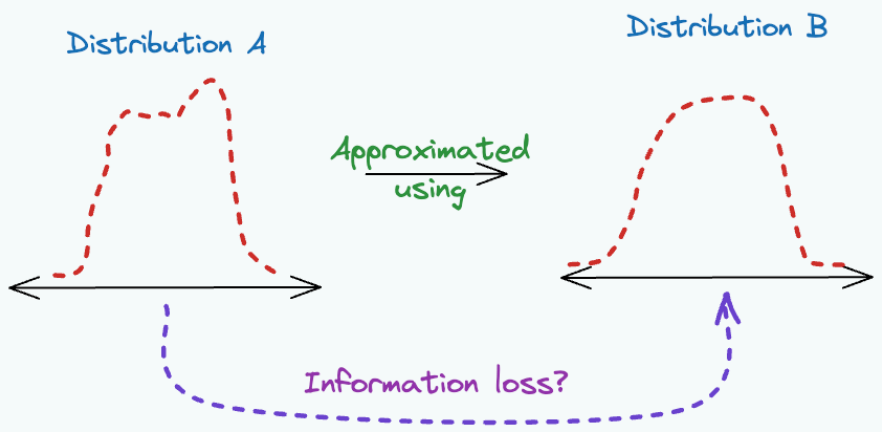
The results also resonate with visual inspection.

KL Divergence vs. Bhattacharyya distance

Now, many often get confused between KL Divergence and Bhattacharyya distance. Effectively, both are quantitative measures to determine the "similarity" between two distributions.

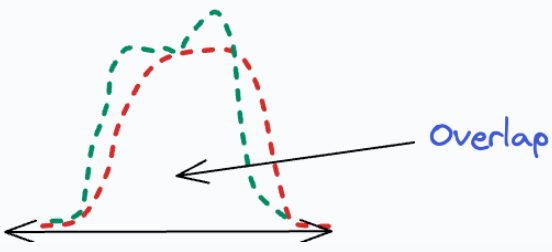
However, their notion of "similarity" is entirely different.

The core idea behind KL Divergence is to assess how much information is lost when one distribution is used to approximate another distribution.



The more information is lost, the more the KL Divergence and, consequently, the less the "similarity". Also, approximating a distribution Q using P may not be the same as doing the reverse — P using Q. This makes KL Divergence asymmetric in nature.

Moving on, Bhattacharyya distance measures the overlap between two distributions.



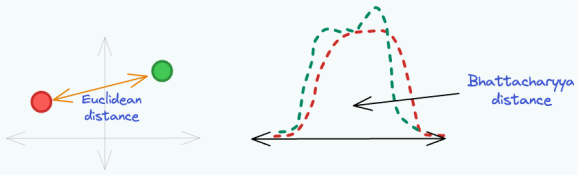
This "overlap" is often interpreted as a measure of closeness (or distance) between the two distributions under consideration. Thus, Bhattacharyya distance primarily serves a distance metric, like Euclidean, for instance.

Being a distance metric, it is symmetric in nature. We can also verify this from the distance formula:

$$D_B(P,Q) = -\ln\left(\int \sqrt{P(x)\cdot Q(x)}\,dx\right)$$

$$D_B(P,Q) = D_B(Q,P) \quad \text{Symmetric}$$

Just like we use Euclidean distance to find the distance between two points, we can use Bhattacharyya distance to find the distance between two distributions.



But if we intend to measure the amount of information lost when we approximate one distribution using another, KL divergence is more apt. In fact, KL divergence also serves as a loss function in machine learning algorithms at times (in t-SNE).

- Say we have an observed distribution (P) and want to approximate it with another simpler distribution Q.
- So, we can define a simpler parametric distribution Q.
- Next, we can measure the information lost by approximating P using Q with KL divergence.
- As we want to minimize the information lost, we can use KL divergence as our objective function.
- Finally, we can use gradient descent to determine the parameters of Q such that we minimize the KL divergence.

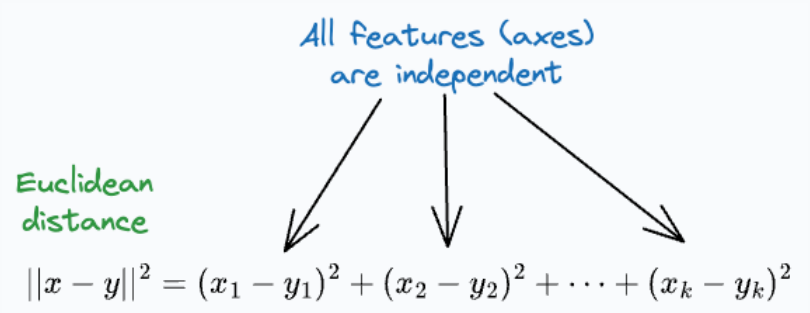
Bhattacharyya distance has many applications, not just in machine learning but in many other domains. For instance:

- Using this distance, we can simplify complex distributions to simple ones if the distance is low.
- In image processing, Bhattacharyya distance is often used for image matching. By comparing the color or texture distributions of images, it helps identify similar objects or scenes, etc.

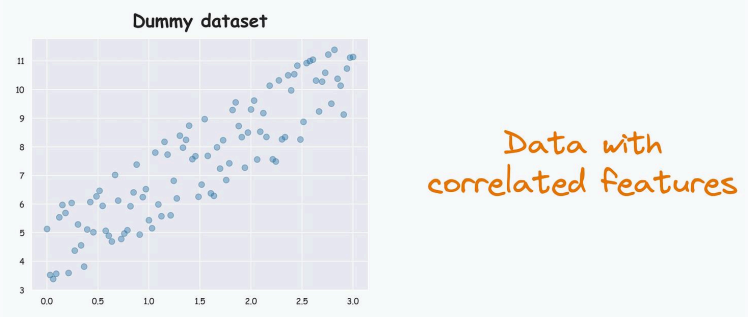
The only small caveat is that Bhattacharyya distance does not satisfy the triangle inequality, so that's something to keep in mind.

Why Prefer Mahalanobis Distance Over Euclidean distance?

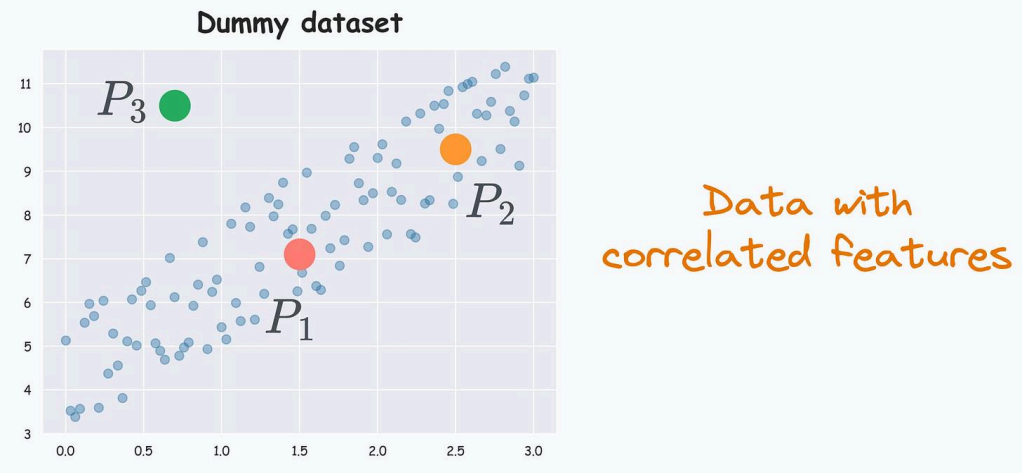
During distance calculation, Euclidean distance assumes independent axes.



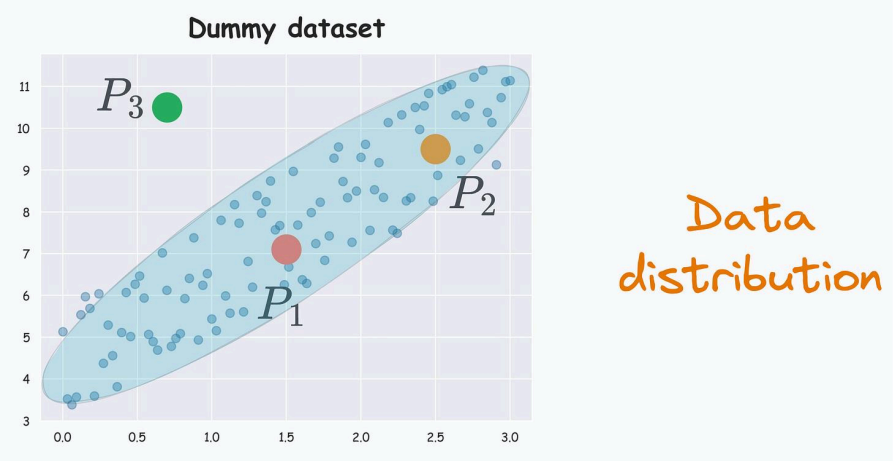
Thus, Euclidean distance will produce misleading results if your features are correlated. For instance, consider this dummy dataset below:



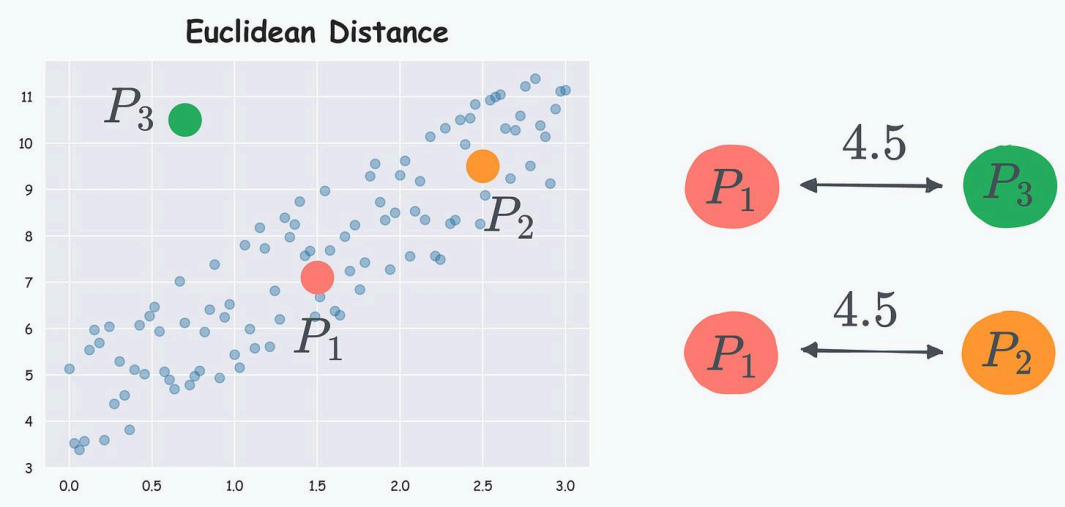
Clearly, the features are correlated. Here, consider three points marked P1, P2, and P3 in this dataset.



Considering the data distribution, something tells us that P2 is closer to P1 than P3. This is because P2 lies more within the data distribution than P3.



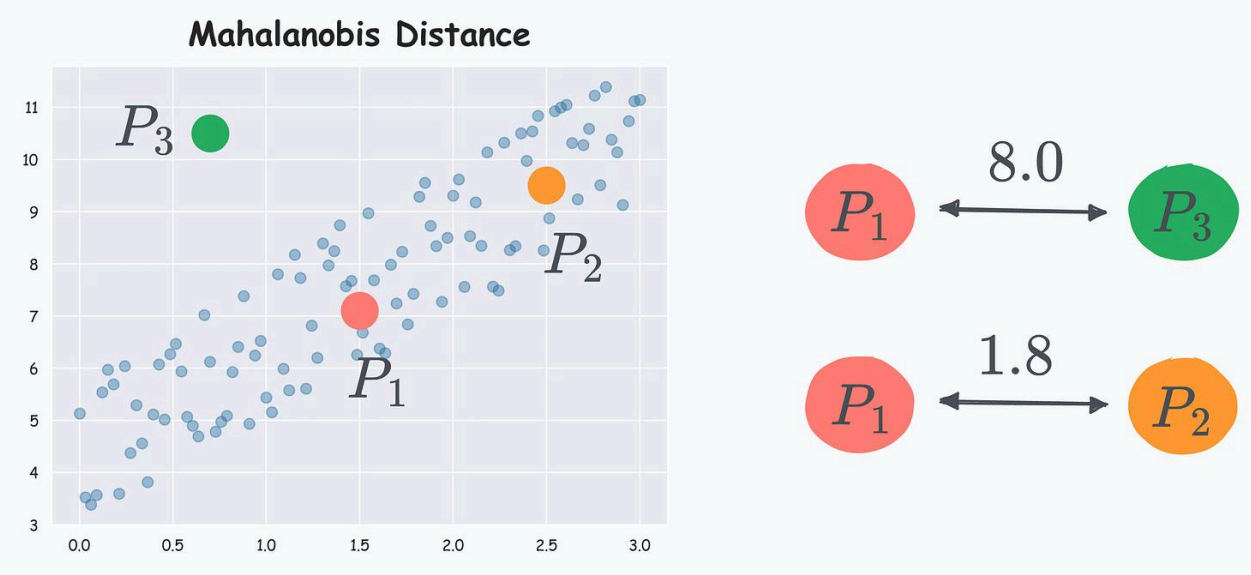
Yet, Euclidean distance ignores this, and P2 and P3 come out to be equidistant to P1, as depicted below:



Mahalanobis distance addresses this limitation. It is a distance metric that takes into account the data distribution.

As a result, it can measure how far away a data point is from the distribution, which Euclidean can not.

Referring to the earlier dataset again, with Mahalanobis distance, P2 comes out to be closer to P1 than P3.



How does it work?

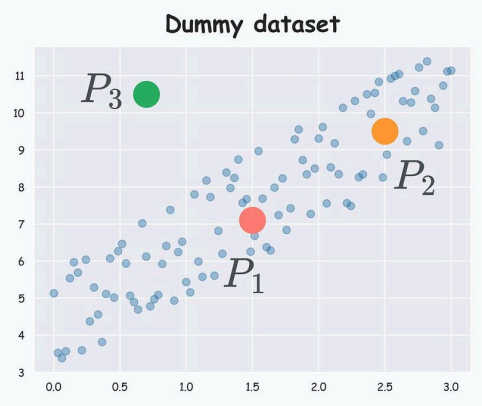
In a gist, the objective is to construct a new coordinate system with independent and orthogonal axes. The steps are:

- Step 1: Transform the columns into uncorrelated variables.
- Step 2: Scale the new variables to make their variance equal to 1.
- Step 3: Find the Euclidean distance in this new coordinate system.

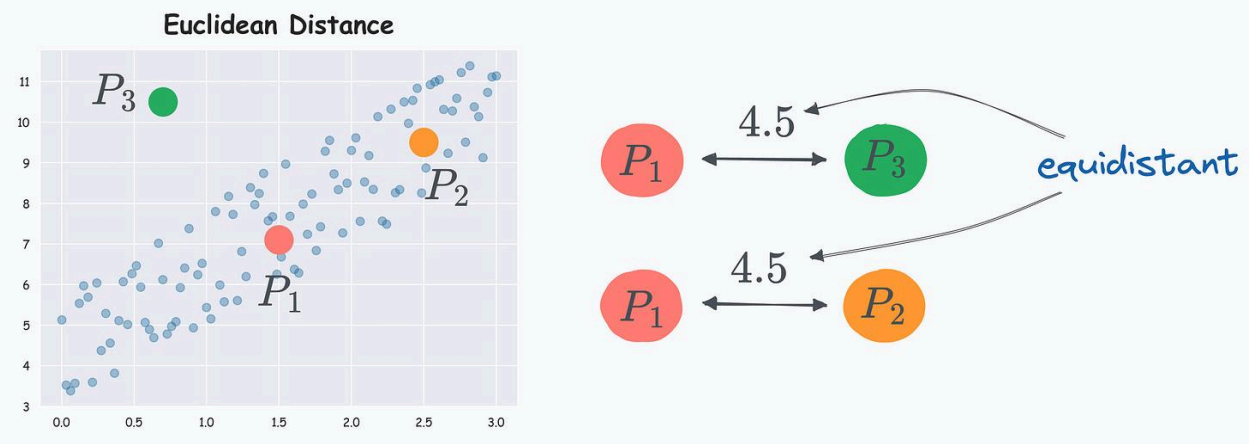
So, eventually, we do use Euclidean distance. However, we first transform the data to ensure that it obeys the assumptions of Euclidean distance.

Uses

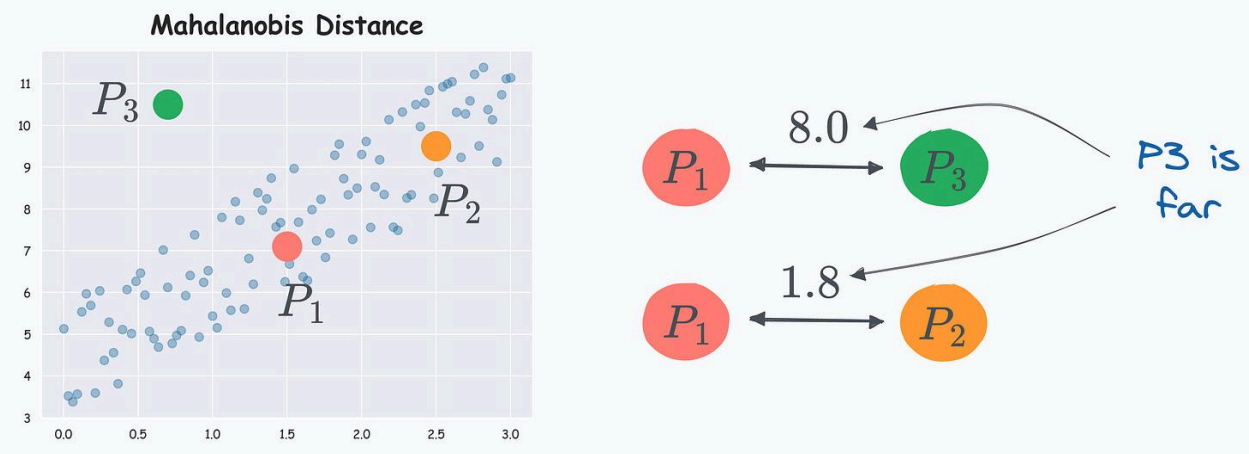
One of the most common use cases of Mahalanobis distance is outlier detection. Reconsidering the dataset we discussed earlier where P3 is clearly an outlier.



If we consider P1 as the distribution's centroid and use Euclidean distance, we will infer that P3 is not an outlier as both P2 and P3 are equidistant to P1.



Using Mahalanobis distance, however, provides a clearer picture:



This becomes more useful in a high-dimensional setting where visualization is infeasible.

Another use case we typically do not hear of often, but that exists is a variant of kNN that is implemented with Mahalanobis distance instead.

Scipy implements the Mahalanobis distance, which you can check here: https://bit.ly/3LjAymm.

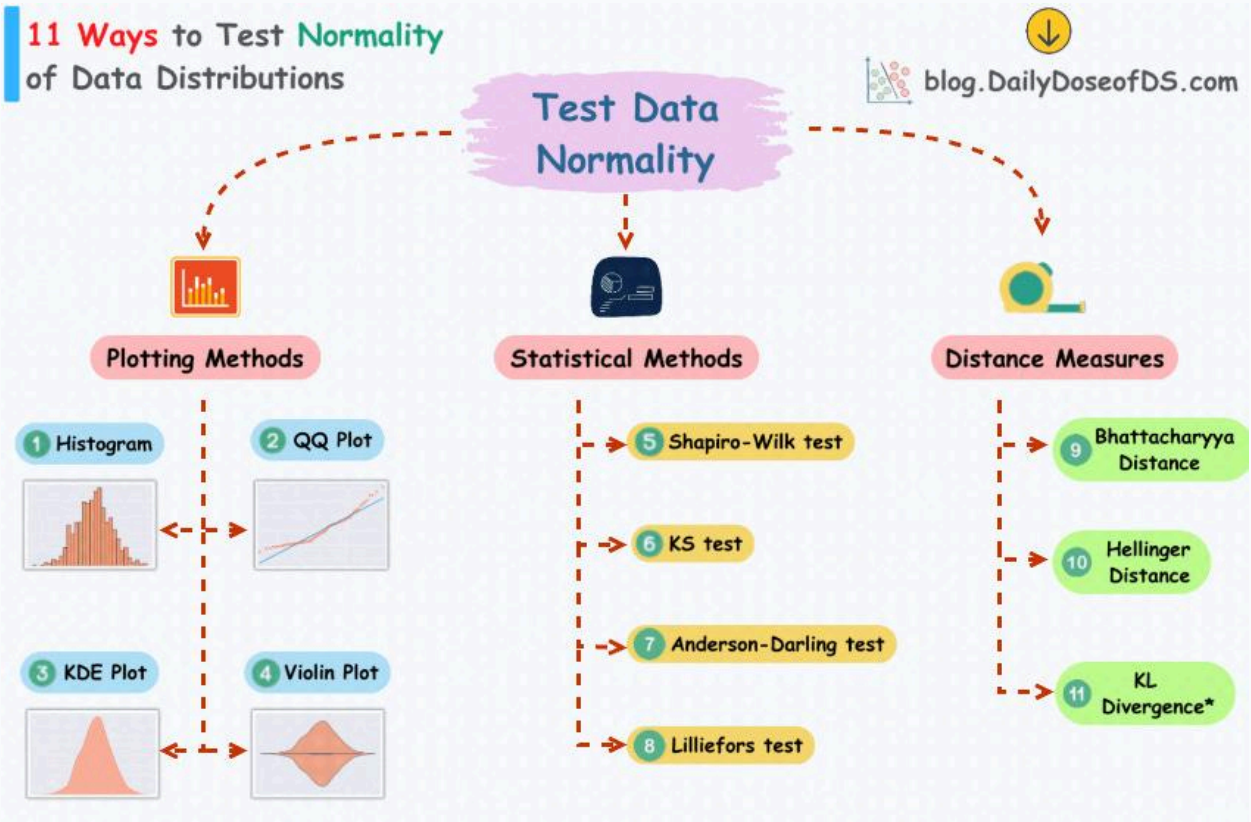
11 Ways to Determine Data Normality

Many ML models assume (or work better) under the presence of normal distribution.

For instance:

- Linear regression assumes residuals are normally distributed.
- At times, transforming the data to normal distribution can be beneficial.
- Linear discriminant analysis (LDA) is derived under the assumption of normal distribution, etc.

Thus, being aware of the ways to test normality is extremely crucial for data scientists. The visual below depicts the 11 essential ways to test normality.



Let's understand these in this chapter.

#1) Plotting Methods (self-explanatory)

- Histogram
- QQ Plot (We shall cover it in the plotting section of this book).
- KDE Plot
- Violin Plot

While plotting is often reliable, it is a subjective approach and prone to errors.

Thus, we must know reliable quantitative measures as well.

#2) Statistical Methods:

1) Shapiro-Wilk test:
    - Finds a statistic using the correlation between the observed data and the expected values under a normal distribution.
    - The p-value indicates the likelihood of observing such a correlation if the data were normally distributed.
    - A high p-value indicates a normal distribution.
2) KS test:
    - Measures the max difference between the cumulative distribution functions (CDF) of observed and normal distribution.
    - The output statistic is based on the max difference between the two CDFs.
    - A high p-value indicates a normal distribution.
3) Anderson-Darling test:
    - Measures the differences between the observed data and the expected values under a normal distribution.
    - Emphasizes the differences in the tail of the distribution.
    - This makes it particularly effective at detecting deviations in the extreme values.

4) Lilliefors test:

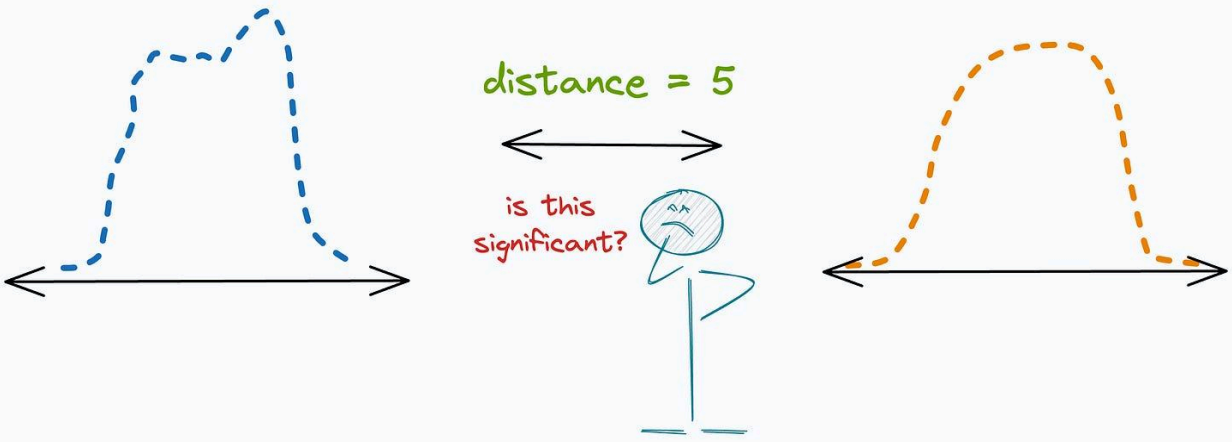
- It is a modification of the KS test.
- The KS test is appropriate in situations where the parameters of the reference distribution are known.
- If the parameters are unknown, Lilliefors is recommended.
- Get started: Statsmodel Docs.

#3) Distance Measures

Distance measures are another reliable and more intuitive way to test normality.

But they can be a bit tricky to use.

See, the problem is that a single distance value needs more context for interpretability.



For instance, if the distance between two distributions is 5, is this large or small?

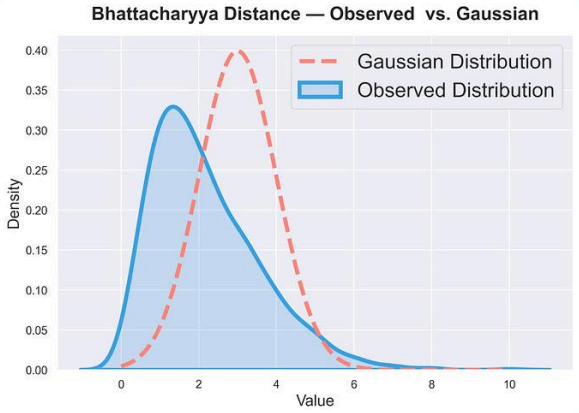
We need more context.

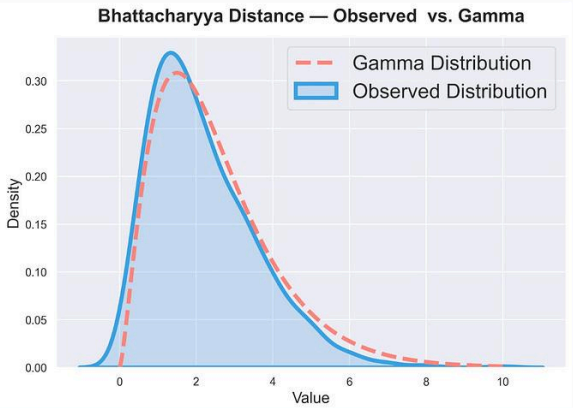
I prefer using these measures as follows:

- Find the distance between the observed distribution and multiple reference distributions.
- Select the reference distribution with the minimum distance to the observed distribution.

Here are a few distance common and useful measures:

1) **Bhattacharyya distance:**



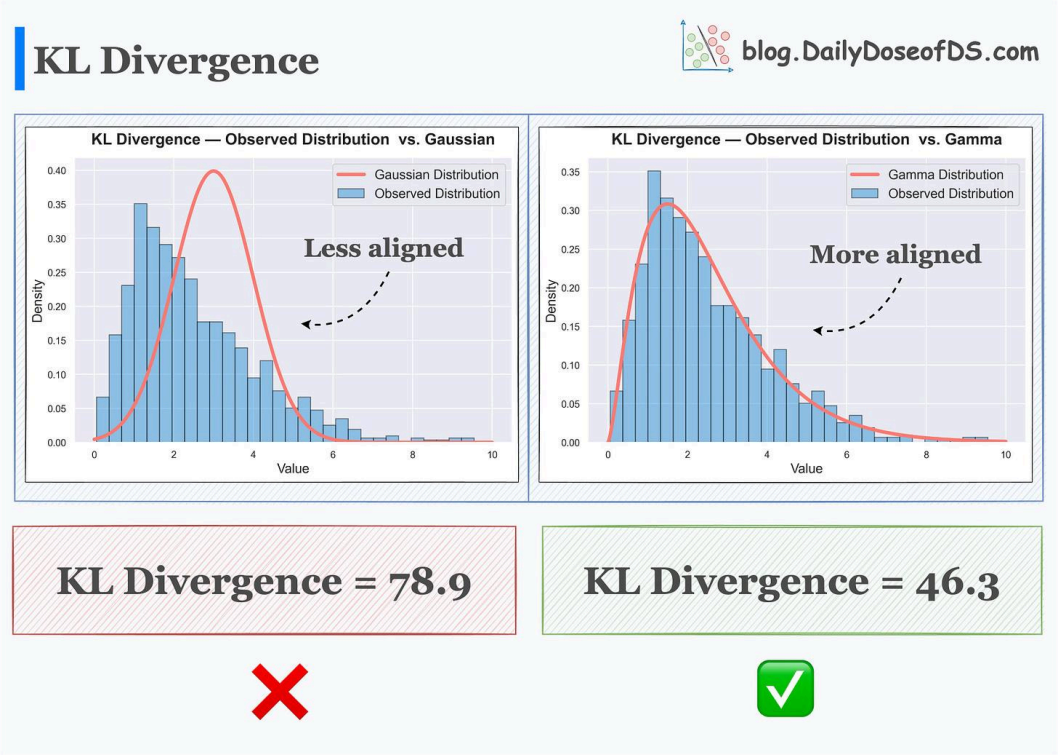


- Measure the overlap between two distributions.
- This "overlap" is often interpreted as closeness between two distributions.
- Choose the distribution that has the least Bhattacharyya distance to the observed distribution.

2) **Hellinger distance:**

- It is used quite similar to how we use the Bhattacharyya distance
- The difference is that Bhattacharyya distance does not satisfy triangular inequality.
- But Hellinger distance does.

3) **KL Divergence:**



- It is not entirely a "distance metric" per se, but can be used in this case.
- Measure information lost when one distribution is approximated using another distribution.
- The more information is lost, the more the KL Divergence.
- Choose the distribution that has the least KL divergence from the observed distribution.

KL divergence is used as a loss function in the t-SNE algorithm.

Probability vs. Likelihood

In data science and statistics, many folks often use "probability" and "likelihood" interchangeably.

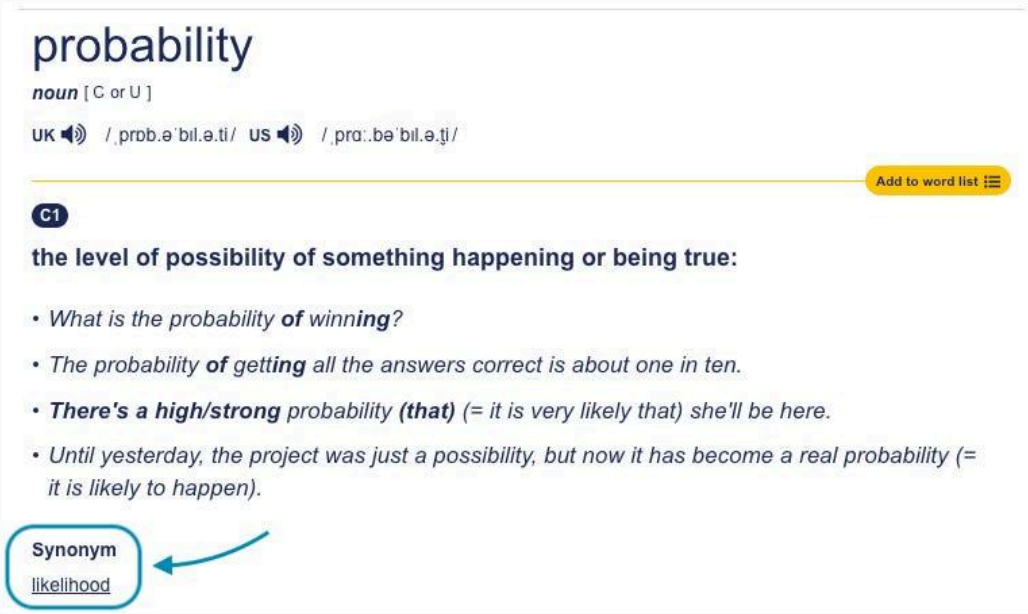
However, likelihood and probability DO NOT convey the same meaning.

And the misunderstanding is somewhat understandable, given that they carry similar meanings in our regular language.

While writing this chapter, I searched for their meaning in the Cambridge Dictionary. Here's what it says:

- Probability: the level of possibility of something happening or being true.
- Likelihood: the chance that something will happen.

If you notice closely, "likelihood" is the only synonym of "probability".



Anyway.

In my opinion, it is crucial to understand that probability and likelihood convey very different meanings in data science and statistics.

Let's understand!

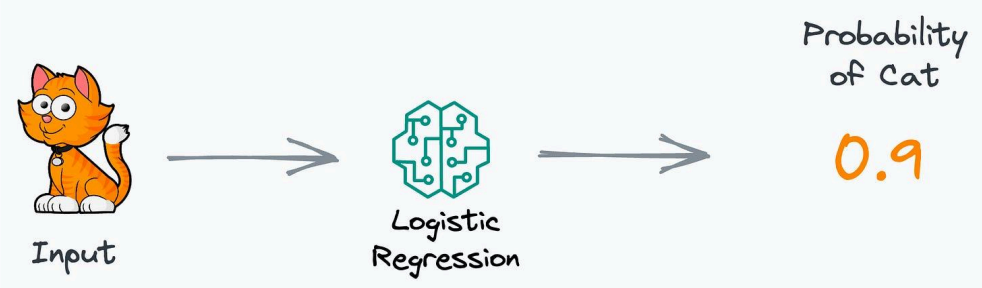
Probability is used in contexts where you wish to know the possibility/odds of an event.

For instance, what is the:

- Probability of obtaining an even number in a die roll?
- Probability of drawing an ace of diamonds from a deck?
- and so on...

When translated to ML, probability can be thought of as:

- What is the probability that a transaction is fraud?
- What is the probability that an image depicts a cat?
- and so on...



Essentially, many classification models, like logistic regression or a classification neural network, etc., assign the probability of a specific label to an input.

When calculating probability, the model's parameters are known. Also, we assume that they are trustworthy.

For instance, to determine the probability of a head in a coin toss, we mostly assume and trust that it is a fair coin.

Likelihood, on the other hand, is about explaining events that have already occurred.

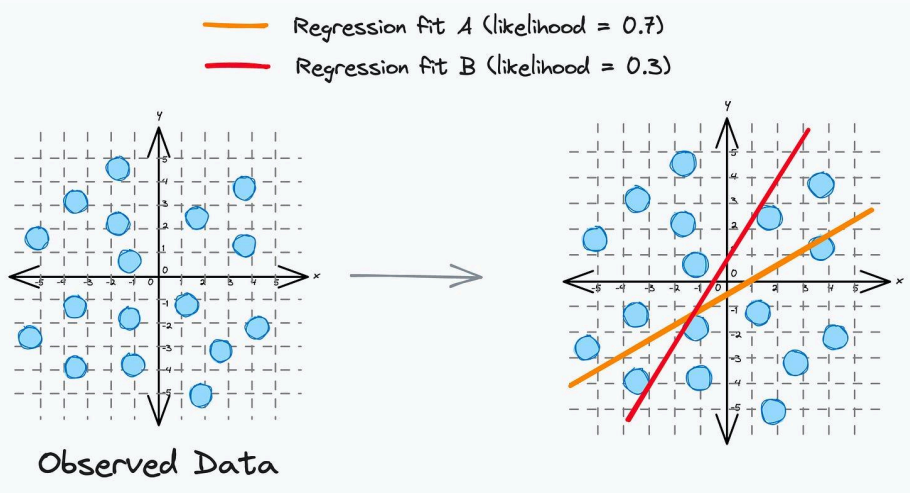
Unlike probability (where parameters are known and assumed to be trustworthy)...

...likelihood helps us determine if we can trust the parameters in a model based on the observed data.

Let me elaborate more on that.

Assume you have collected some 2D data and wish to fit a straight line with two parameters — slope (m) and intercept (c).

Here, likelihood is defined as the support provided by a data point for some particular parameter values in your model.

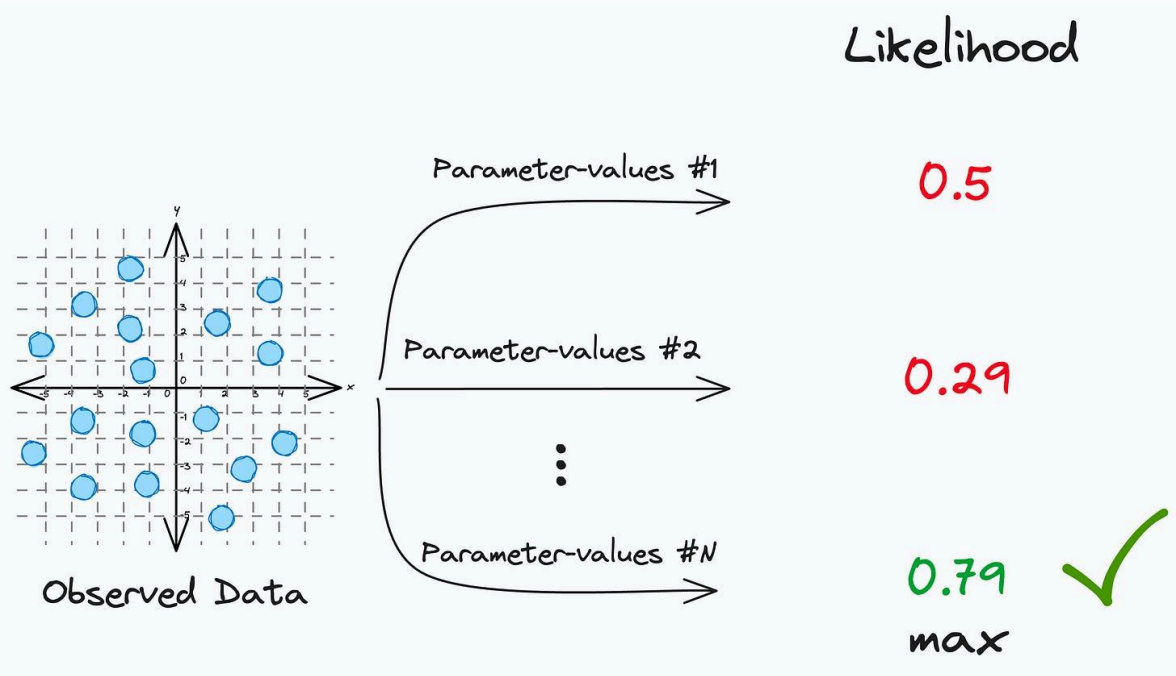


Here, you will ask questions like:

- If I model this data with the parameters:
    - m=2 and c=1, what is the likelihood of observing the data?
    - m=3 and c=2, what is the likelihood of observing the data?
    - and so on...

The above formulation popularly translates into the maximum likelihood estimation (MLE), which we discussed here: (MLE vs. EM — What's the Difference?)

In maximum likelihood estimation, you have some observed data and you are trying to determine the specific set of parameters (θ) that maximize the likelihood of observing the data.



Using the term "likelihood" is like:

- I have a possible explanation for my data. (In the above illustration, "explanation" can be thought of as the parameters you are trying to determine)
- How well does my explanation explain what I've already observed? This is precisely quantified with likelihood.

For instance:

- Observation: The outcomes of 10 coin tosses are "~~HHHHHHHHTHH~~".
- Explanation: I think it is a fair coin ($p=0.5$).
- What is the likelihood that my above explanation is true based on the observed data?

To summarize...

It is immensely important to understand that in data science and statistics, likelihood and probability DO NOT convey the same meaning.

As explained above, they are pretty different.

In probability:

- We determine the possibility of an event.
- We know the parameters associated with the event and assume them to be trustworthy.

In likelihood:

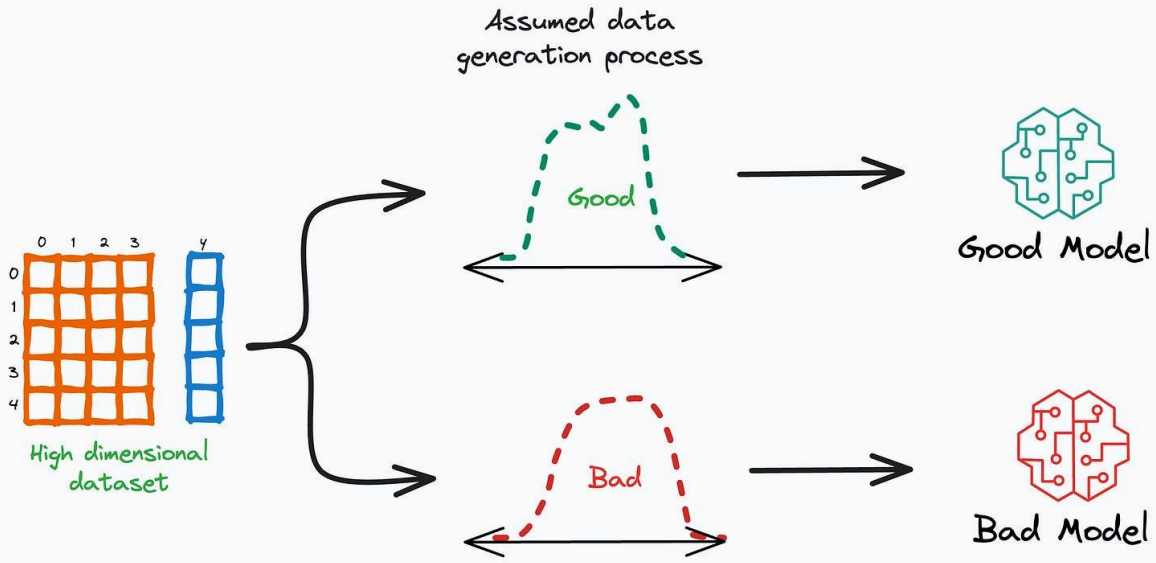
- We have some observations.
- We have an explanation (or parameters).
- Likelihood helps us quantify whether the explanation is trustworthy.

Hope that helped!

11 Key Probability Distributions in Data Science

Statistical models assume an underlying data generation process.

This is exactly what lets us formulate the generation process, using which we define the maximum likelihood estimation (MLE) step.



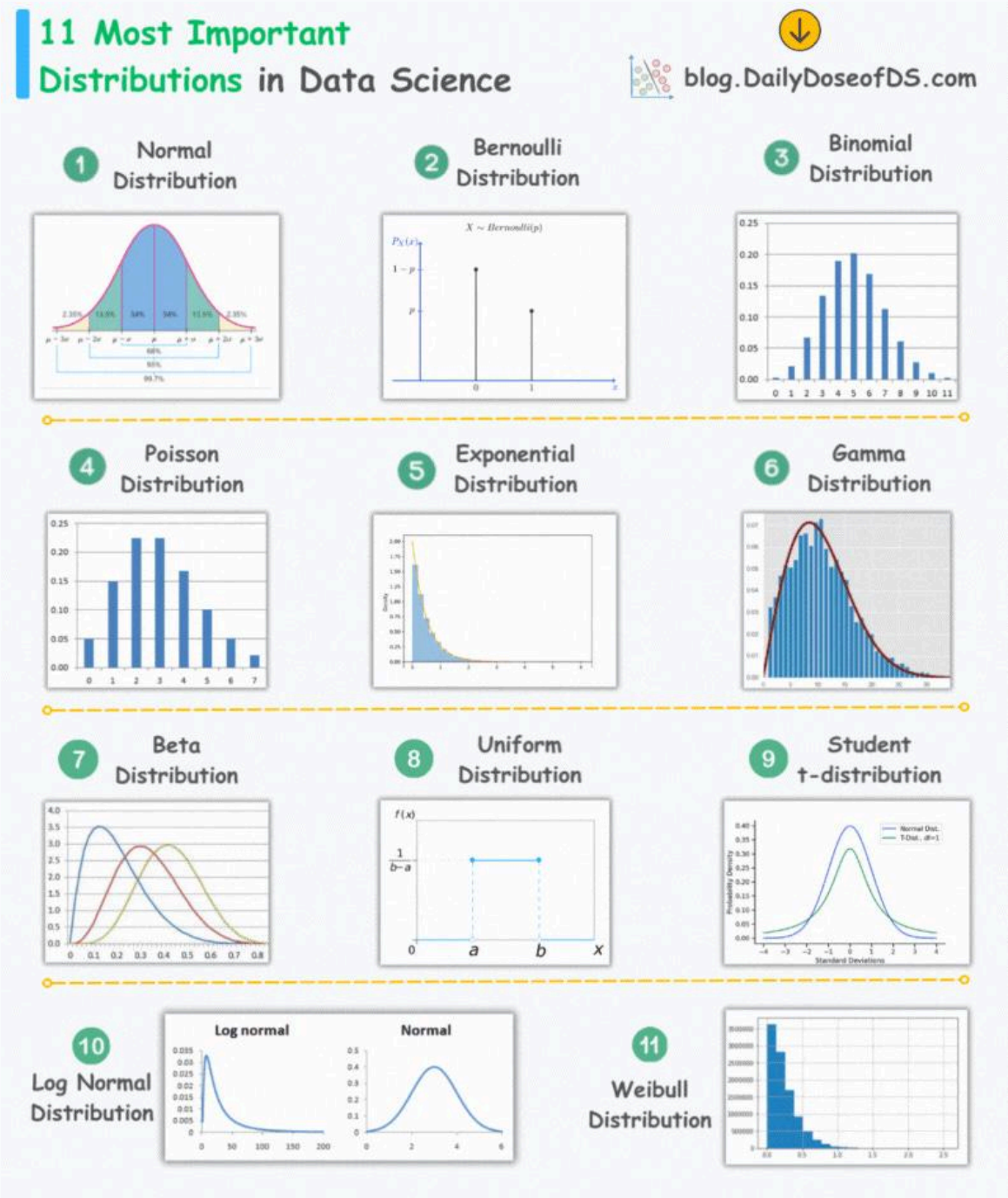
Thus, when dealing with statistical models, the model performance becomes entirely dependent on:

- Your understanding of the data generation process.
- The distribution you chose to model data with, which, in turn, depends on how well you understand various distributions.

We shall also look at this in the chapter on GLMs (this chapter).

Thus, it is crucial to be aware of some of the most important distributions and the type of data they can model.

The visual below depicts the 11 most important distributions in data science:



Let's understand them briefly and how they are used.

Normal Distribution

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

- The most widely used distribution in data science.
- Characterized by a symmetric bell-shaped curve
- It is parameterized by two parameters—mean and standard deviation.
- Example: Height of individuals.

Bernoulli Distribution

$$P(X = x) = \begin{cases} p \ for \ x = 1 \\ 1 - p \ for \ x = 0 \end{cases}$$

- A discrete probability distribution that models the outcome of a binary event.
- It is parameterized by one parameter—the probability of success.
- Example: Modeling the outcome of a single coin flip.

Binomial Distribution

$$P\left(X = x\right) = \frac{n!}{x!(n-x)!} p^x \left(1 - p\right)^{(n-x)}$$

- It is Bernoulli distribution repeated multiple times.
- A discrete probability distribution that represents the number of successes in a fixed number of independent Bernoulli trials.
- It is parameterized by two parameters—the number of trials and the probability of success.

Poisson Distribution

$$P(X) = \frac{\lambda^x e^{-\lambda}}{X!}$$

- A discrete probability distribution that models the number of events occurring in a fixed interval of time or space.
- It is parameterized by one parameter—lambda, the rate of occurrence.
- Example: Analyzing the number of goals a team will score during a specific time period.

Exponential Distribution

$$f(x, \lambda) = \begin{cases} \lambda \cdot e^{-(\lambda x)} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

- A continuous probability distribution that models the time between events occurring in a Poisson process.
- It is parameterized by one parameter—lambda, the average rate of events.
- Example: Analyzing the time between goals scored by a team.

Gamma Distribution

$$f(x;\alpha,\beta) = \begin{cases} \frac{1}{\beta^{\alpha}\Gamma(\alpha)} x^{\alpha-1} e^{-x/\beta} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- It is a variation of the exponential distribution.
- A continuous probability distribution that models the waiting time for a specified number of events in a Poisson process.
- It is parameterized by two parameters—alpha (shape) and beta (rate).
- Example: Analysing the time it would take for a team to score three goals.

Beta Distribution

PDF	$\frac{x^{\alpha-1}(1-x)^{\beta-1}}{\mathrm{B}(\alpha,\beta)}$ where $\mathrm{B}(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$ and Γ is the Gamma function.

- It is used to model probabilities, thus, it is bounded between [0,1].
- Differs from Binomial in this respect that in Binomial, probability is a parameter.
- But in Beta, the probability is a random variable.

Uniform Distribution

$$f(x) = \begin{cases} \frac{1}{b-a}, & x \in [a,b] \\ 0, & otherwise \end{cases}$$

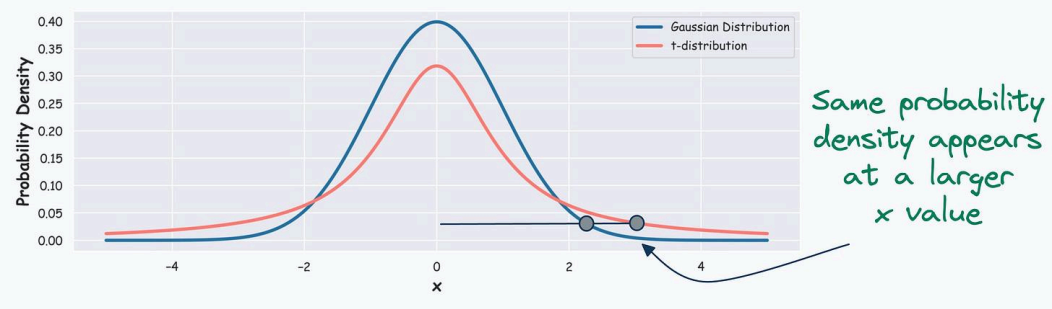
- All outcomes within a given range are equally likely.
- It can be continuous or discrete.
- It is parameterized by two parameters: a (min value) and b (max value).
- Example: Simulating the roll of a fair six-sided die, where each outcome (1, 2, 3, 4, 5, 6) has an equal probability.

Log-Normal Distribution

$$X \sim N(\mu, \sigma^2) \qquad Y = e^X$$
$$Y \sim Lognormal(\mu, \sigma^2)$$

- A continuous probability distribution where the logarithm of the variable follows a normal distribution.
- It is parameterized by two parameters—mean and standard deviation.
- Example: Typically, in stock returns, the natural logarithm follows a normal distribution.

Student t-distribution



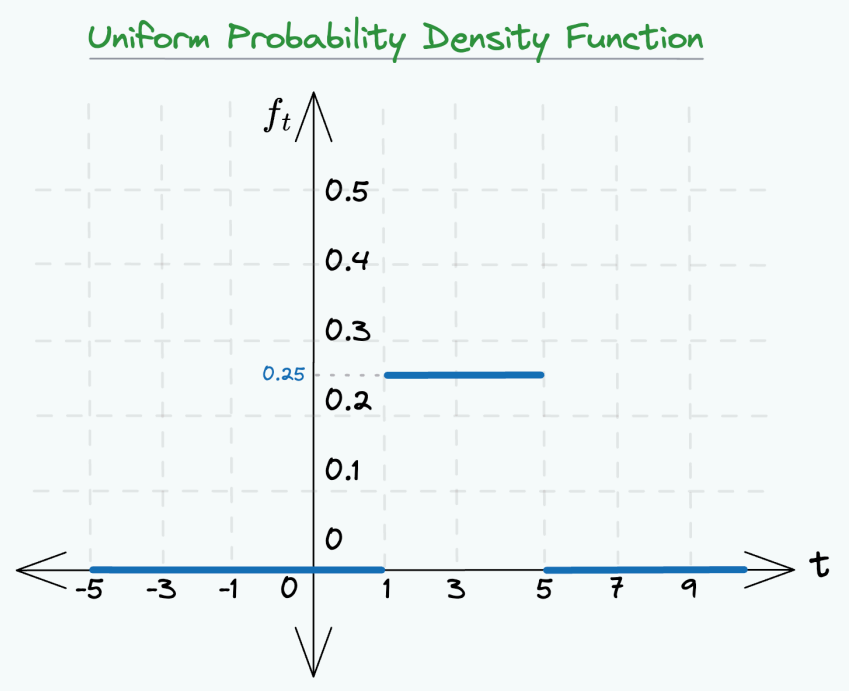
- It is similar to normal distribution but with longer tails (shown above).
- It is used in t-SNE to model low-dimensional pairwise similarities.

Weibull

- Models the waiting time for an event.
- Often employed to analyze time-to-failure data.

A Common Misinterpretation of Continuous Probability Distributions

Consider the following probability density function of a continuous probability distribution. Say it represents the time one may take to travel from point A to B.



- For simplicity, we are assuming a uniform distribution in the interval [1,5].
- Essentially, it says that it will take somewhere between 1 and 5 minutes to go from A to B. Never more, never less.

Thus, the probability density function (PDF) can be written as follows:

$$f_T(t) = \begin{cases} \frac{1}{5-1} & : 1 \leq t \leq 5 \\ 0 & : \text{otherwise} \end{cases}$$

Answer the following question for me:

Q) What is the probability that one will take precisely three minutes P(T=3) to reach point B?

- A) 1/4 (or 0.25)
- B) Area under the curve from t=[1,3].
- C) Area under the curve from t=[3,5].

Decide on an answer before you read further.

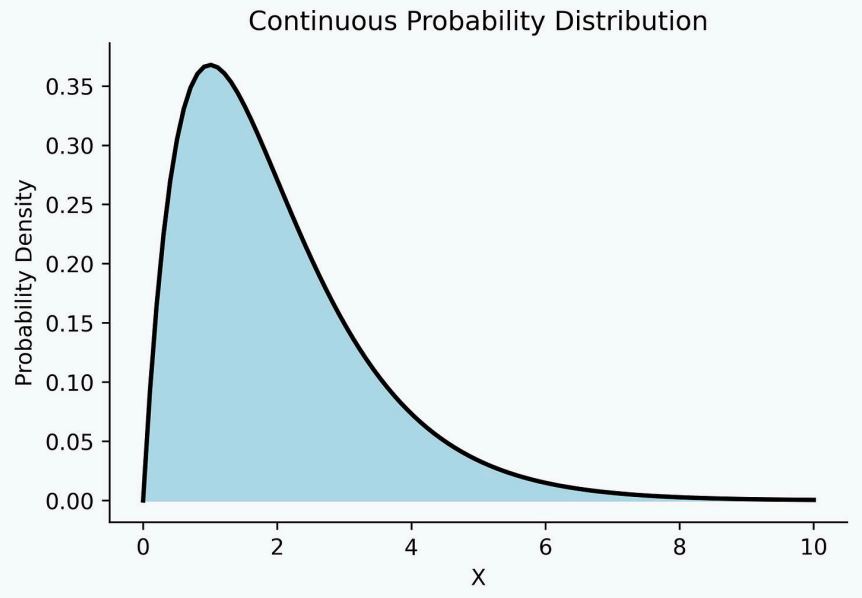
Well, all of the above answers are wrong.

The correct answer is ZERO.

And I intentionally kept only wrong answers here so that you never forget something fundamentally important about continuous probability distributions.

Let's dive in!

The probability density function of a continuous probability distribution may look as follows:



Some conditions for this probability density function are:

- It should be defined for all real numbers (can be zero for some values).

$$PDF \rightarrow f_X(x) \; ; \; x \in \mathbb{R}$$

This is in contrast to a discrete probability distribution which is only defined for a list of values.

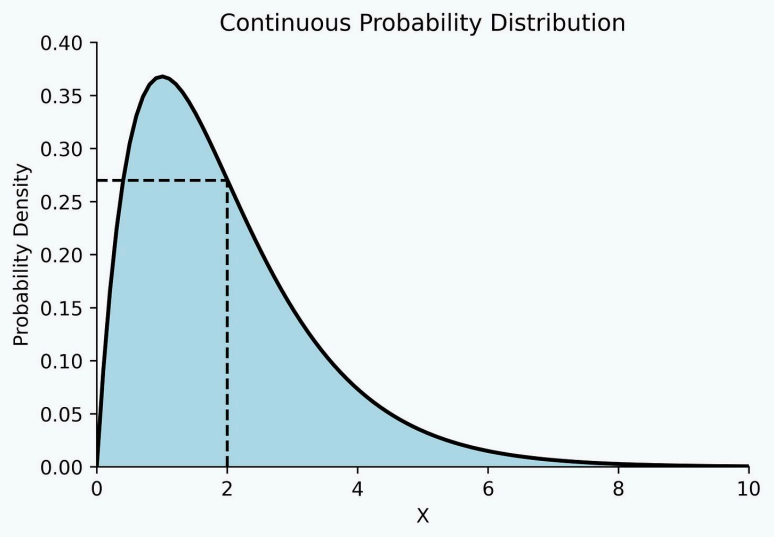
- The area should be 1.

$$\int_{-\infty}^{\infty} f_X(x) \, dx = 1$$

- The function should be non-negative for all real values.

$$f_X(x) \geq 0 \;\; \forall \;\; x$$

Here, many folks often misinterpret that the probability density function represents the probability of obtaining a specific value.



For instance, by looking at the above probability density function, many incorrectly conclude that the probability of the random variable X being 2 is close to 0.27.

$$P(X = 2) = 0.27$$

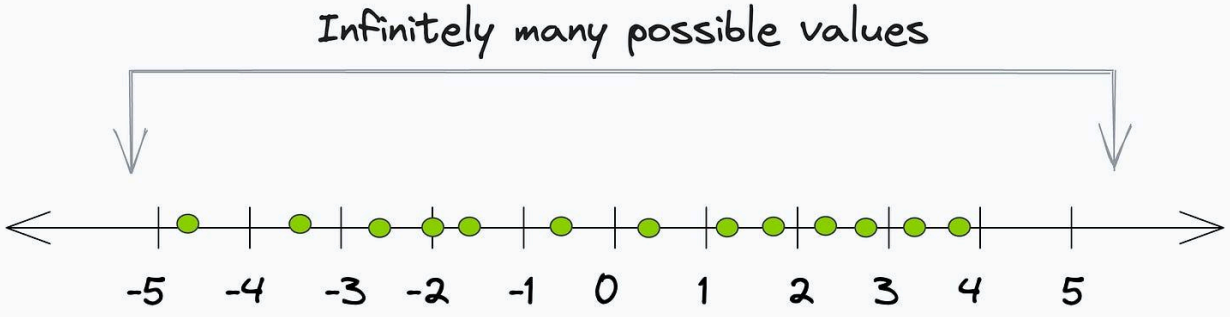
But contrary to this common belief, a probability density function:

- DOES NOT depict the probabilities of a specific value.
- is not meant to depict a discrete random variable.

Instead, a probability density function:

- depicts the rate at which probabilities accumulate around each point.
- is only meant to depict a continuous random variable.

Now, there are infinitely possible values that a continuous random variable may take.



So the probability of obtaining a specific value is always zero (or infinitesimally small).

Thus, answering our original question, the probability that one will take three minutes to reach point B is ZERO.

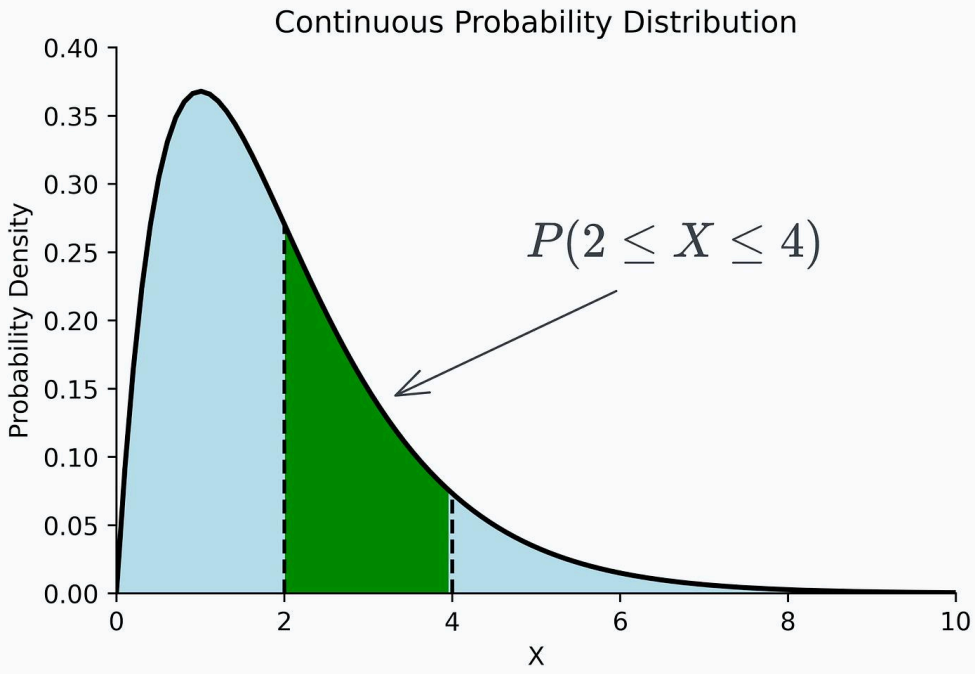
So what is the purpose of using a probability density function?

In statistics, a PDF is used to calculate the probability over an interval of values.

Thus, we can use it to answer questions such as...

- What is the probability that it will take between:
    - 3 to 4 minutes to reach point B from point A, or,
    - 2 to 4 minutes to reach point B from point A, and so on...

And we do this using integrals.



More formally, the probability that a random variable X will take values in the interval [a,b] is:

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

Simply put, it's the area under the curve from [a,b].

From the above probability estimation over an interval, we can also verify that the probability of obtaining a specific value is indeed zero.

By substituting b=a, we get:

$$P(a \leq X \leq b) = \int_{a}^{b} f_X(x)\, dx$$

Substitute $b = a$

$$P(a \leq X \leq a) = \int_{a}^{a} f_X(x)\, dx$$

$$P(a \leq X \leq a) = 0$$

$$\boxed{P(a) = 0}$$

To summarize, always remember that in a continuous probability distribution:

- The probability density function does not depict the exact probability of obtaining a specific value.
- Estimating the probability for a precise value of the random value makes no sense because it is infinitesimally small.

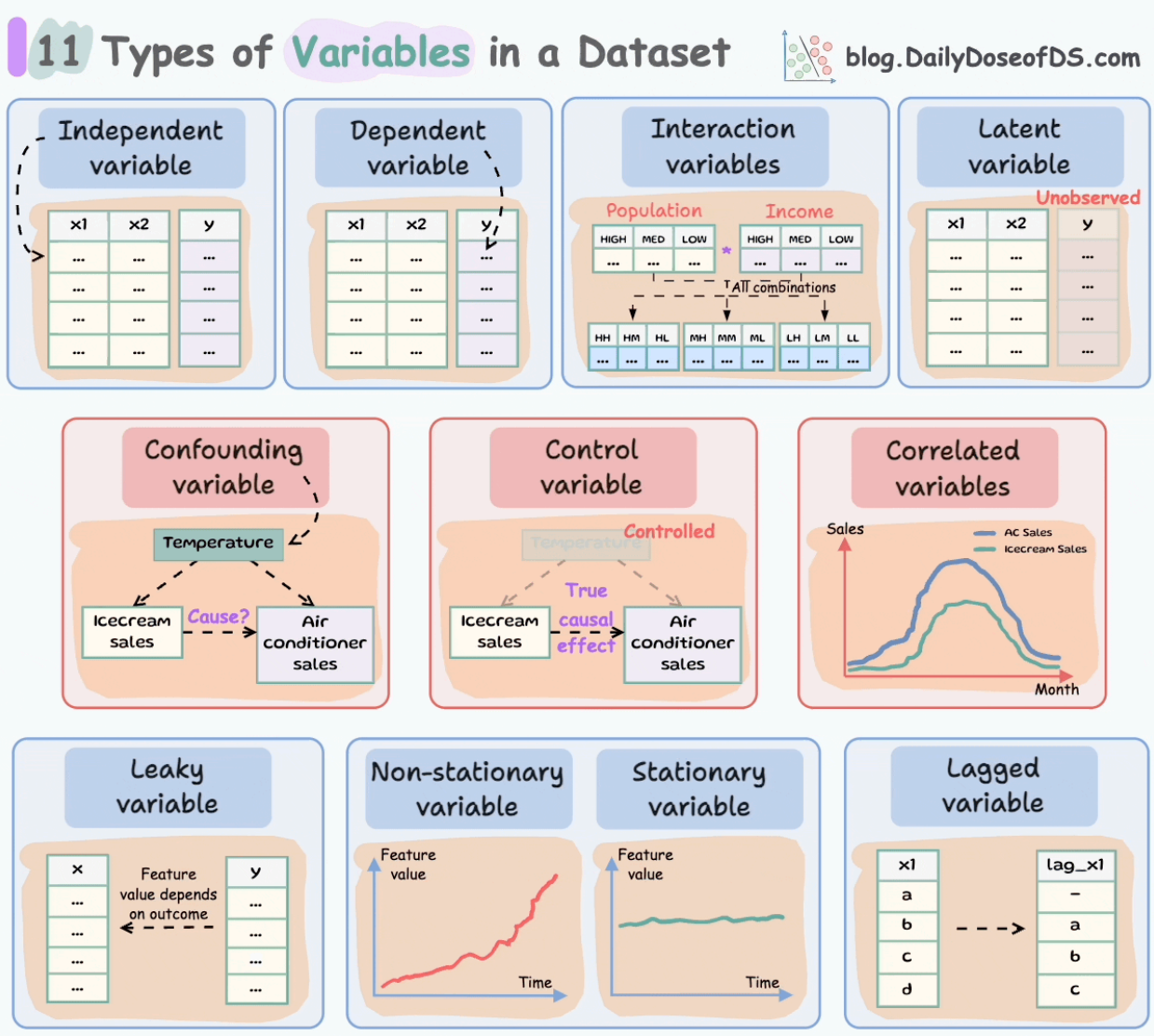
Instead, we use the probability density function to calculate the probability over an interval of values.

Feature Definition, Engineering and Selection

11 Types of Variables in a Dataset

In any tabular dataset, we typically categorize the columns as either a feature or a target.

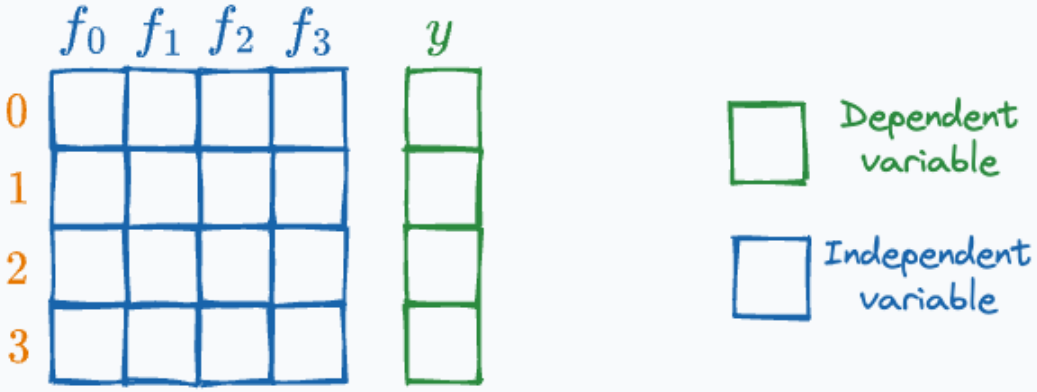
However, there are so many variables that one may find/define in their dataset, which I want to discuss in this chapter. These are depicted in the image below:



#1-2) Independent and dependent variables

These are the most common and fundamental to ML.

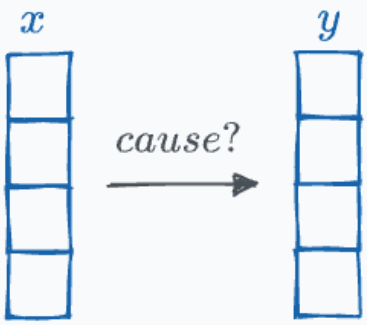
Independent variables are the features that are used as input to predict the outcome. They are also referred to as predictors/features/explanatory variables.



The dependent variable is the outcome that is being predicted. It is also called the target, response, or output variable.

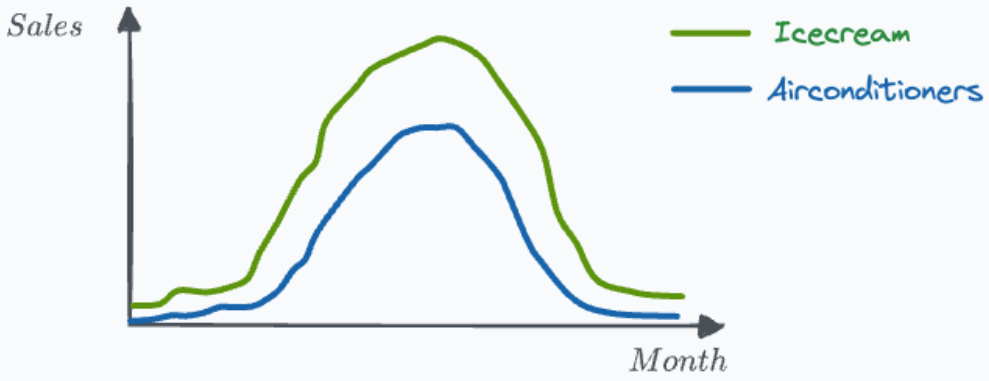
#3-4) Confounding and correlated variables

Confounding variables are typically found in a cause-and-effect study (causal inference).



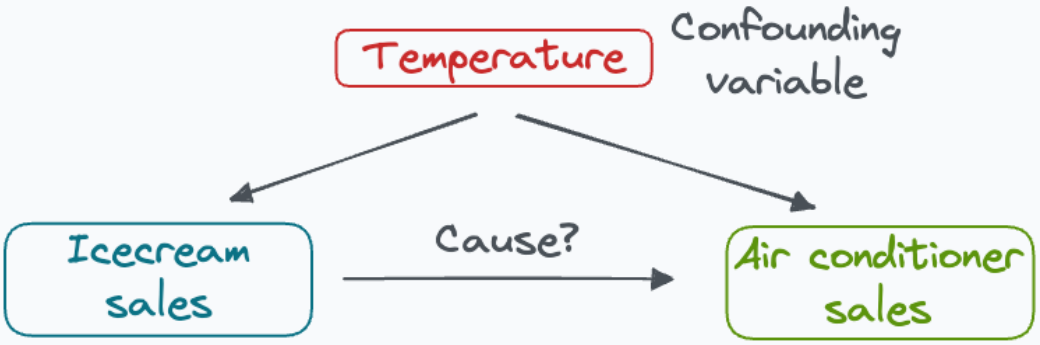
These variables are not of primary interest in the cause-and-effect equation but can potentially lead to spurious associations.

To exemplify, say we want to measure the effect of ice cream sales on the sales of air conditioners.



As you may have guessed, these two measurements are highly correlated.

However, there's a confounding variable — temperature, which influences both ice cream sales and the sales of air conditioners.



To study the true casual impact, it is essential to consider the confounder (temperature). Otherwise, the study will produce misleading results.

In fact, it is due to the confounding variables that we hear the statement: "Correlation does not imply causation."

In the above example:

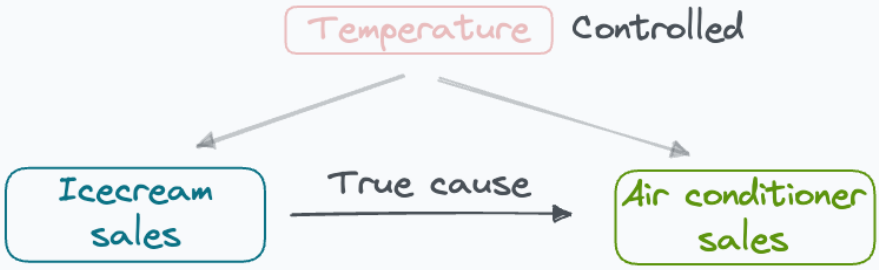
- There is a high correlation between ice cream sales and sales of air conditioners.
- But the sales of air conditioners (effect) are NOT caused by ice cream sales.

Also, in this case, the air conditioner and ice cream sales are correlated variables.

More formally, a change in one variable is associated with a change in another.

#5) Control variables

In the above example, to measure the true effect of ice cream sales on air conditioner sales, we must ensure that the temperature remains unchanged throughout the study.



Once controlled, temperature becomes a control variable.

More formally, these are variables that are not the primary focus of the study but are crucial to account for to ensure that the effect we intend to measure is not biased or confounded by other factors.

#6) Latent variables

A variable that is not directly observed but is inferred from other observed variables.

For instance, we use clustering algorithms because the true labels do not exist, and we want to infer them somehow.

The true label is a latent variable in this case.

Another common example of a latent variable is "intelligence."

Intelligence itself cannot be directly measured; it is a latent variable.

However, we can infer intelligence through various observable indicators such as test scores, problem-solving abilities, and memory retention.

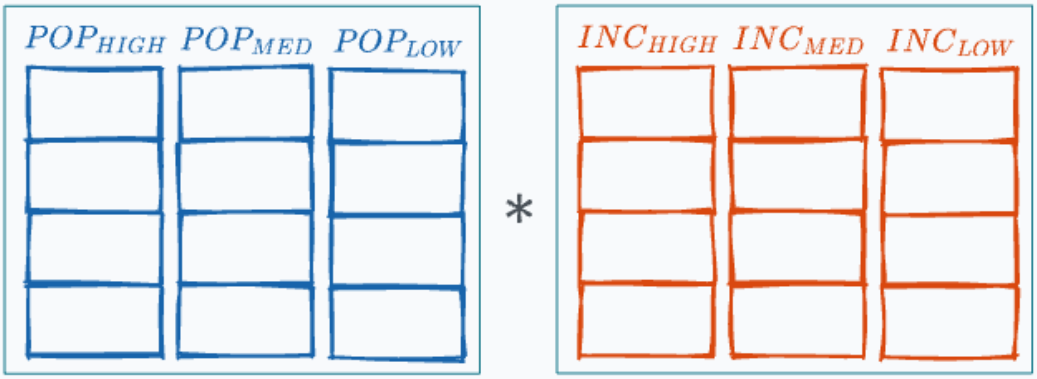
#7) Interaction variables

As the name suggests, these variables represent the interaction effect between two or more variables, and are often used in regression analysis.

Here's an instance I remember using them in.

In a project, I studied the impact of population density and income levels on spending behavior.

- I created three groups for population density — HIGH, MEDIUM, and LOW (one-hot encoded).
- Likewise, I created three groups for income levels — HIGH, MEDIUM, and LOW (one-hot encoded).



To do regression analysis, I created interaction variables by cross-multiplying both one-hot columns.

This produced 9 interaction variables:

- Population-High and Income-High
- Population-High and Income-Med
- Population-High and Income-Low
- Population-Med and Income-High
- and so on...

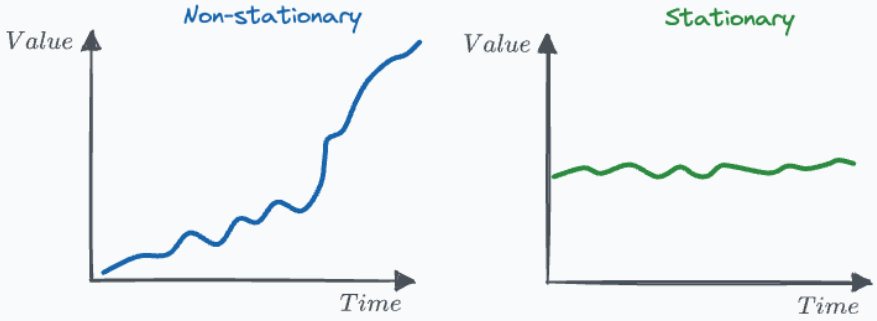
Conducting the regression analysis on interaction variables revealed more useful insights than what I observed without them.

To summarize, the core idea is to study two or more variables together rather than independently.

#8-9) Stationary and Non-Stationary variables:

The concept of stationarity often appears in time-series analysis.

Stationary variables are those whose statistical properties (mean, variance) DO NOT change over time.



On the flip side, if a variable's statistical properties change over time, they are called non-stationary variables.

Preserving stationarity in statistical learning is critical because these models are fundamentally reliant on the assumption that samples are identically distributed.

But if the probability distribution of variables is evolving over time, (non-stationary), the above assumption gets violated.

That is why, typically, using direct values of the non-stationary feature (like the absolute value of the stock price) is not recommended.

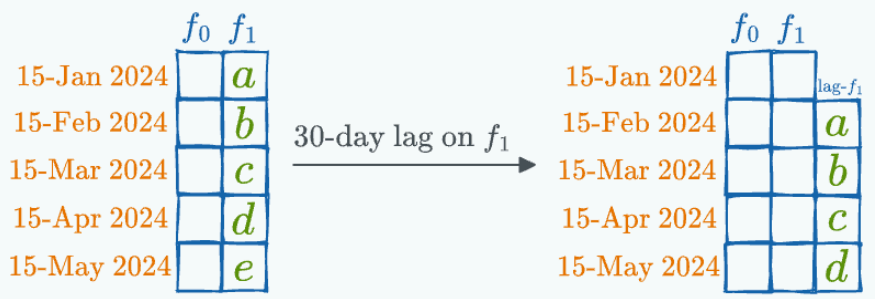
Instead, I have always found it better to define features in terms of relative changes:

$$\frac{\delta P}{P} \rightarrow \text{relative change in stock price}$$

#10) Lagged variables

Talking of time series, lagged variables are pretty commonly used in feature engineering and data analytics.

As the name suggests, a lagged variable represents previous time points' values of a given variable, essentially shifting the data series by a specified number of periods/rows.



For instance, when predicting next month's sales figures, we might include the sales figures from the previous month as a lagged variable.

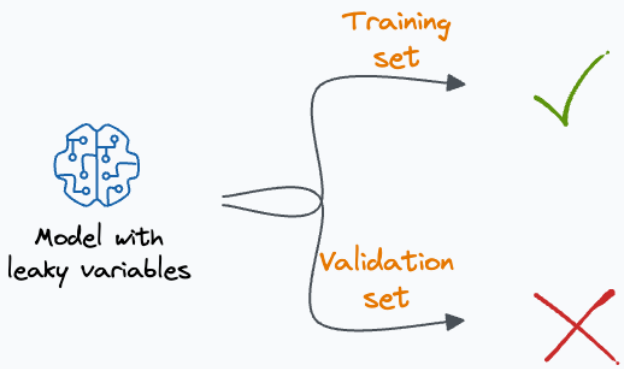
Lagged features may include:

- 7-day lag on website traffic to predict current website traffic.
- 30-day lag on stock prices to predict the next month's closing prices.
- And so on...

#11) Leaky variables

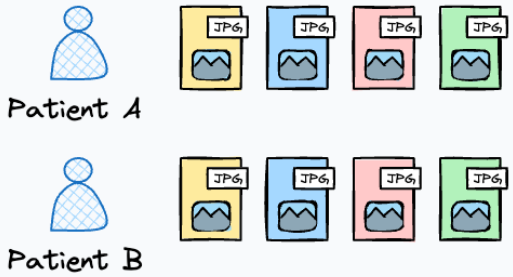
Yet again, as the name suggests, these variables (unintentionally) provide information about the target variable that would not be available at the time of prediction.

This leads to overly optimistic model performance during training but fails to generalize to new data.



Consider a dataset containing medical imaging data.

Each sample consists of multiple images (e.g., different views of the same patient's body part), and the model is intended to detect the severity of a disease.



In this case, randomly splitting the images into train and test sets will result in data leakage.

This is because images of the same patient will end up in both the training and test sets, allowing the model to "see" information from the same patient during training and testing.

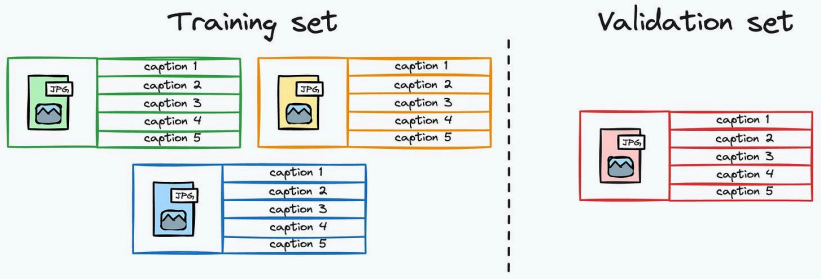
Here's a paper that committed this mistake (and later corrected it):

3.1. Training

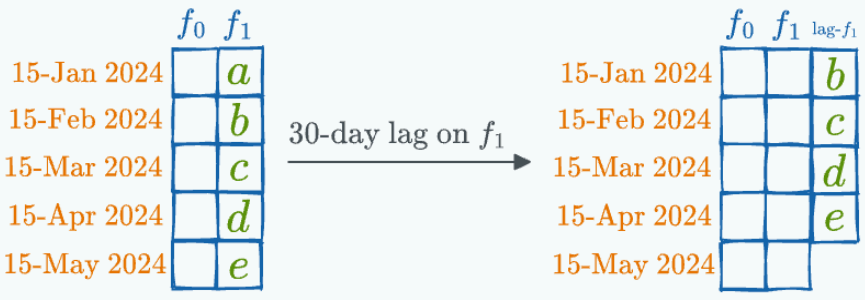
We use the ChestX-ray14 dataset released by Wang et al. (2017) which contains 112,120 frontal-view X-ray images of 30,805 unique patients. Wang et al. (2017) annotate each image with up to 14 different thoracic pathology labels using automatic extraction methods on radiology reports. We label images that have pneumonia as one of the annotated pathologies as positive examples and label all other images as negative examples for the pneumonia detection task. We randomly split the entire dataset into 80% training, and 20% validation.

To avoid this, a patient must only belong to the test or train/val set, not both.

This is called group splitting:



Creating forward-lag features is another way leaky variables get created unintentionally at times:



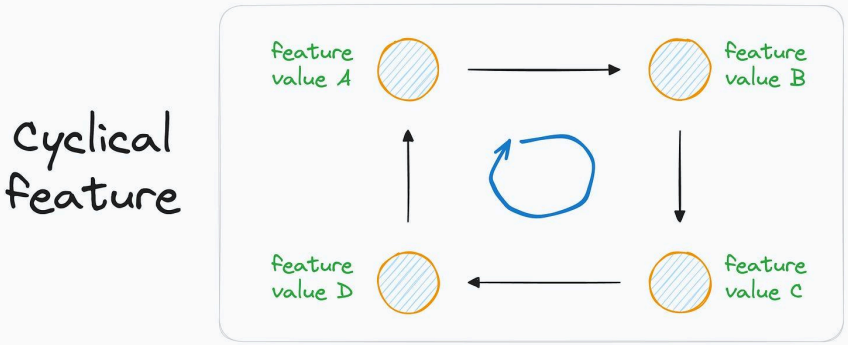
Let's get into more detail about the issue with random splitting below.

Cyclical feature encoding

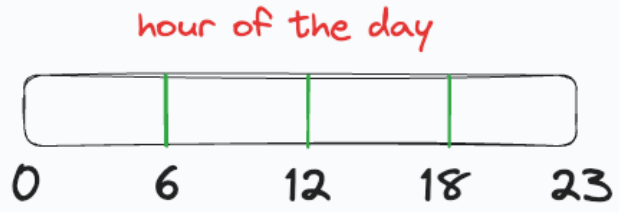
In typical machine learning datasets, we mostly find features that progress from one value to another: For instance:

- Numerical features like age, income, transaction amount, etc.
- Categorical features like t-shirt size, income groups, age groups, etc.

However, there is one more type of feature, which, in most cases, deserves special feature engineering effort but is often overlooked. These are cyclical features, i.e., features with a recurring pattern (or cycle).



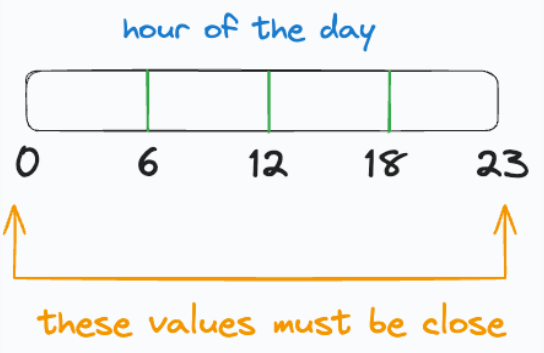
Unlike other features that progress continuously (or have no inherent order), cyclical features exhibit periodic behavior and repeat after a specific interval. For instance, the hour-of-the-day, the day-of-the-week, and the month-of-an-year are all common examples of cyclical features. Talking specifically about, say, the hour-of-the-day, its value can range between 0 to 23:



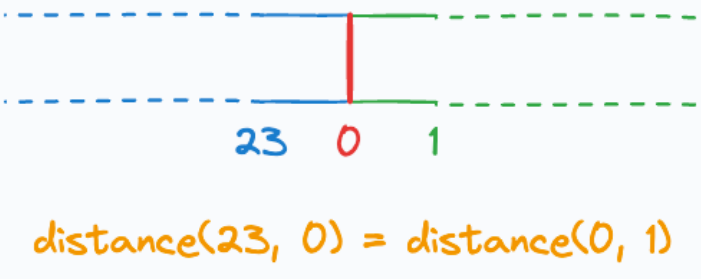
If we DON'T consider this as a cyclical feature and don't utilize appropriate feature engineering techniques, we will lose some really critical information.

To understand better, consider this:

Realistically speaking, the values "23" and "0" must be close to each other in our "ideal" feature representation of the hour-of-the-day.



Moreover, the distance between "0" and "1" must be the same as the distance between "23" and "0".



However, standard representation does not fulfill these properties. Thus, the value "23" is far from "0". In fact, the distance property isn't satisfied either.

Now, think about it for a second. Intuitively speaking, don't you think this feature deserves special feature engineering, i.e., one that preserves the inherent natural property?

I am sure you do!

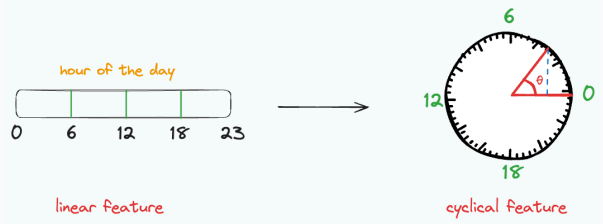
Let's understand how we typically do it.

Cyclical feature encoding

One of the most common techniques to encode such a feature is using trigonometric functions, specifically, sine and cosine. These are helpful because sine and cosine are periodic, bounded, and defined for all real values.

Of course, even other trigonometric functions are also periodic, but they are also undefined for some values, like, tan(pi/2).

For instance, consider representing the linear hour-of-the-day feature as a cyclical feature:



The central angle (2π) represents 24 hours. Thus, the linear feature values can be easily converted into cyclical features as follows:

```
import numpy as np

df['sin_hour'] = np.sin(df['hour'] * 2 * np.pi / 24)
df['cos_hour'] = np.cos(df['hour'] * 2 * np.pi / 24)
```

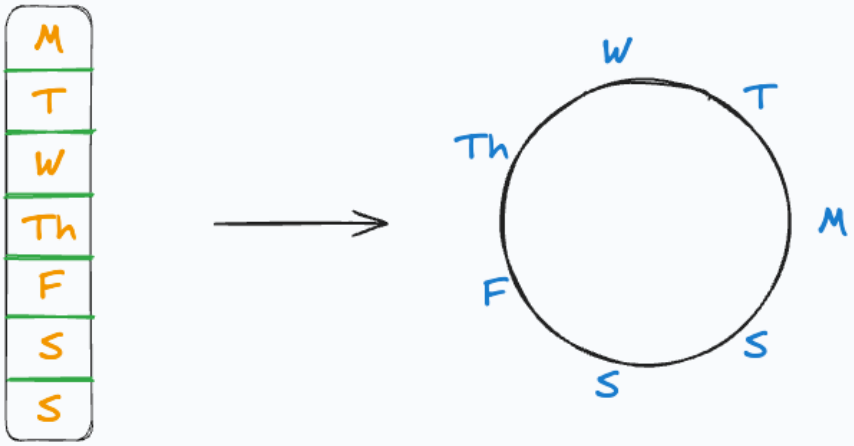
The benefit of doing this is how neatly the engineered feature satisfies the properties we discussed earlier:

$$|sin((0-23) * \frac{2\pi}{24})| = |sin((0-1) * \frac{2\pi}{24})|$$

$$|cos((0-23) * \frac{2\pi}{24})| = |cos((0-1) * \frac{2\pi}{24})|$$

As depicted above, the distance between the cyclical feature representation of "23" and "0" is the same as the distance between "0" and "1". The standard linear representation of the hour-of-the-day feature, however, violates this property, which results in loss of information...

...or rather, I should say that the standard linear representation of the hour-of-the-day feature results in an underutilization of information, which the model can benefit from. Had it been the day-of-the-week instead, the central angle (2π) must have represented 7 days.



The same idea can be extended to all sorts of cyclical features you may find in your dataset:

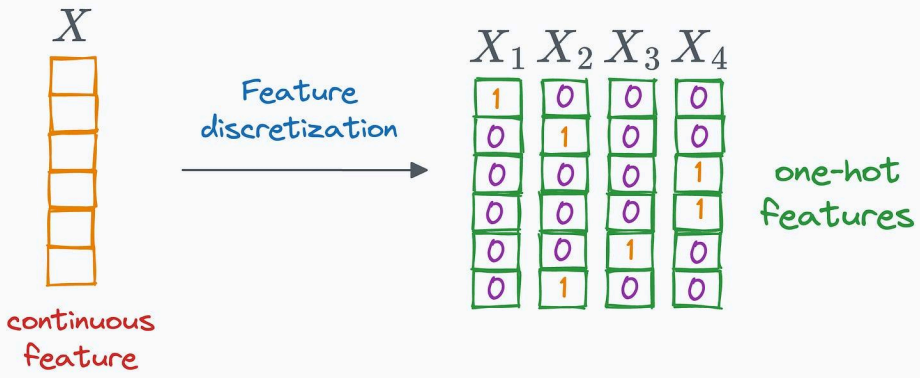
- Wind direction, if represented categorically, will go in this order: N, NE, E, SE, S, SW, W, NW, and then back to N.
- Phases of the moon, like new moon, first quarter, full moon, and last quarter, can be represented as categories with a cyclical order.
- Seasons, such as spring, summer, fall, and winter, are categorical features with a cyclical pattern as they repeat annually.

The point is that as you will inspect the dataset features, you will intuitively know which features are cyclical and which are not.

Typically, the model will find it easier to interpret the engineered features and utilize them in modeling the dataset accurately.

Feature Discretization

During model development, one of the techniques that many don't experiment with is feature discretization. As the name suggests, the idea behind discretization is to transform a continuous feature into discrete features.



Why, when, and how would you do that? Let's understand in this chapter.

Motivation

My rationale for using feature discretization has almost always been simple: "It just makes sense to discretize a feature."

For instance, consider your dataset has an age feature:

$$\hat{y} = w_1 * age + w_2 * x_2 + w_3 * x_3 + \ldots$$

In many use cases, like understanding spending behavior based on transaction history, such continuous variables are better understood when they are discretized into meaningful groups → youngsters, adults, and seniors.

For instance, say we model this transaction dataset without discretization. This would result in some coefficients for each feature, which would tell us the influence of each feature on the final prediction.

$$\hat{y} = w_1 * age + w_2 * x_2 + w_3 * x_3 + \ldots$$

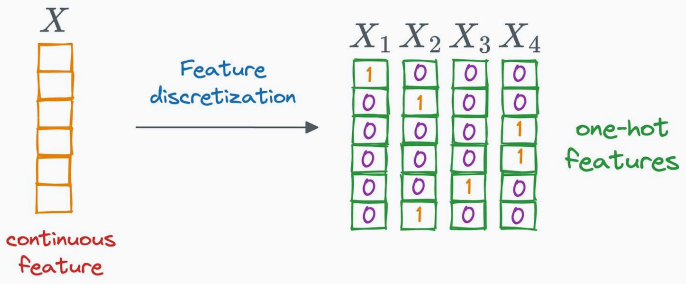
But if you think again, in our goal of understanding spending behavior, are we really interested in learning the correlation between exact age and spending behavior?

It makes very little sense to do that. Instead, it makes more sense to learn the correlation between different age groups and spending behavior.

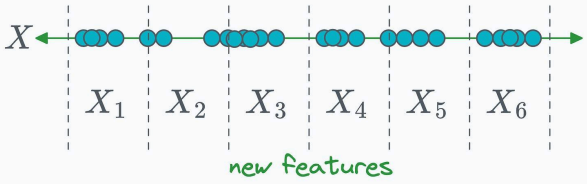
As a result, discretizing the age feature can potentially unveil much more valuable insights than using it as a raw feature.

2 common techniques

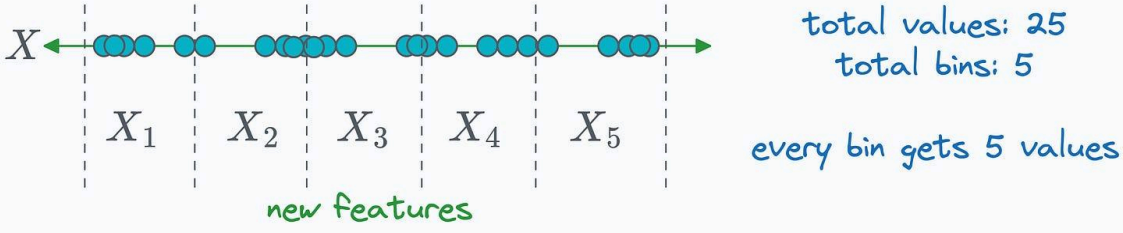
Now that we understand the rationale, there are 2 techniques that are widely preferred.



One way of discretizing features involves decomposing a feature into equally sized bins.

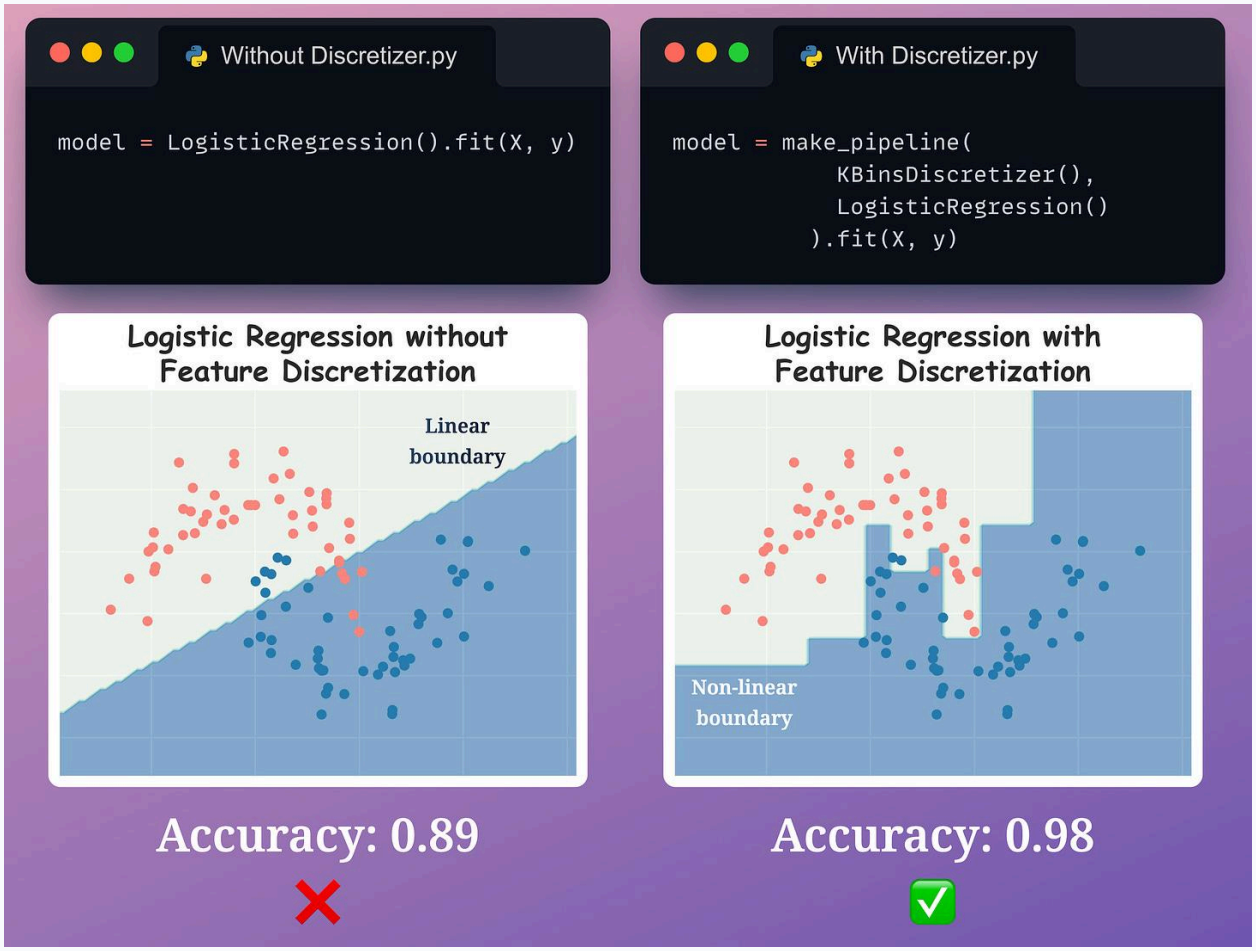


Another technique involves decomposing a feature into equal frequency bins:



After that, the discrete values are one-hot encoded.

One advantage of feature discretization is that it enables non-linear behavior even though the model is linear. This can potentially lead to better accuracy, which is also evident from the image below:



A linear model with feature discretization results in a:

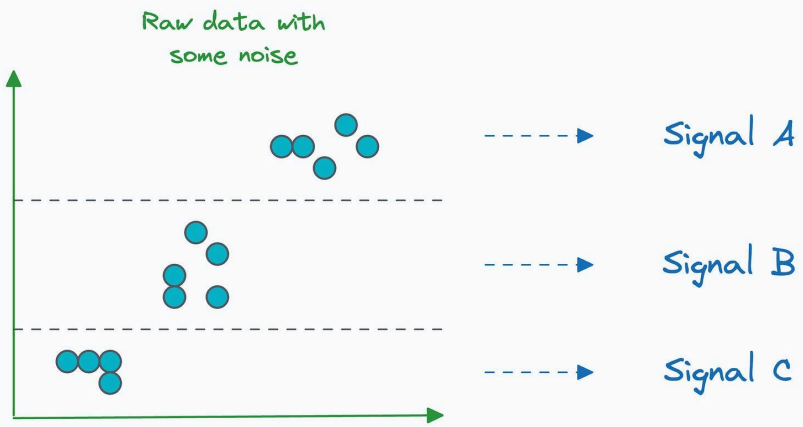
- non-linear decision boundary.
- better test accuracy.

So, in a way, we get to use a simple linear model but still get to learn non-linear patterns.

Another advantage of discretizing continuous features is that it helps us improve the signal-to-noise ratio.

Simply put, "signal" refers to the meaningful or valuable information in the data.

Binnng a feature helps us mitigate the influence of minor fluctuations, which are often mere noise.



Each bin acts as a means of "smoothing" out the noise within specific data segments.

Before I conclude, do remember that feature discretization with one-hot encoding increases the number of features → thereby increasing the data dimensionality.

And typically, as we progress towards higher dimensions, data become more easily linearly separable. Thus, feature discretization can lead to overfitting.

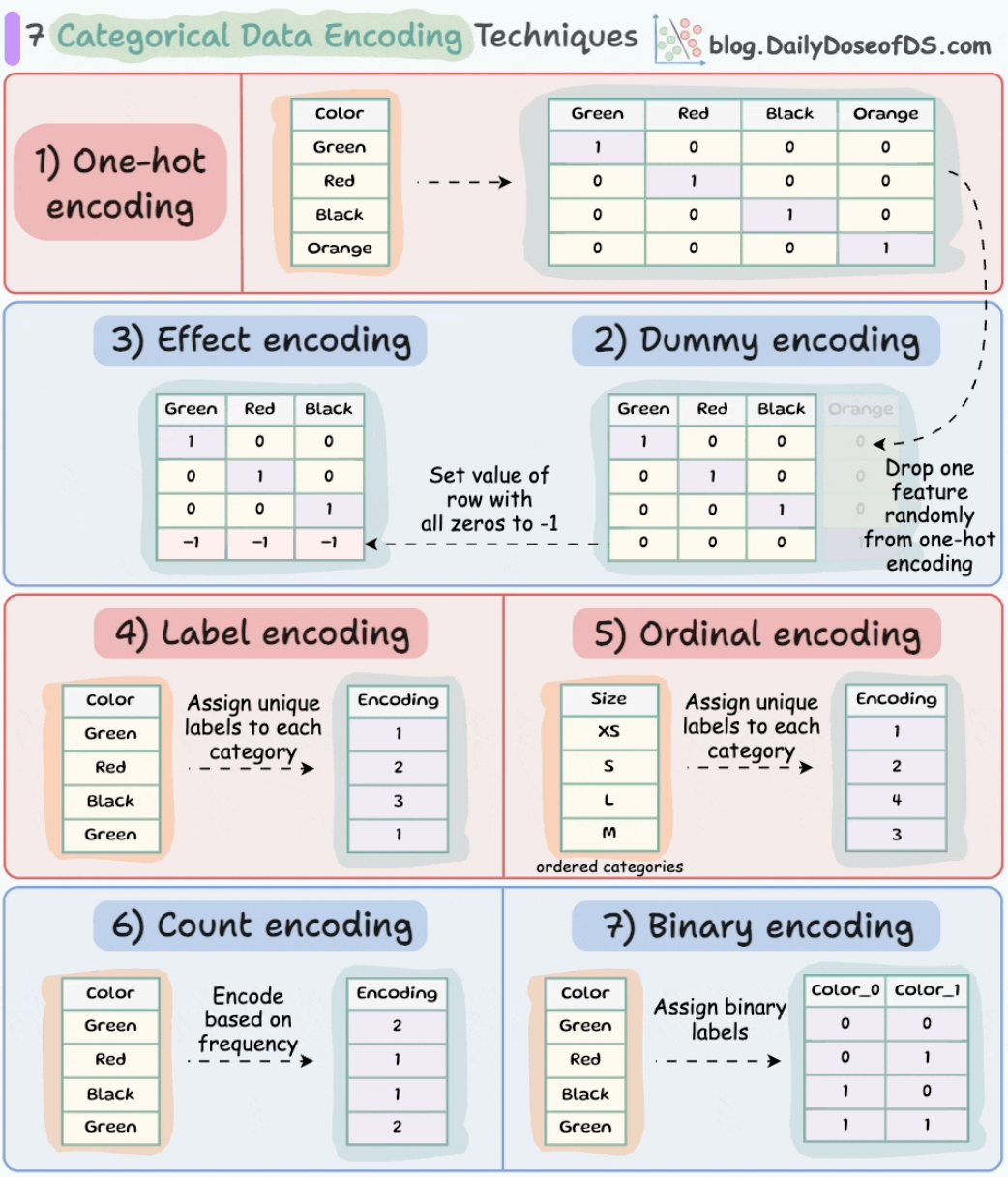
To avoid this, don't overly discretize all features. Instead, use it when it makes intuitive sense, as we saw earlier.

Of course, its utility can vastly vary from one application to another, but at times, I have found that:

- Discretizing geospatial data like latitude and longitude can be useful.
- Discretizing age/weight-related data can be useful.
- Features that are typically constrained between a range makes sense, like savings/income (practically speaking), etc.

7 Categorical Data Encoding Techniques

Here are 7 ways to encode categorical features:



We covered them in detail here: https://bit.ly/3LkfVq5.

One-hot encoding:

- Each category is represented by a binary vector of 0s and 1s.
- Each category gets its own binary feature, and only one of them is "hot" (set to 1) at a time, indicating the presence of that category.
- Number of features = Number of unique categorical labels

Dummy encoding:

- Same as one-hot encoding but with one additional step.
- After one-hot encoding, we drop a feature randomly.
- We do this to avoid the dummy variable trap (discussed in this chapter).
- Number of features = Number of unique categorical labels - 1

Effect encoding:

- Similar to dummy encoding but with one additional step.
- Alter the row with all zeros to -1.
- This ensures that the resulting binary features represent not only the presence or absence of specific categories but also the contrast between the reference category and the absence of any category.
- Number of features = Number of unique categorical labels - 1.

Label encoding:

- Assign each category a unique label.
- Label encoding introduces an inherent ordering between categories, which may not be the case.
- Number of features = 1.

Ordinal encoding:

- Similar to label encoding — assign a unique integer value to each category.
- The assigned values have an inherent order, meaning that one category is considered greater or smaller than another.
- Number of features = 1.

Count encoding:

- Also known as frequency encoding.
- Encodes categorical features based on the frequency of each category.
- Thus, instead of replacing the categories with numerical values or binary representations, count encoding directly assigns each category with its corresponding count.
- Number of features = 1.

Binary encoding:

- Combination of one-hot encoding and ordinal encoding.
- It represents categories as binary code.
- Each category is first assigned an ordinal value, and then that value is converted to binary code.
- The binary code is then split into separate binary features.
- Useful when dealing with high-cardinality categorical features (or a high number of features) as it reduces the dimensionality compared to one-hot encoding.
- Number of features = log(n) (in base 2).

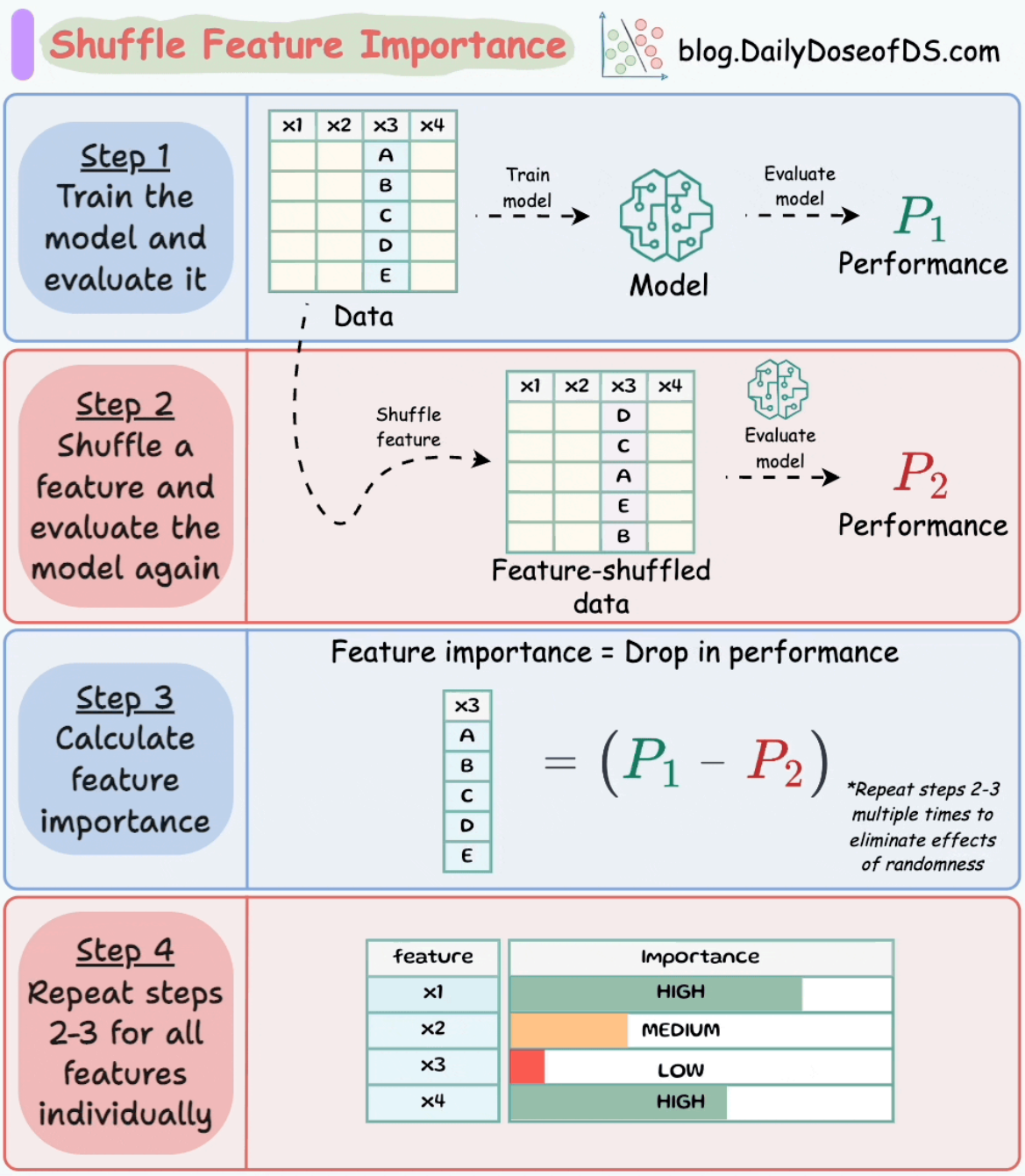
While these are some of the most popular techniques, do note that these are not the only techniques for encoding categorical data.

You can try plenty of techniques with the category-encoders library: https://pypi.org/project/category-encoders.

Shuffle Feature Importance

I often find "Shuffle Feature Importance" to be a handy and intuitive technique to measure feature importance.

As the name suggests, it observes how shuffling a feature influences the model performance. The visual below illustrates this technique in four simple steps:

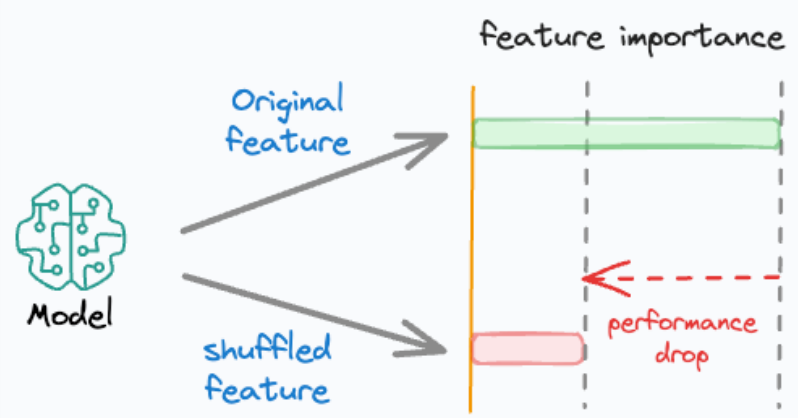


Here's how it works:

- Train the model and measure its performance → P1.
- Shuffle one feature randomly and measure performance again → P2 (model is NOT trained again).
- Measure feature importance using performance drop = (P1-P2).
- Repeat for all features.

This makes intuitive sense as well, doesn't it?

Simply put, if we randomly shuffle just one feature and everything else stays the same, then the performance drop will indicate how important that feature is.



- If the performance drop is low → This means the feature has a very low influence on the model's predictions.
- If the performance drop is high → This means that the feature has a very high influence on the model's predictions.

Do note that to eliminate any potential effects of randomness during feature shuffling, it is recommended to:

- Shuffle the same feature multiple times
- Measure average performance drop.

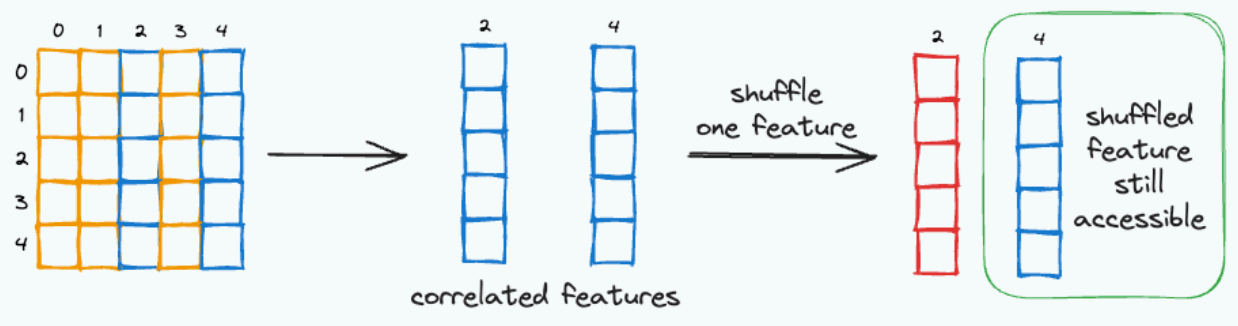
A few things that I love about this technique are:

- It requires no repetitive model training. Just train the model once and measure the feature importance.
- It is pretty simple to use and quite intuitive to interpret.
- This technique can be used for all ML models that can be evaluated.

Of course, there is one caveat as well.

Say two features are highly correlated, and one of them is permuted/shuffled.

In this case, the model will still have access to the feature through its correlated feature.



This will result in a lower importance value for both features.

One way to handle this is to cluster highly correlated features and only keep one feature from each cluster.

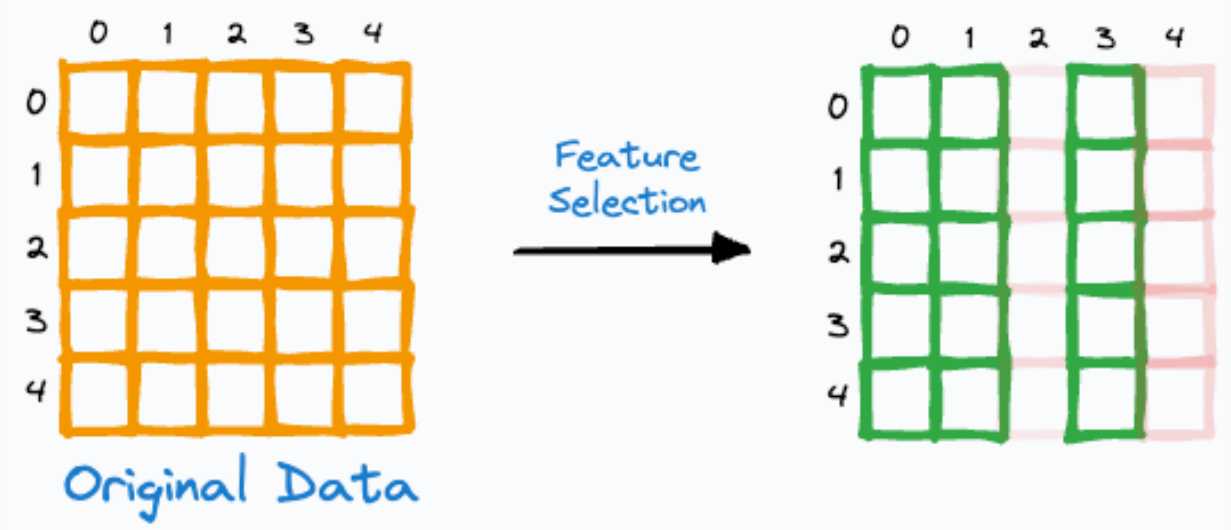
The Probe Method for Feature Selection

Real-world ML development is all about achieving a sweet balance between speed, model size, and performance.

One common way to:

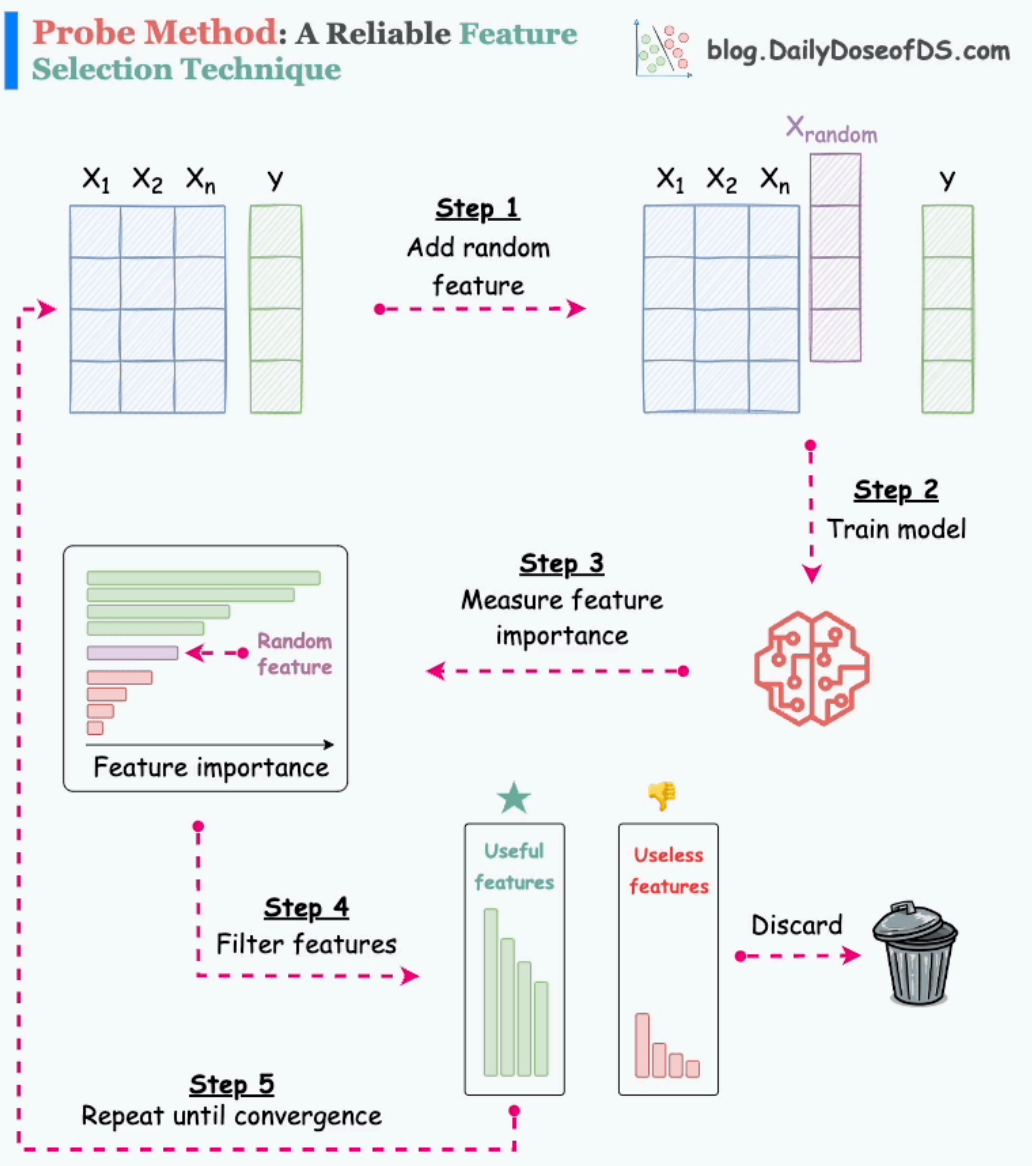
- Improve speed,
- Reduce size, and
- Maintain (or minimally degrade) performance...

...is by using featuring selection. The idea is to select the most useful subset of features from the dataset.



While there are many many methods for feature selection, I have often found the "Probe Method" to be pretty reliable, practical and intuitive to use.

The image below depicts how it works:



- Step 1) Add a random feature (noise).
- Step 2) Train a model on the new dataset.
- Step 3) Measure feature importance (can use shuffle feature importance)
- Step 4) Discard original features that rank below the random feature.
- Step 5) Repeat until convergence.

This whole idea makes intuitive sense as well.

More specifically, if a feature's importance is ranked below a random feature, it is probably a useless feature for the model.

This can be especially useful in cases where we have plenty of features, and we wish to discard those that don't contribute to the model.

Of course, one shortcoming is that when using the Probe Method, we must train multiple models:

1. Train the first model with the random feature and discard useless features.
2. Keep training new models until the random feature is ranked as the least important feature (although typically, convergence does not result in plenty of models).
3. Train the final model without the random feature.

Nonetheless, the approach can be quite useful to reduce model complexity.

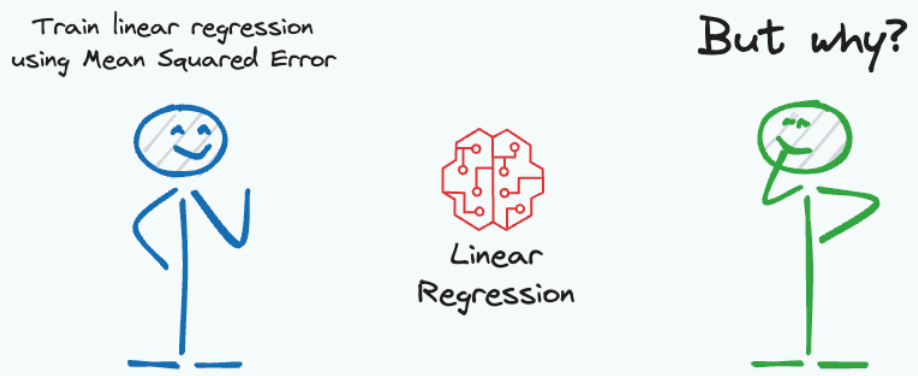
Regression

Why Mean Squared Error (MSE)?

Say you wish to train a linear regression model. We know that we train it by minimizing the squared error:

$$\text{Squared Error} = \sum_{i}^{n} \frac{(y_i - \theta^T x_i)^2}{n}$$

But have you ever wondered why we specifically use the squared error?



See, many functions can potentially minimize the difference between observed and predicted values. But of all the possible choices, what is so special about the squared error?

In my experience, people often say:

- Squared error is differentiable. That is why we use it as a loss function. WRONG.
- It is better than using absolute error as squared error penalizes large errors more. WRONG.

Sadly, each of these explanations are incorrect.

But approaching it from a probabilistic perspective helps us understand why the squared error is the ideal choice.

Let's understand in this chapter.

In linear regression, we predict our target variable y using the inputs X as follows:

$$y_i = \theta^T x_i + \epsilon_i$$

Here, epsilon is an error term that captures the random noise for a specific data point (i).

We assume the noise is drawn from a Gaussian distribution with zero mean based on the central limit theorem:

$$\epsilon \sim N(0, \sigma^2)$$

Thus, the probability of observing the error term can be written as:

$$p(\epsilon_i) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{\epsilon_i^2}{2\sigma^2})$$

Substituting the error term from the linear regression equation, we get:

$$\epsilon_i = \theta^T x_i - y_i$$

$$p(y_i|x_i;\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y_i - \theta^T x_i)^2}{2\sigma^2})$$

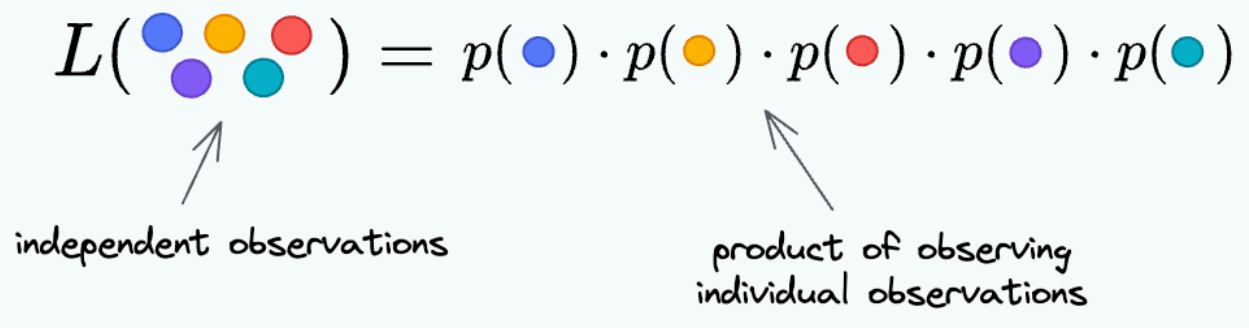
For a specific set of parameters θ, the above tells us the probability of observing a data point (i).

Next, we can define the likelihood function as follows:

$$L(\theta) = p(y|X;\theta)$$

It means that by varying θ, we can fit a distribution to the observed data and quantify the likelihood of observing it.

We further write it as a product for individual data points because we assume all observations are independent.



Thus, we get:

$$\begin{aligned} L(\theta) &= \prod_{i}^{n} p(y_i|x_i;\theta) \\ &= \prod_{i}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y_i - \theta^T x_i)^2}{2\sigma^2}) \end{aligned}$$

Since the log function is monotonic, we use the log-likelihood and maximize it. This is called maximum likelihood estimation (MLE).

$$\begin{aligned} log(L(\theta)) &= log(\prod_{i}^{n} p(y_i|x_i;\theta)) \\ &= log(\prod_{i}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y_i - \theta^T x_i)^2}{2\sigma^2})) \end{aligned}$$

Simplifying, we get:

$$log(L(\theta)) = \sum_{i}^{n} log(\frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y_i - \theta^T x_i)^2}{2\sigma^2}))$$

distribute the logarithm

$$= \sum_{i}^{n} log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{(y_i - \theta^T x_i)^2}{2\sigma^2}$$

distribute the summation

$$= \boxed{n \cdot log(\frac{1}{\sqrt{2\pi\sigma^2}})} - \boxed{\sum_{i}^{n} \frac{(y_i - \theta^T x_i)^2}{2\sigma^2}}$$

constant

To reiterate, the objective is to find the θ that maximizes the above expression. But the first term is independent of θ. Thus, maximizing the above expression is equivalent to minimizing the second term. And if you notice closely, it's precisely the squared error.

$$\frac{1}{2\sigma^2} \sum_{i}^{n} (y_i - \theta^T x_i)^2$$

Thus, you can maximize the log-likelihood by minimizing the squared error. And this is the origin of least-squares in linear regression. See, there's clear proof and reasoning behind using squared error as a loss function in linear regression.

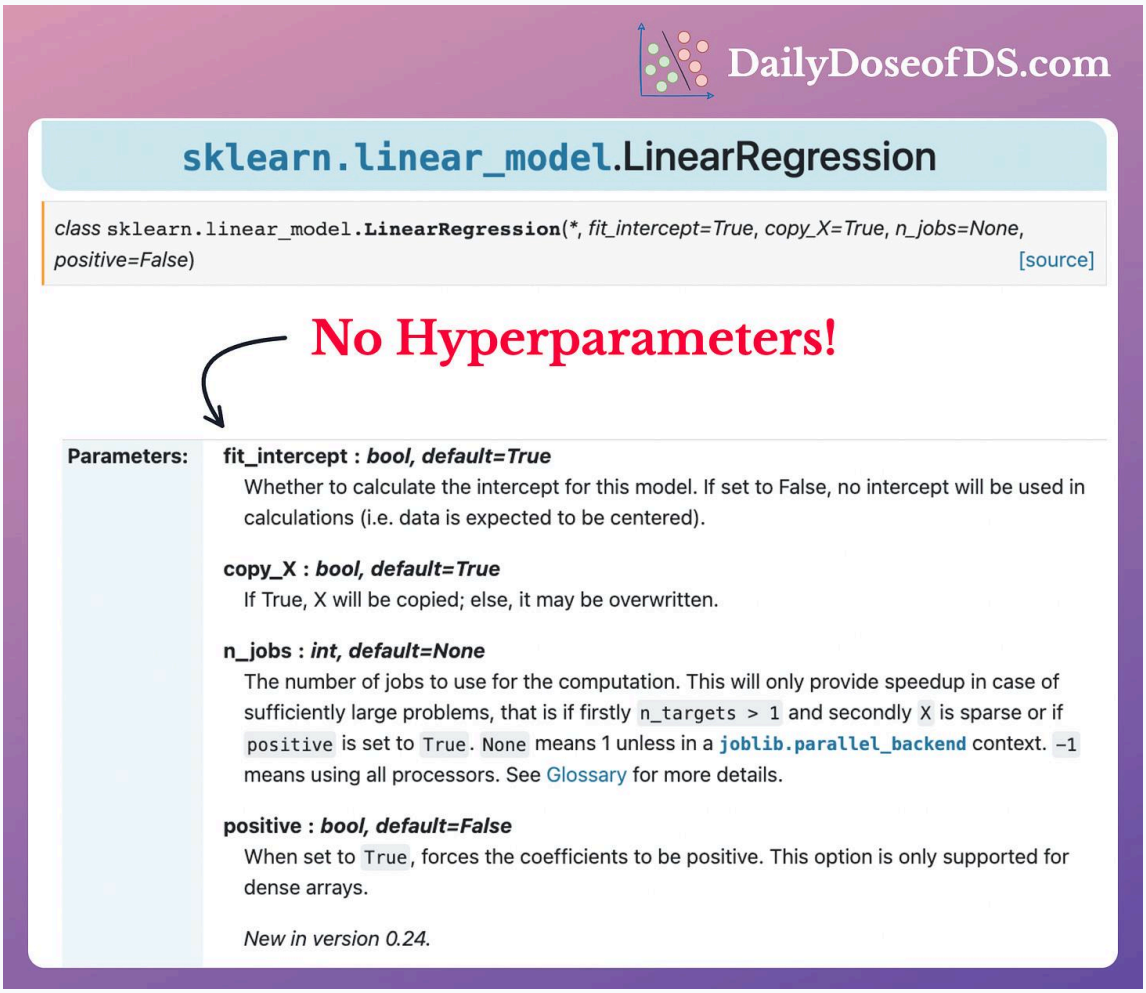
Nothing comes from thin air in machine learning.

Sklearn Linear Regression Has No Hyperparameters

Almost all ML models we work with have some hyperparameters, such as:

- Learning rate
- Regularization
- Layer size (for neural network), etc.

But as shown in the image below, why don't we see any hyperparameter in Sklearn's Linear Regression implementation?

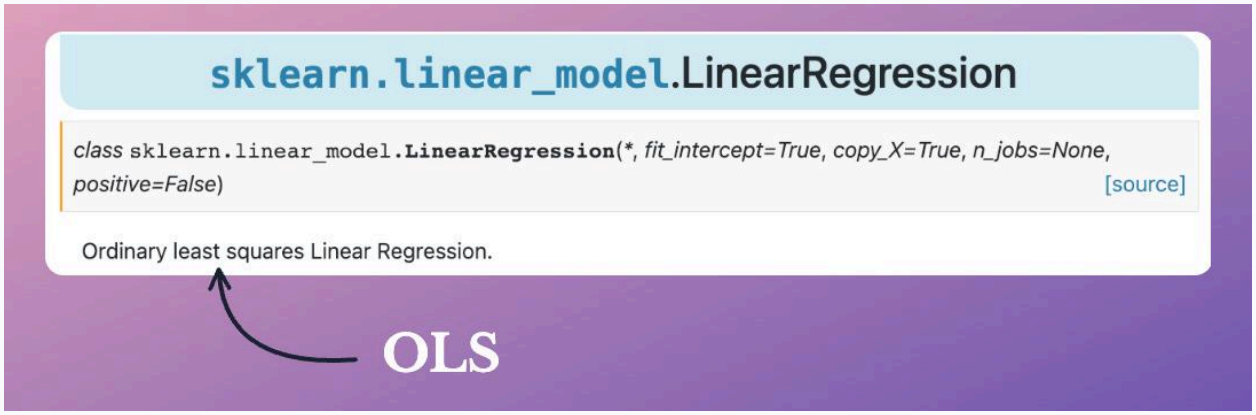


It must have learning rate as a hyperparameter, right?

To understand the reason why it has no hyperparameters, we first need to learn that the Linear Regression can model data in two different ways:

1. Gradient Descent (which many other ML algorithms use for optimization):
    - It is a stochastic algorithm, i.e., involves some randomness.
    - It finds an approximate solution using optimization.
    - It has hyperparameters.
2. Ordinary Least Square (OLS):
    - It is a deterministic algorithm. Thus, if run multiple times, it will always converge to the same weights.
    - It always finds the optimal solution.
    - It has no hyperparameters.

Now, instead of the typical gradient descent approach, Sklearn's Linear Regression class implements the OLS method.



That is why it has no hyperparameters.

How does OLS work?

With OLS, the idea is to find the set of parameters (Θ) such that:

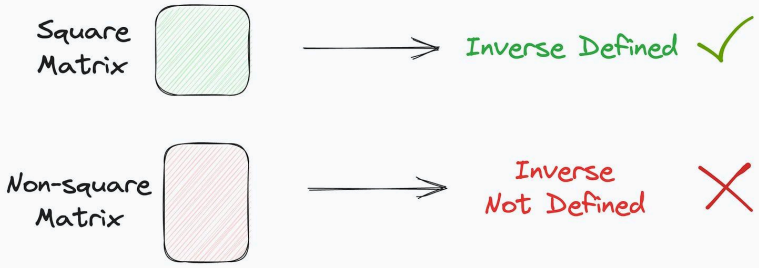
$$X\theta = Y$$

- X: input data with dimensions (n,m).
- Θ: parameters with dimensions (m,1).
- y: output data with dimensions (n,1).
- n: number of samples.
- m: number of features.

One way to determine the parameter matrix Θ is by multiplying both sides of the equation with the inverse of X, as shown below:

$$\theta = X^{-1}Y$$

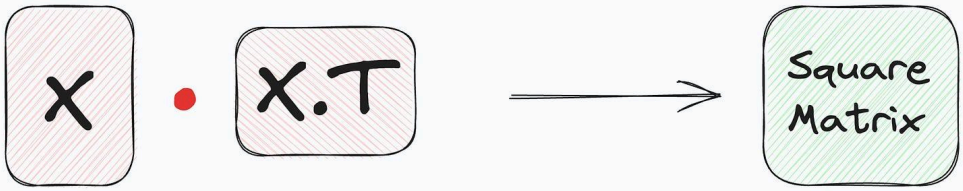
But because X might be a non-square matrix, its inverse may not be defined.



To resolve this, first, we multiply with the transpose of X on both sides, as shown below:

$$X^T X\theta = X^T y$$

This makes the product of X with its transpose a square matrix.



The obtained matrix, being square, can be inverted (provided it is non-singular).

Next, we take the collective inverse of the product to get the following:

$$\theta = (X^T X)^{-1} X^T y$$

It's clear that the above definition has:

- No hyperparameters.
- No randomness. Thus, it will always return the same solution, which is also optimal.

This is precisely what the Linear Regression class of Sklearn implements.

To summarize, it uses the OLS method instead of gradient descent.

That is why it has no hyperparameters.

Of course, do note that there is a significant tradeoff between run time and convenience when using OLS vs. gradient descent.

This is also clear from the time-complexity table we discussed in an earlier chapter.

As depicted above, the run-time of OLS is cubically related to the number of features (m).

Thus, when we have many features, it may not be a good idea to use the `LinearRegression()` class. Instead, use the `SGDRegressor()` class from Sklearn.

Of course, the good thing about LinearRegression() class is that it involves no hyperparameter tuning.

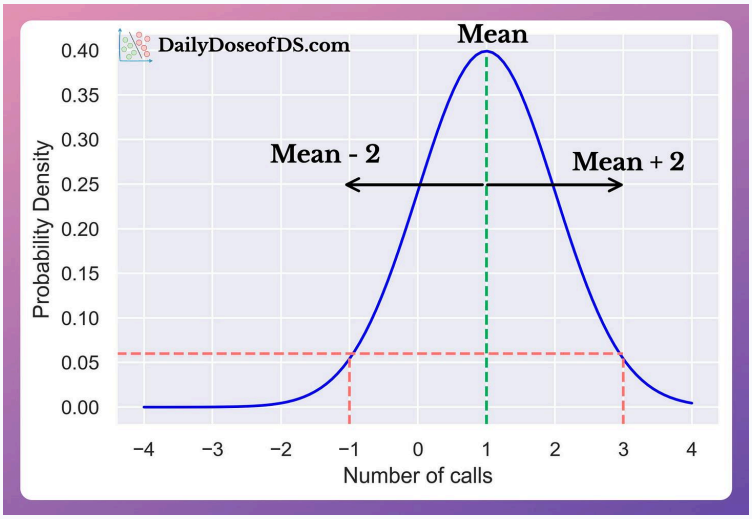
Thus, when we use OLS, we trade run-time for finding an optimal solution without hyperparameter tuning.

Poisson Regression vs. Linear Regression

Linear regression comes with its own set of challenges/assumptions. For instance, after modeling, the output can be negative for some inputs.

But this may not make sense at times — predicting the number of goals scored, number of calls received, etc. Thus, it is clear that it cannot model count (or discrete) data.

Furthermore, in linear regression, residuals are expected to be normally distributed around the mean. So the outcomes on either side of the mean (m-x, m+x) are equally likely.



For instance:

- if the expected number (mean) of calls received is 1...
- ...then, according to linear regression, receiving 3 calls (1+2) is just as likely as receiving -1 (1-2) calls. (This relates to the concept of prediction intervals, which we covered in the an earlier chapter.)
- But in this case, a negative prediction does not make any sense.

Thus, if the above assumptions do not hold, linear regression won't help.

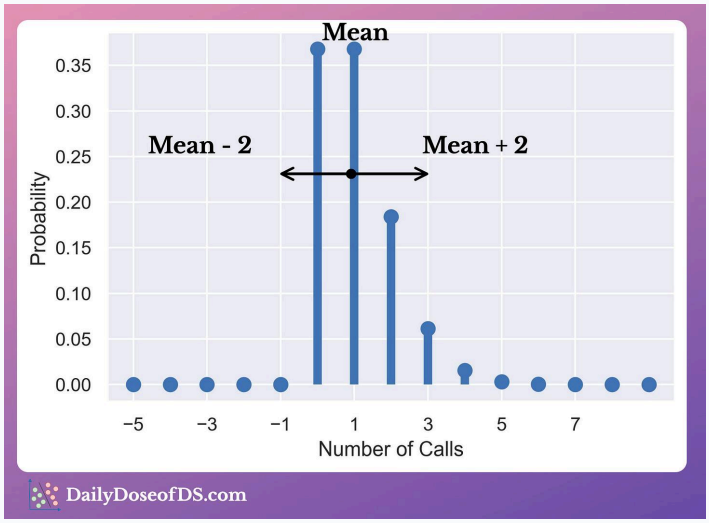
Instead, in this specific case, what you may need is Poisson regression.

Poisson regression is more suitable if your response (or outcome) is count-based. It assumes that the response comes from a Poisson distribution.

$$P(x) = \frac{e^{-\lambda}\lambda^{x}}{x!}$$

It is a type of generalized linear model (GLM) that is used to model count data. It works by estimating a Poisson distribution parameter (λ), which is directly linked to the expected number of events in a given interval.

Contrary to linear regression, in Poisson regression, residuals may follow an asymmetric distribution around the mean (λ). Hence, outcomes on either side of the mean (λ-x, λ+x) are NOT equally likely.



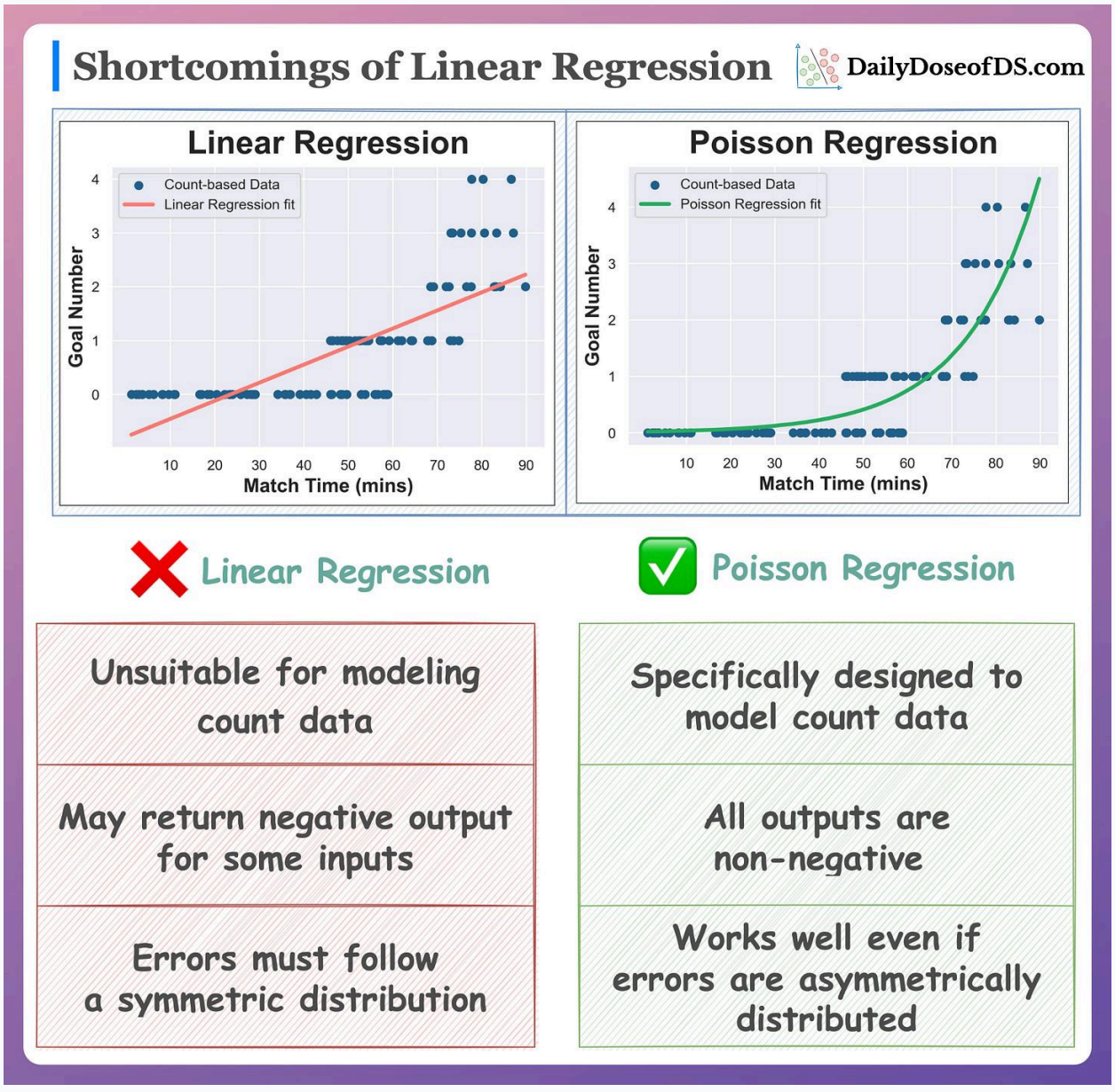
For instance:

- if the expected number (mean) of calls received is 1...
- ...then, according to Poisson regression, it is possible to receive 3 (1+2) calls, but it is impossible to receive -1 (1-2) calls.
- This is because its outcome is also non-negative.

The regression fit is mathematically defined as follows:

$$ln(E(Y|X)) = w_1X_1 + w_2X_2 + \cdots + w_nX_n$$

The following visual neatly summarizes this post:



We shall continue this discussion in the next chapter.

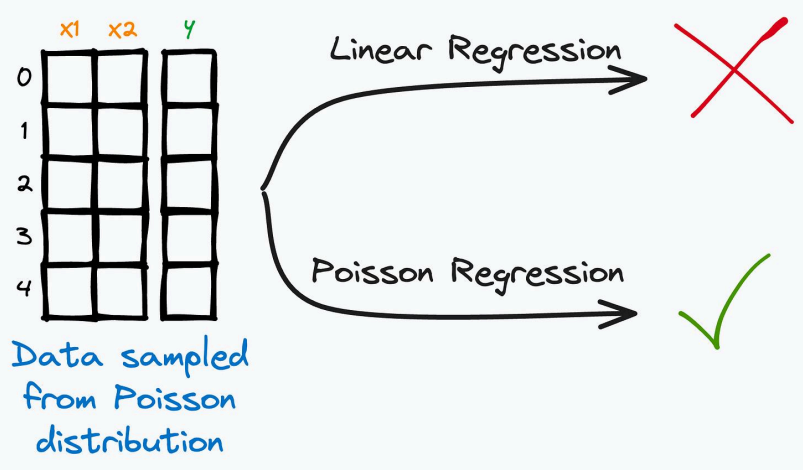
How to Build Linear Models?

In this chapter, I will help you cultivate what I think is one of the MOST overlooked and underappreciated skills in developing linear models.

I can guarantee that harnessing this skill will give you a lot of clarity and intuition in the modeling stages.

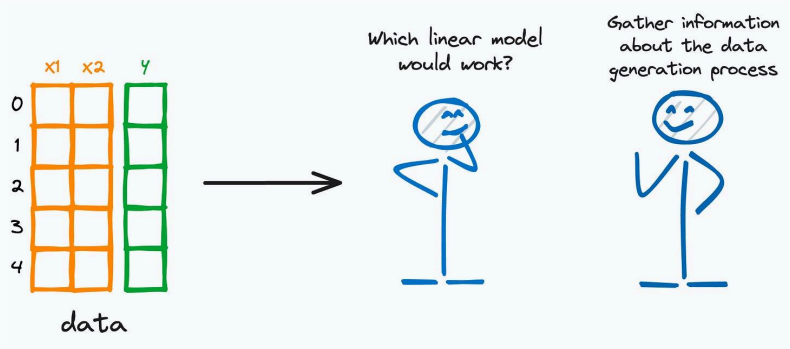
To begin, understand that the Poisson regression we disussed above is no magic.

It's just that, in our specific use case, the data generation process didn't perfectly align with what linear regression is designed to handle. In other words, earlier when we trained a linear regression model, we inherently assumed that the data was sampled from a normal distribution. But that was not true in this Poisson regression case.

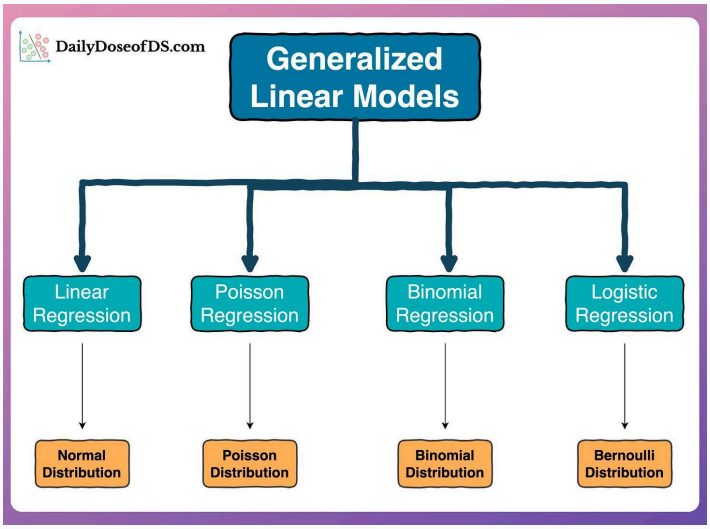


Instead, it came from a Poisson distribution, which is why Poisson regression worked better. Thus, the takeaway is that whenever you train linear models, always always and always think about the data generation process. It goes like this:

- Okay, I have this data.
- I want to fit a linear model through it.
- What information do I get about the data generation process that can help me select an appropriate linear model?



You'd start appreciating the importance of data generation when you realize that literally every member of the generalized linear model family stems from altering the data generation process.

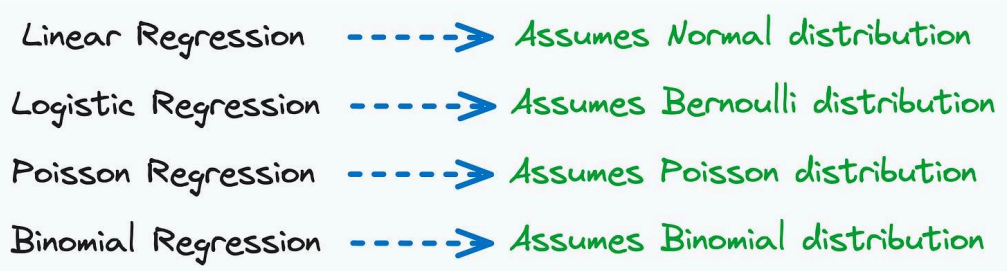


For instance:

- If the data generation process involves a Normal distribution → you get linear regression.
- If the data has only positive integers in the response variable, maybe it came from a Poisson distribution → and this gives us Poisson regression. This is precisely what we discussed yesterday.
- If the data has only two targets — 0 and 1, maybe it was generated using Bernoulli distribution → and this gives rise to logistic regression.
- If the data has finite and fixed categories (0, 1, 2,...n), then this hints towards Binomial distribution → and we get Binomial regression.

See...

Every linear model makes an assumption and is then derived from an underlying data generation process.



Thus, developing a habit of holding for a second and thinking about the data generation process will give you so much clarity in the modeling stages.

I am confident this will help you get rid of that annoying and helpless habit of relentlessly using a specific sklearn algorithm without truly knowing why you are using it.

Consequently, you'd know which algorithm to use and, most importantly, why.

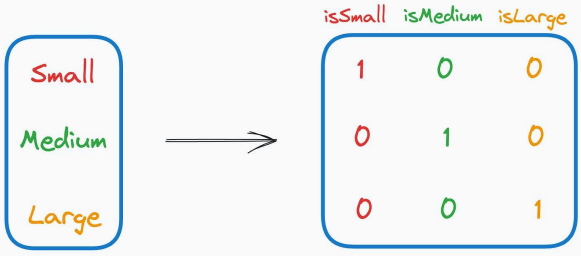
This improves your credibility as a data scientist and allows you to approach data science problems with intuition and clarity rather than hit-and-trial.

In fact, once you understand the data generation process, you will automatically get to know about most of the assumptions of that specific linear model.

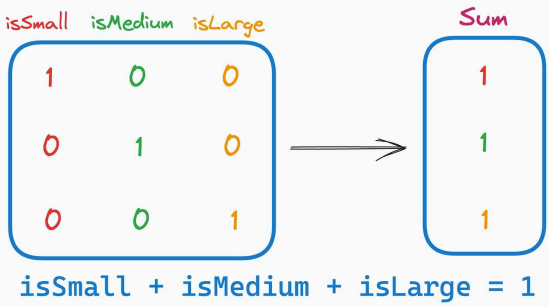
Dummy Variable Trap

With one-hot encoding, we introduce a big problem in the data.

When we one-hot encode categorical data, we unknowingly introduce perfect multicollinearity.



Multicollinearity arises when two or more features can predict another feature. In this case, as the sum of one-hot encoded features is always 1, it leads to perfect multicollinearity.

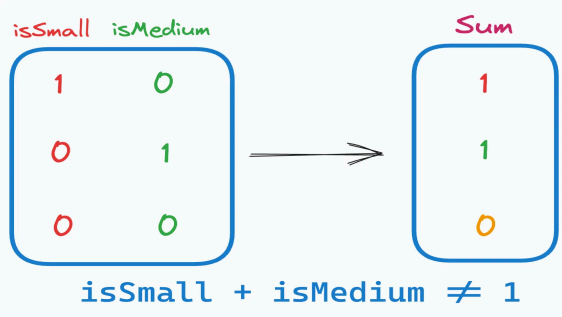


This is often called the Dummy Variable Trap. It is bad because the model has redundant features. Morevero, the regressions coefficients aren't reliable in the presence of multicollinearity.

So how to resolve this?

The solution is simple. Drop any arbitrary feature from the one-hot encoded features.

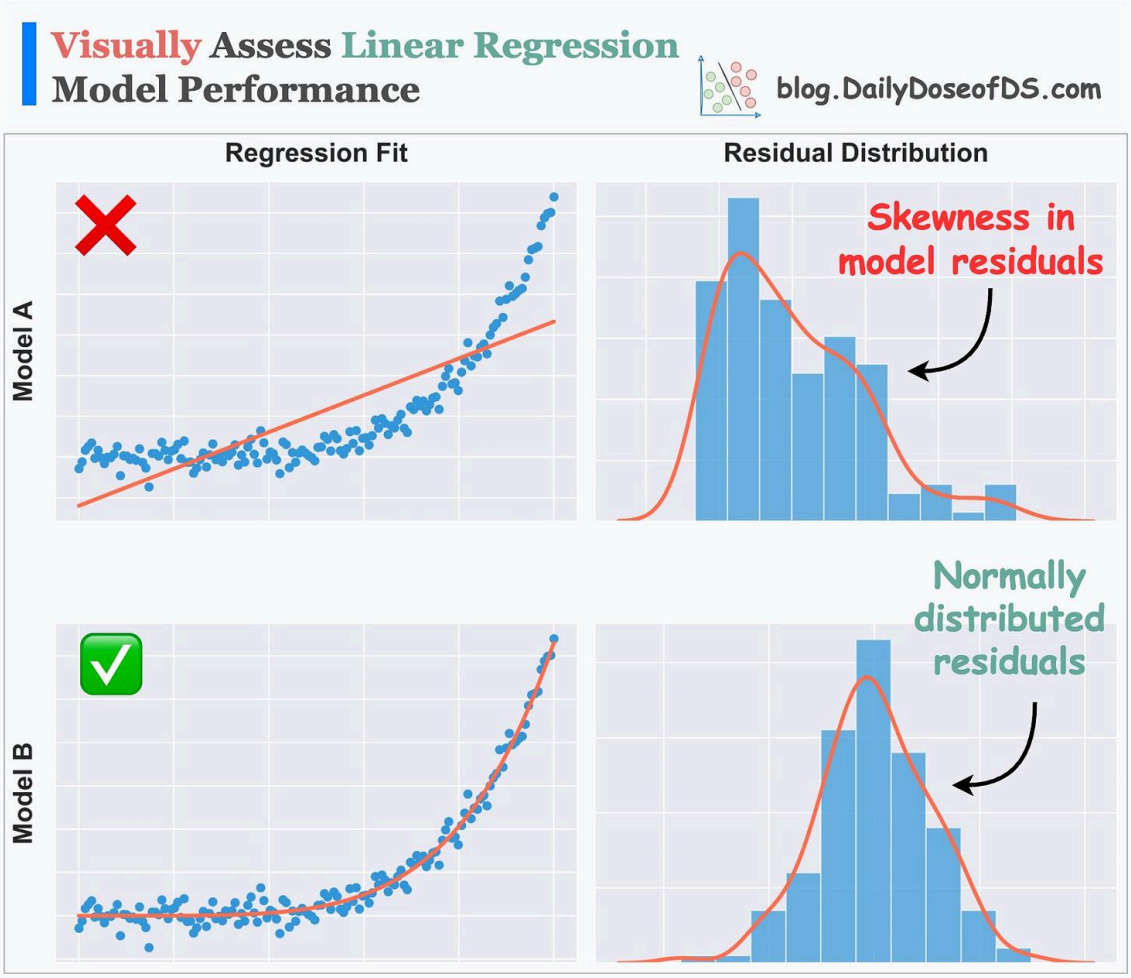
This instantly mitigates multicollinearity and breaks the linear relationship which existed before.



Visually Assess Linear Regression Performance

Linear regression assumes that the model residuals (=actual-predicted) are normally distributed. If the model is underperforming, it may be due to a violation of this assumption.

Here, I often use a residual distribution plot to verify this and determine the model's performance. As the name suggests, this plot depicts the distribution of residuals (=actual-predicted), as shown below:



A good residual plot will:

- Follow a normal distribution
- NOT reveal trends in residuals

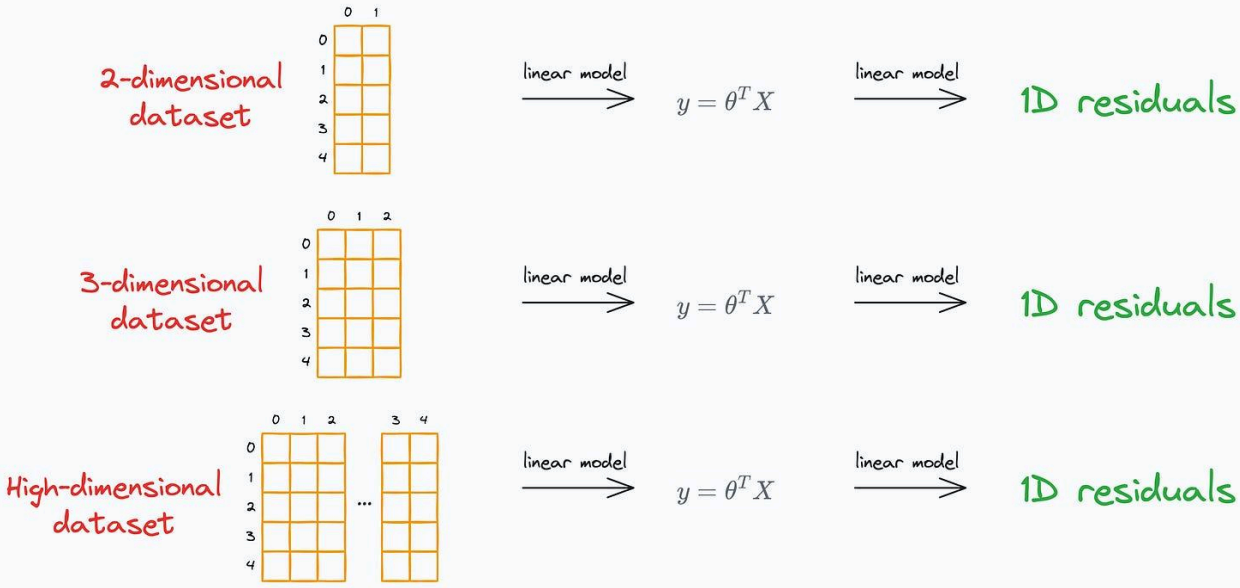
A bad residual plot will:

- Show skewness
- Reveal patterns in residuals

Thus, the more normally distributed the residual plot looks, the more confident we can be about our model. This is especially useful when the regression line is difficult to visualize, i.e., in a high-dimensional dataset.

Why?

Because a residual distribution plot depicts the distribution of residuals, which is always one-dimensional.



Thus, it can be plotted and visualized easily.

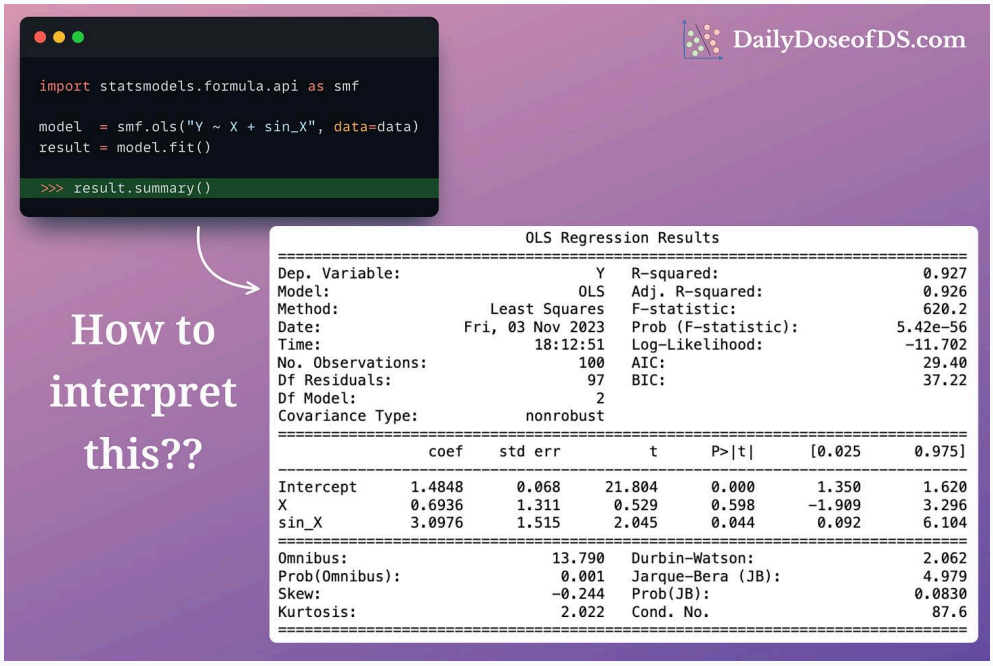
Of course, this was just about validating one assumption — the normality of residuals.

However, linear regression relies on many other assumptions, which must be tested as well.

Statsmodel provides a pretty comprehensive report for this, which we shall discuss in the next chapter.

Statsmodel Regression Summary

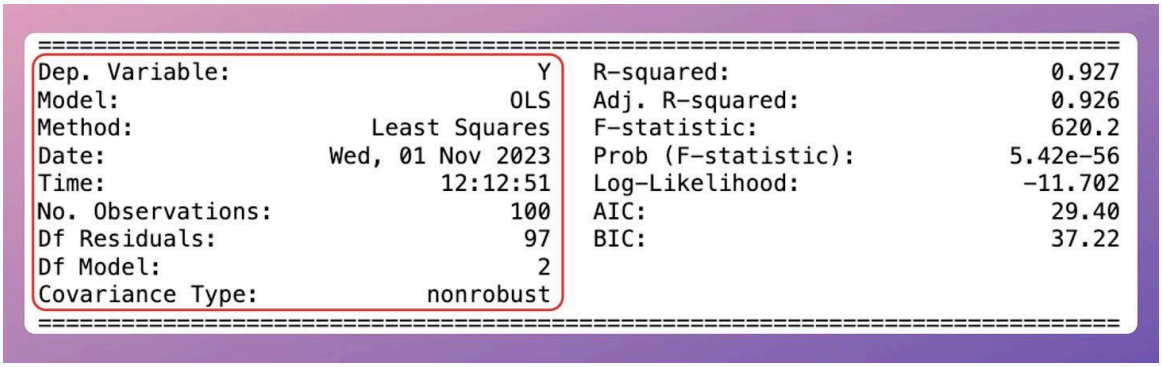
Statsmodel provides one of the most comprehensive summaries for regression analysis.



Yet, I have seen so many people struggling to interpret the critical model details mentioned in this report. In this chapter, let's understand the entire summary support provided by statsmodel and why it is so important.

Section #1

The first column of the first section lists the model's settings (or config). This part has nothing to do with the model's performance.



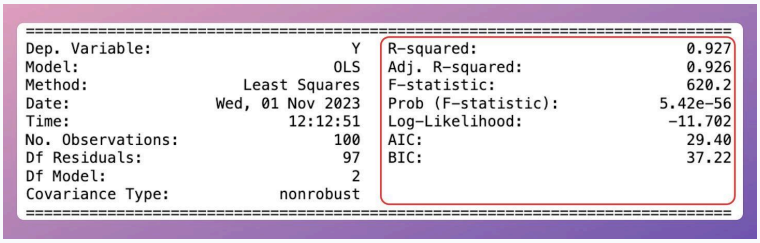
- Dependent variable: The variable we are predicting.
- Model and Method: We are using OLS to fit a linear model.
- Date and time: You know it.
- No. of observations: The dataset's size.
- Df residuals: The degrees of freedom associated with the residuals. It is essentially the number of data points minus the number of parameters estimated in the model (including intercept term).
- Df Model: This represents the degrees of freedom associated with the model. It is the number of predictors, 2 in our case — X and sin_X.

If your data has categorical features, statsmodel will one-hot encode them. But in that process, it will drop one of the one-hot encoded features.

This is done to avoid the dummy variable trap, which we discussed in an earlier chapter (this chapter).

- Covariance type: This is related to the assumptions about the distribution of the residual.
    - In linear regression, we assume that the residuals have a constant variance (homoscedasticity).
    - We use "nonrobust" to train a model under that assumption.

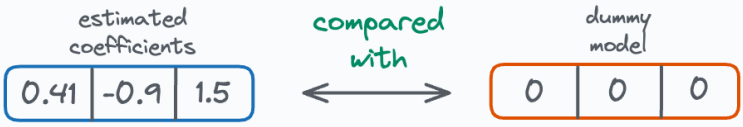
In the second column, statsmodel provides the overall performance-related details:



- R-squared: The fraction of original data variability captured by the model.

$$R^2 = \frac{\text{Variability captured by the Model}}{\text{Variability in the original dataset}}$$

  - For instance, in this case, 0.927 means that the current model captures 92.7% of the original variability in the training data.
  - Statsmodel reports R2 on the input data, so you must not overly optimize for it. If you do, it will lead to overfitting.
- Adj. R-squared:
  - It is somewhat similar to R-squared, but it also accounts for the number of predictors (features) in the model.
  - The problem is that R-squared always increases as we add more features.
  - So even if we add totally irrelevant features, R-squared will never decrease. Adj. R-squared penalizes this behavior of R-squared.
- F-statistic and Prob (F-statistic):
  - These assess the overall significance of a regression model.
  - They compare the estimated coefficients by OLS with a model whose all coefficients (except for the intercept) are zero.



  - F-statistic tests whether the independent variables collectively have any effect on the dependent variable or not.
  - Prob (F-statistic) is the associated p-value with the F-statistic.
  - A small p-value (typically less than 0.05) indicates that the model as a whole is statistically significant.
  - This means that at least one independent variable has a significant effect on the dependent variable.
- Log-Likelihood:

- This tells us the log-likelihood that the given data was generated by the estimated model.
- The higher the value, the more likely the data was generated by this model.

- AIC and BIC:
  - Like adjusted R-squared, these are performance metrics to determine goodness of fit while penalizing complexity.
  - Lower AIC and BIC values indicate a better fit.

Section #2

The second section provides details related to the features:

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	1.4848	0.068	21.804	0.000	1.350	1.620
X	0.6936	1.311	0.529	0.598	-1.909	3.296
sin_X	3.0976	1.515	2.045	0.044	0.092	6.104

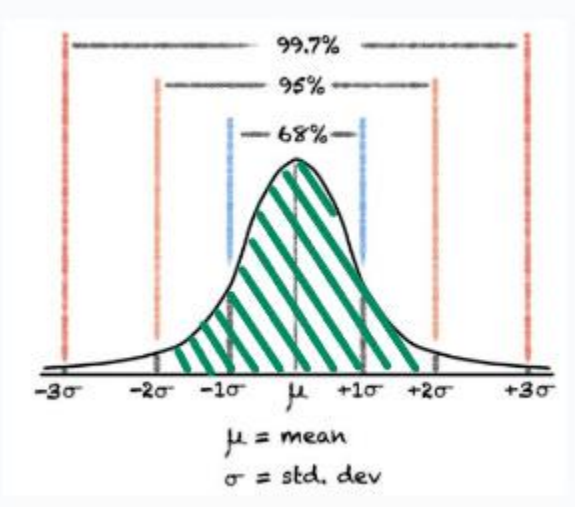
- coef: The estimated coefficient for a feature.

- t and P>|t|:
  - Earlier, we used F-statistic to determine the statistical significance of the model as a whole.
  - t-statistic is more granular on that front as it determines the significance of every individual feature.
  - P>|t| is the associated p-value with the t-statistic.
  - A small p-value (typically less than 0.05) indicates that the feature is statistically significant.

  - For instance, the feature "X" has a p-value of ~0.6. This suggests that there is a 60% chance that the feature "X" has no effect on "Y".

- [0.025, 0.975] and std err:

- See, the coefficients we have obtained from the model are just estimates. They may not be absolute true coefficients of the process that generated the data.
- Thus, the estimated parameters are subject to uncertainty, aren't they?
- Note: The width of the interval [0.025, 0.975] is 0.95 → or 95%. This constitutes the area between 2 standard deviations from the mean in a normal distribution.



- A 95% confidence interval provides a range of values within which you can be 95% confident that the true value of the parameter lies.

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	1.4848	0.068	21.804	0.000	1.350	1.620
X	0.6936	1.311	0.529	0.598	-1.909	3.296
sin_X	3.0976	1.515	2.045	0.044	0.092	6.104

- For instance, the interval for sin_X is (0.092, 6.104). So although the estimated coefficient is 3.09, we can be 95% confident that the true coefficient lies in the range (0.092, 6.104).

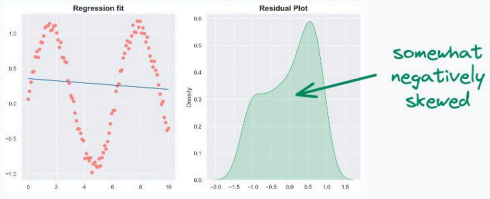
Section #3

Details in the last section of the report test the assumptions of linear regression.

```
==============================================================================
Omnibus:                       13.790   Durbin-Watson:                   2.062
Prob(Omnibus):                  0.001   Jarque-Bera (JB):                4.979
Skew:                          -0.244   Prob(JB):                       0.0830
Kurtosis:                       2.022   Cond. No.                        87.6
==============================================================================
```

- Omnibus and Prob(Omnibus):
  - They test the normality of the residuals.
  - Omnibus value of zero means residuals are perfectly normal.
  - Prob(Omnibus) is the corresponding p-value.
    - In this case, Prob(Omnibus) is 0.001. This means there is a 0.1% chance that the residuals are normally distributed.
- Skew and Kurtosis:
  - They also provide information about the distribution of the residuals.
  - Skewness measures the asymmetry of the distribution of residuals.
    - Zero skewness means perfect symmetry.
    - Positive skewness indicates a distribution with a long right tail. This indicates a concentration of residuals on lower values. Good to check for outliers in this case.

Negative skewness indicates a distribution with a long left tail. This is mostly indicative of poor features. For instance, consider fitting a sin curve with a linear feature (X). Most residuals will be high, resulting in negative skewness.



- Durbin-Watson:
    - This measures autocorrelation between residuals.
    - Autocorrelation occurs when the residuals are correlated, indicating that the error terms are not independent.
    - But linear regression assumes that residuals are not correlated.
    - The Durbin-Watson statistic ranges between 0 and 4.
        - A value close to 2 indicates no autocorrelation.
        - Values closer to 0 indicate positive autocorrelation.
        - Values closer to 4 indicate negative autocorrelation.
- Jarque-Bera (JB) and Prob(JB):
    - They solve the same purpose as Omnibus and Prob(Omnibus) — measuring the normality of residuals.
- Condition Number:
    - This tests multicollinearity.
    - Multicollinearity occurs when two features are correlated, or two or more features determine the value of another feature.
    - A standalone value for Condition Number can be difficult to interpret so here's how I use it:
        - Add features one by one to the regression model and notice any spikes in the Condition Number.

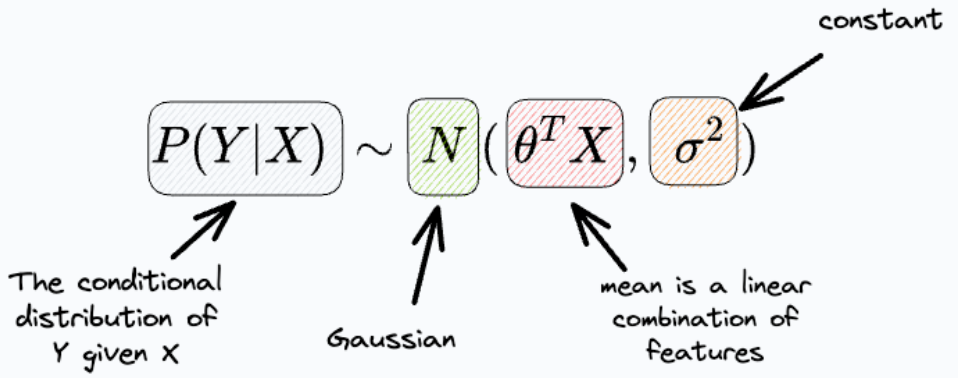
As discussed above, every section of this report has its importance:

- The first section tells us about the model's config, the overall performance of the model, and its statistical significance.
- The second section tells us about the statistical significance of individual features, the model's confidence in finding the true coefficient, etc.
- The last section lets us validate the model's assumptions, which are immensely critical to linear regression's performance.

Now you know how to interpret the entire regression summary from statsmodel.

Generalized Linear Models (GLMs)

A linear regression model is undeniably an extremely powerful model, in my opinion. However, it makes some strict assumptions about the type of data it can model, as depicted below.

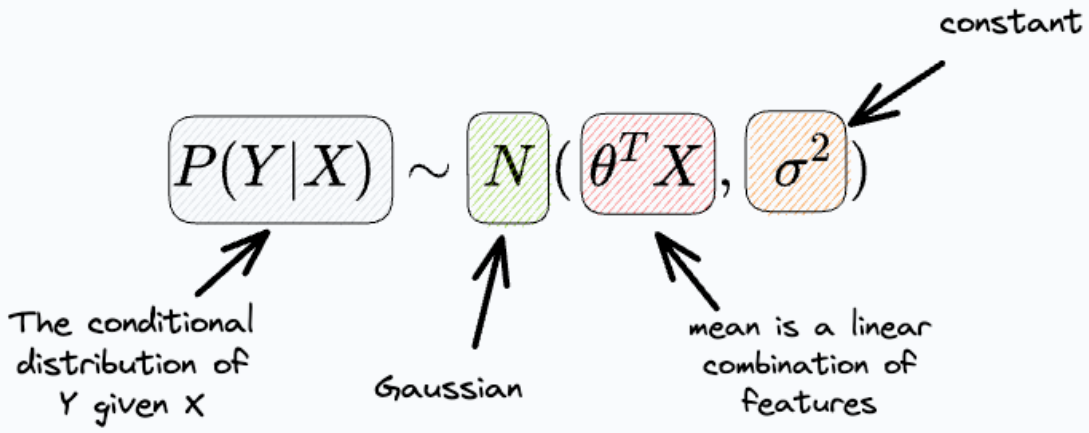


These conditions often restrict its applicability to data situations that do not obey the above assumptions. That is why being aware of its extensions is immensely important.

Generalized linear models (GLMs) precisely do that. They relax the assumptions of linear regression to make linear models more adaptable to real-world datasets.

Why GLMs?

Linear regression is pretty restricted in terms of the kind of data it can model. For instance, its assumed data generation process looks like this:



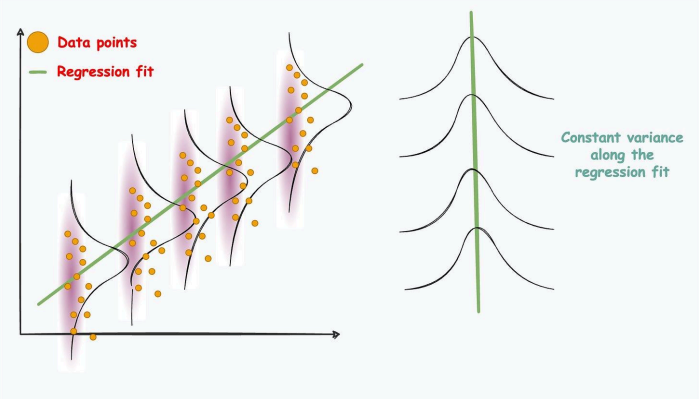
The assumed data generation process of linear regression

- Firstly, it assumes that the conditional distribution of Y given X is a Gaussian.
- Next, it assumes a very specific form for the mean of the above Gaussian. It says that the mean should always be a linear combination of the features (or predictors).

$$\mu(x) = \sum_{j=1}^{K} \theta_j \cdot x_j$$

mean is a linear combination of features

- Lastly, it assumes a constant variance for the conditional distribution P(Y|X) across all levels of X. A graphical way of illustrating this is as follows:



These conditions often restrict its applicability to data situations that do not obey the above assumptions.

In other words, nothing stops real-world datasets from violating these assumptions.

In fact, in many scenarios, the data might exhibit complex relationships, heteroscedasticity (varying variance), or even follow entirely different distributions altogether.

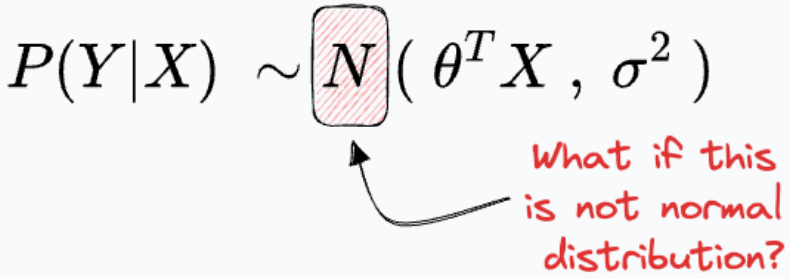
Yet, if we intend to build linear models, we should formulate better algorithms that can handle these peculiarities.

Generalized linear models (GLMs) precisely do that.

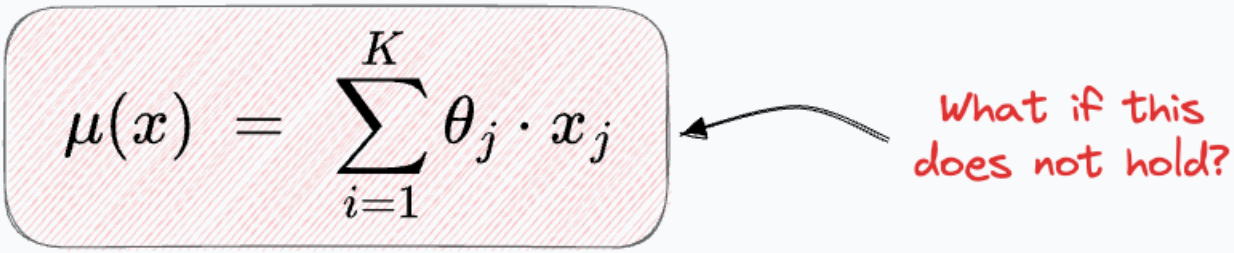
They relax the assumptions of linear regression to make linear models more adaptable to real-world datasets.

More specifically, they consider the following:

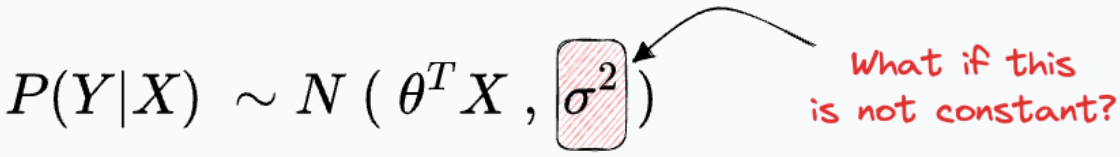
- What if the distribution isn't normal but some other distribution?



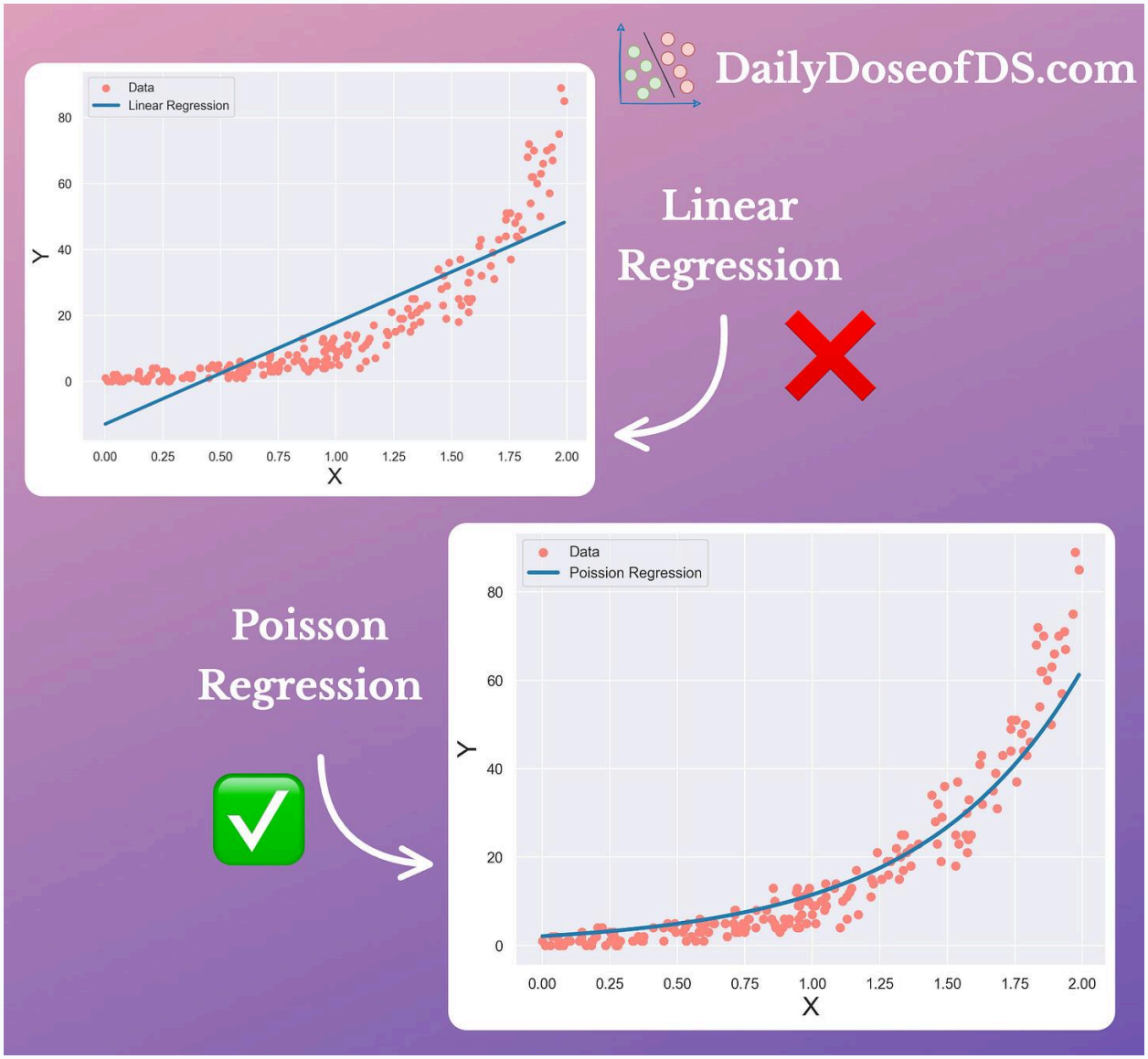
- What if X has a more sophisticated relationship with the mean?



- What if the variance varies with X?



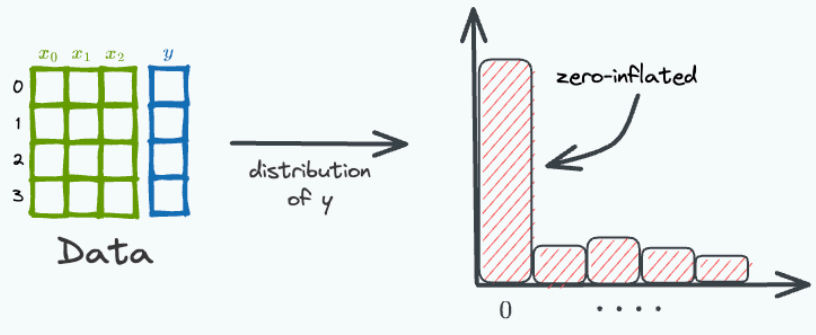
The effectiveness of a specific GLM — Poisson regression (which we discussed in an earlier chapter) over linear regression is evident from the image below:



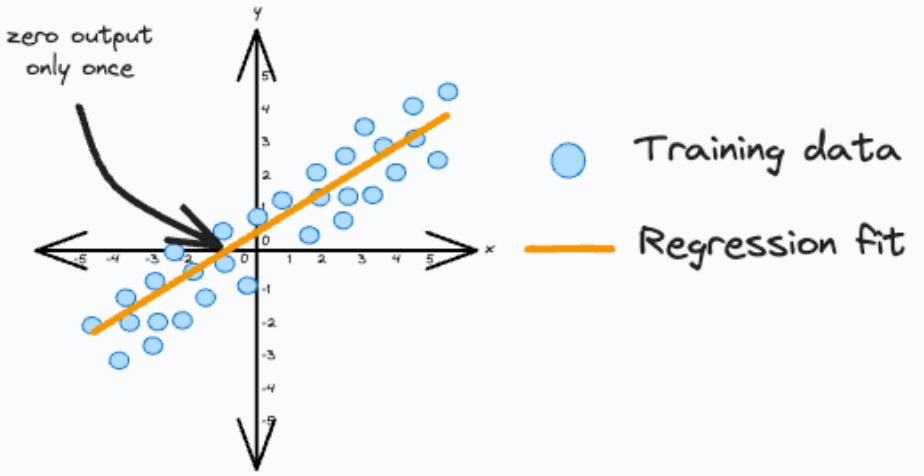
- Linear regression assumes the data is drawn from a Gaussian, when in reality, it isn't. Hence, it underperforms.
- Poisson regression adapts its regression fit to a non-Gaussian distribution. Hence, it performs significantly better.

Zero-inflated Regression

The target variable of typical regression datasets is somewhat evenly distributed. But, at times, the target variable may have plenty of zeros. Such datasets are called zero-inflated datasets.



They may raise many problems during regression modeling. This is because a regression model can not always predict exact "zero" values when, ideally, it should. For instance, consider simple linear regression. The regression line will output exactly "zero" only once (if it has a non-zero slope).

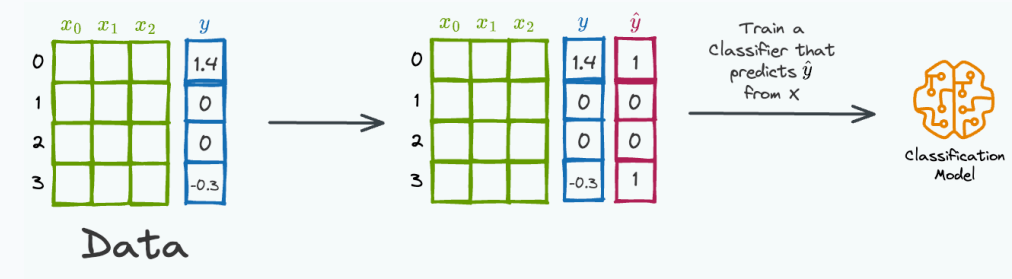


This issue persists not only in higher dimensions but also in complex models like neural nets for regression.

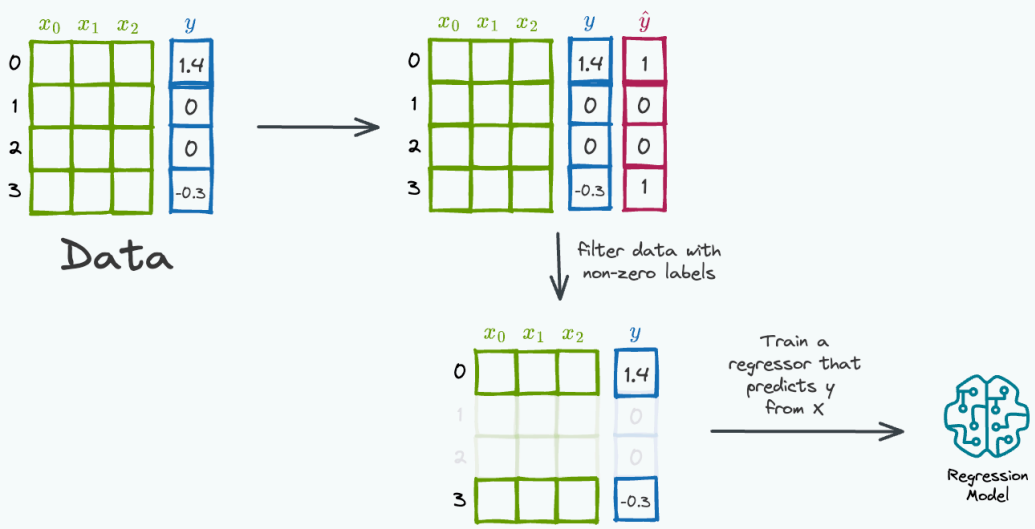
One great way to solve this is by training a combination of a classification and a regression model.

This goes as follows:

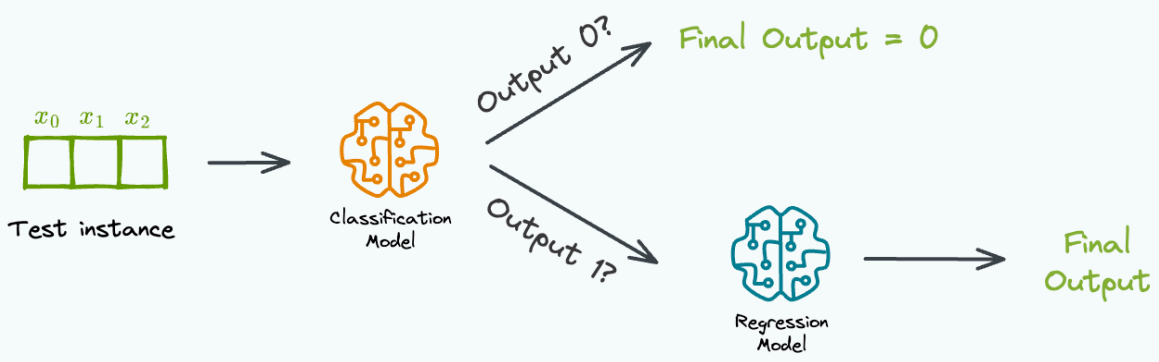
- Mark all non-zero targets as “1” and the rest as “0”.
- Train a binary classifier on this dataset.



- Next, train a regression model only on those data points with a non-zero true target.

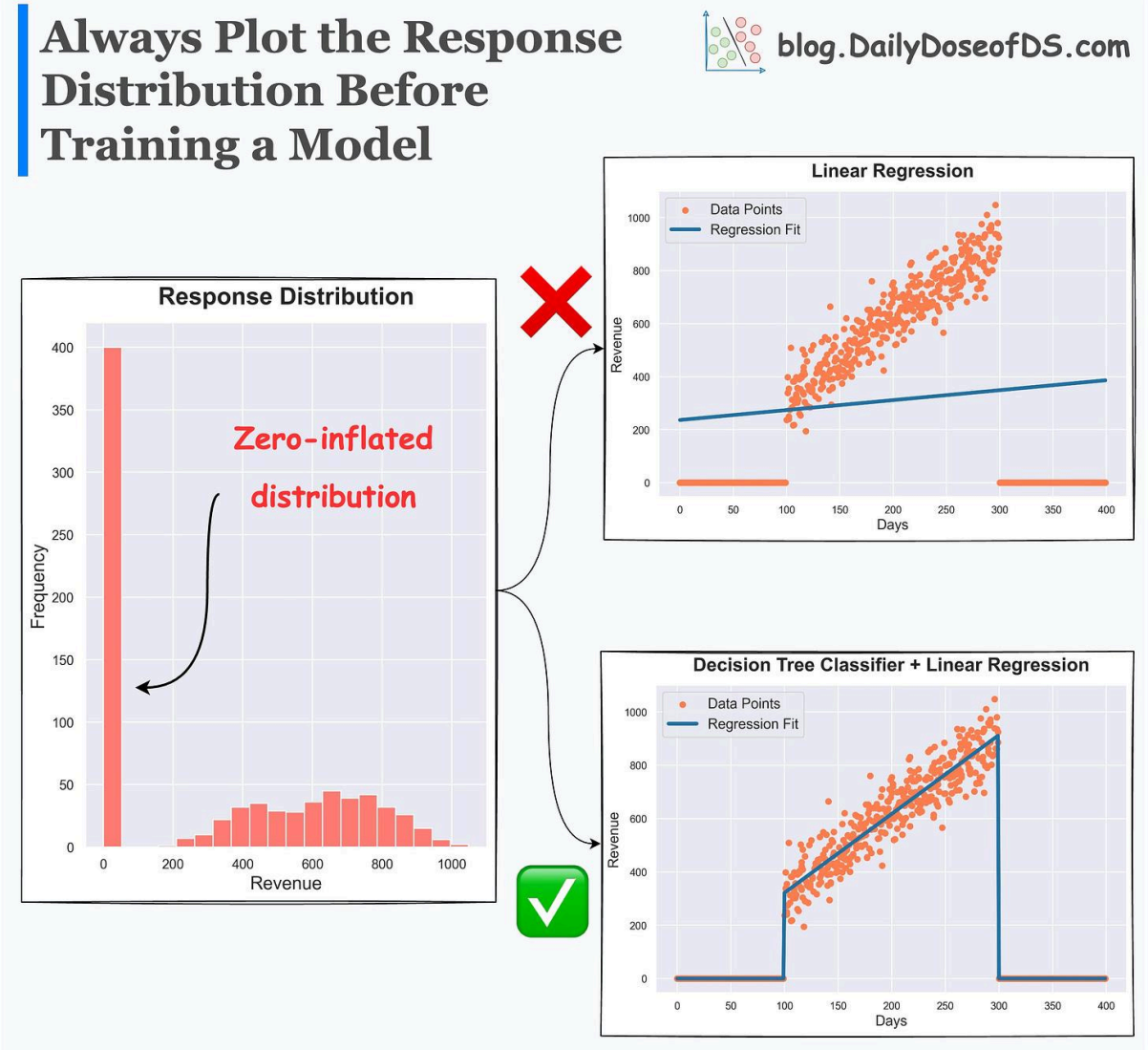


During prediction:



- If the classifier's output is “0”, the final output is also zero.
- If the classifier's output is “1”, use the regression model to predict the final output.

Its effectiveness over the regular regression model is evident from the image below:



Regression vs. Regression + Classification results

- Linear regression alone underfits the data.
- Linear regression with a classifier performs as expected.

Huber Regression

One big problem with regression models is that they are sensitive to outliers. Consider linear regression. Even a few outliers can significantly impact Linear Regression performance, as shown below:

And it isn't hard to identify the cause of this problem. Essentially, the loss function (MSE) scales quickly with the residual term (true-predicted).

Thus, even a few data points with a large residual can impact parameter estimation.

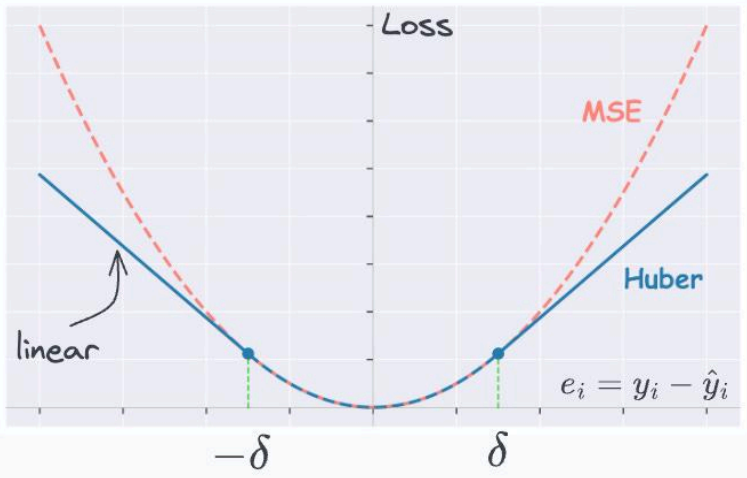
Huber loss (or Huber Regression) precisely addresses this problem. In a gist, it attempts to reduce the error contribution of data points with large residuals.

One simple, intuitive, and obvious way to do this is by applying a threshold (δ) on the residual term:

- If the residual is smaller than the threshold, use MSE (no change here).
- Otherwise, use a loss function which has a smaller output than MSE — linear, for instance.

This is depicted below:

- For residuals smaller than the threshold (δ) → we use MSE.
- Otherwise, we use a linear loss function which has a smaller output than MSE.



Mathematically, Huber loss is defined as follows:

$$e_i = y_i - \hat{y}_i$$

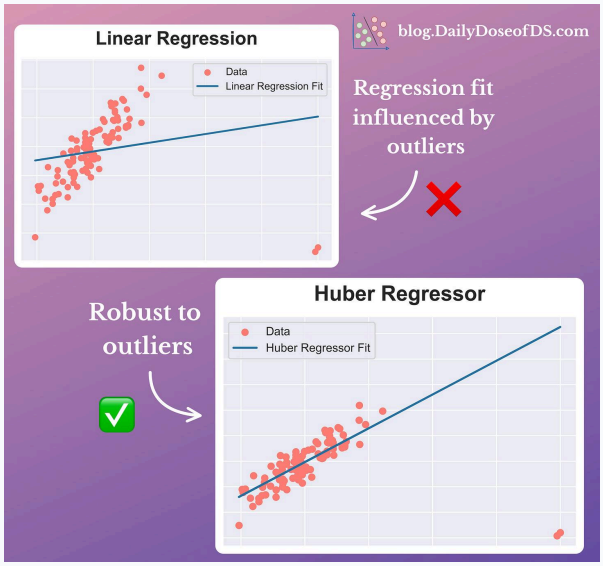
$$L_{Huber}(e_i) = \begin{cases} \frac{1}{2}e_i^2 & : e_i \leq \delta \\ \delta \cdot (|e_i| - \frac{1}{2}\delta) & : \text{otherwise} \end{cases}$$

MSE

Linear

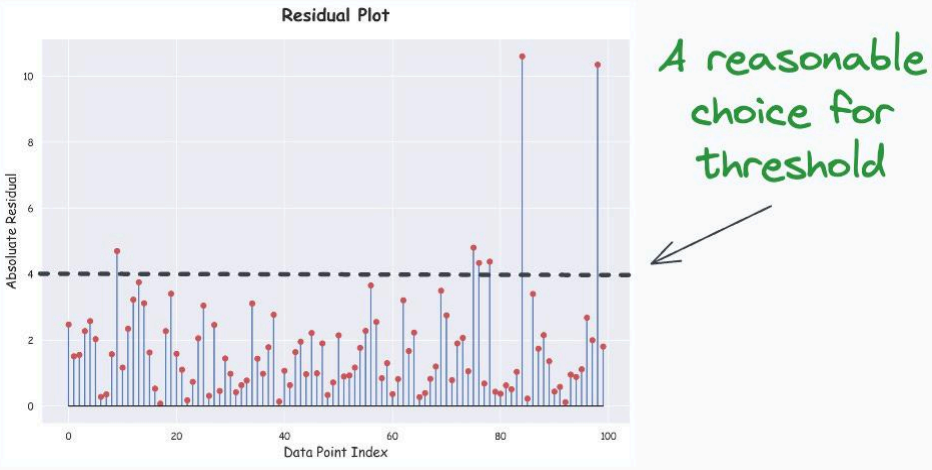
Its effectiveness is evident from the following image:

- Linear Regression is affected by outliers
- Huber Regression is more robust.



How do we determine the threshold (δ)?

While trial and error is one way, I often like to create a residual plot. This is depicted below: The below plot is generally called a lollipop plot because of its appearance.



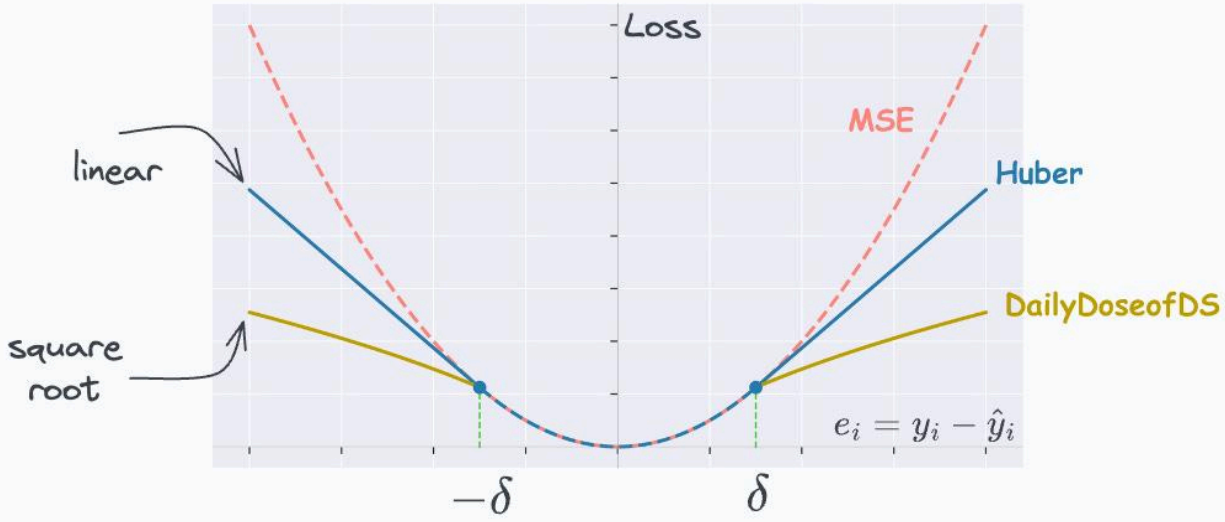
- Train a linear regression model as you usually would.
- Compute the residuals (=true-predicted) on the training data.
- Plot the absolute residuals for every data point.

One good thing is that we can create this plot for any dimensional dataset. The objective is just to plot (true-predicted) values, which will always be 1D.

Next, you can subjectively decide a reasonable threshold value δ.

Here's another interesting idea.

By using a linear loss function in Huber regressor, we intended to reduce the large error contributions that would have happened otherwise by using MSE. Thus, we can further reduce that error contribution by using, say, a square root loss function, as shown below:



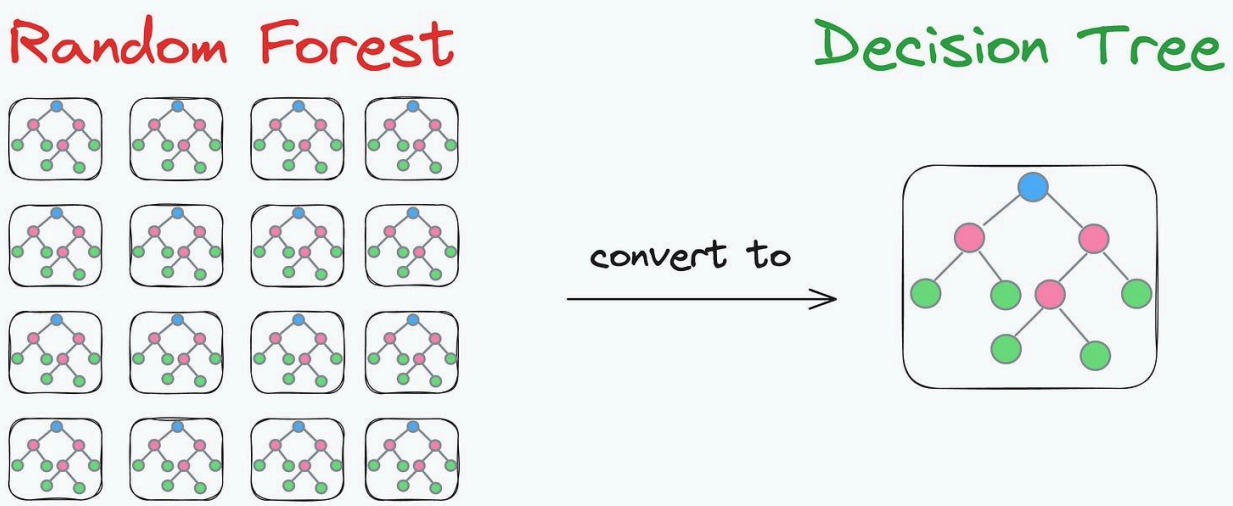
I am unsure if this has been proposed before, so I decided to call it the DailyDoseofDataScience Regressor 😉.

It is clear that the error contribution of the square root loss function is the lowest for all residuals above the threshold δ.

Decision Trees and Ensemble Methods

Condense Random Forest into a Decision Tree

There's an interesting technique, using which, we can condense an entire random forest model into a single decision tree.



The benefits?

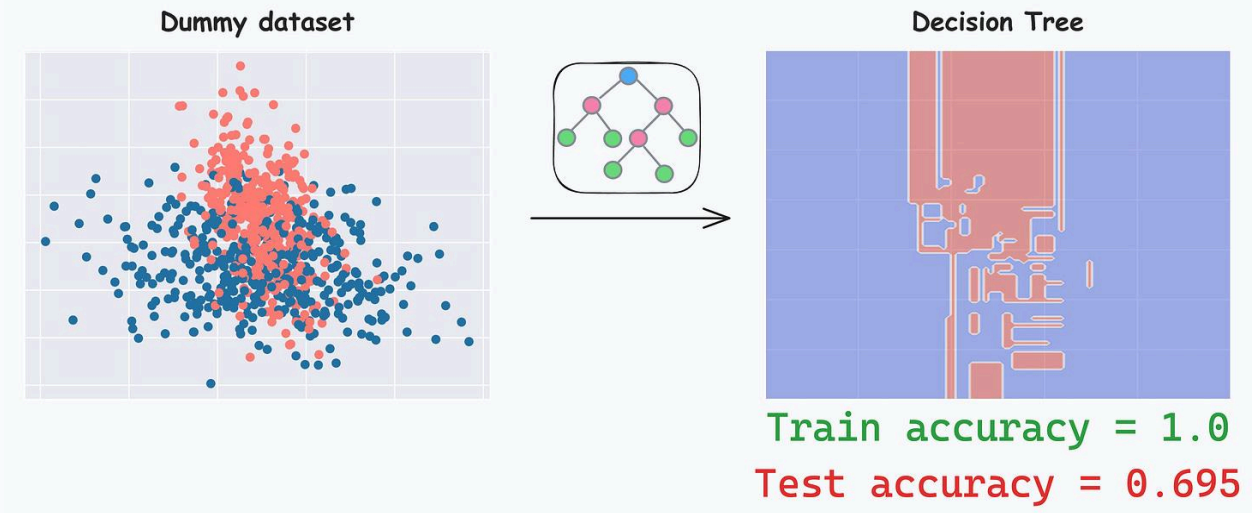
This technique can:

- Decrease the prediction run-time.
- Improve interpretability.
- Reduce the memory footprint.
- Simplify the model.
- Preserve the generalization power of the random forest model.

Let's understand in this chapter.

Technique Walkthrough

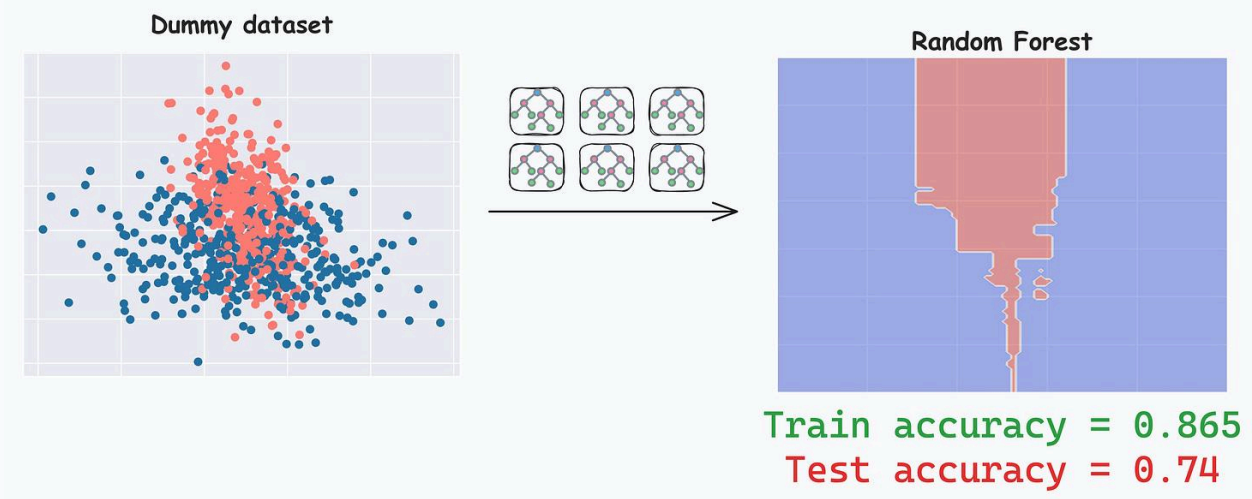
Let's fit a decision tree model on the following dummy dataset. It produces a decision region plot shown on the right.



It's clear that there is high overfitting.

In fact, we must note that, by default, a decision tree can always 100% overfit any dataset (we will use this information shortly). This is because it is always allowed to grow until all samples have been classified correctly.

This overfitting problem is resolved by a random forest model, as depicted below:

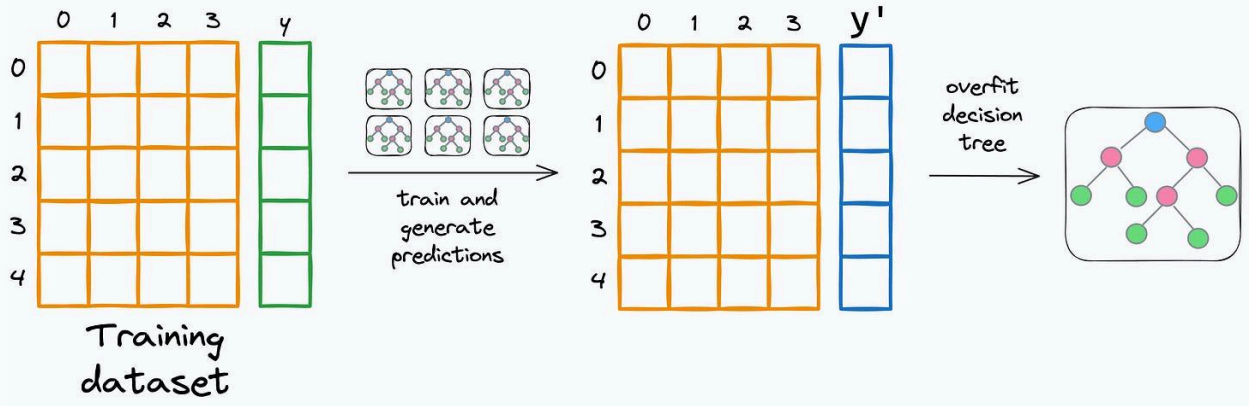


This time, the decision region plot suggests that we don't have a complex decision boundary. The test accuracy has also improved (69.5% to 74%).

Now, here's an interesting thing we can do.

We know that the random forest model has learned some rules that generalize on unseen data.

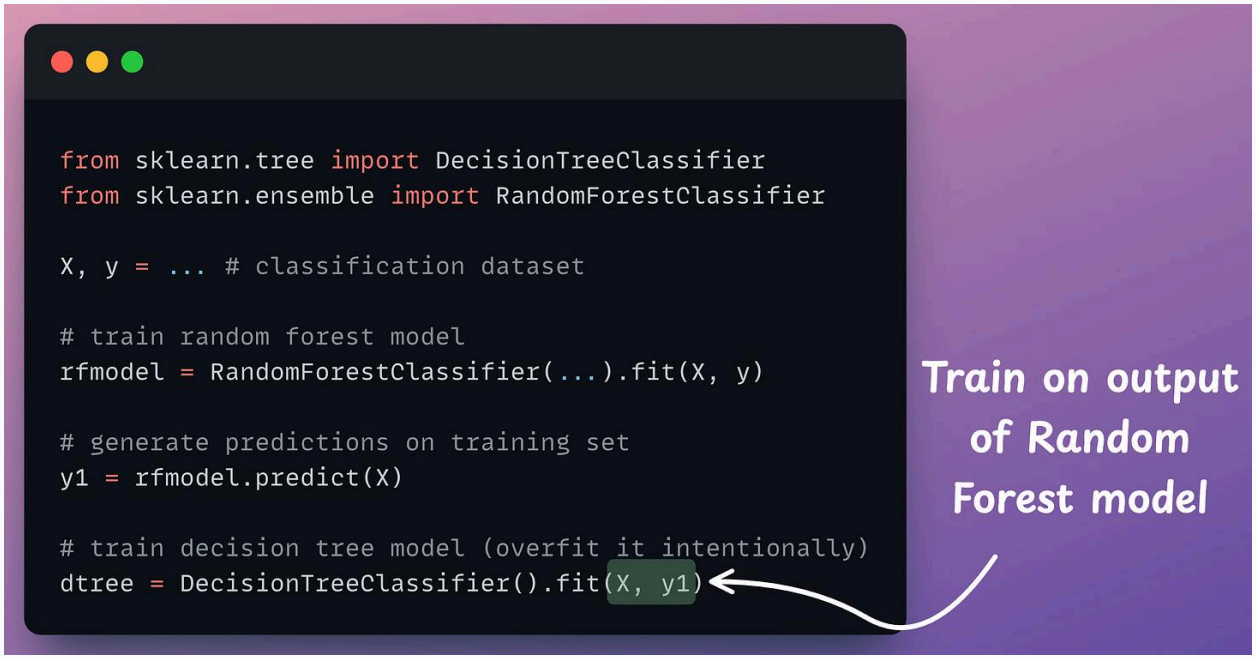
So, how about we train a decision tree on the predictions generated by the random forest model on the training set?



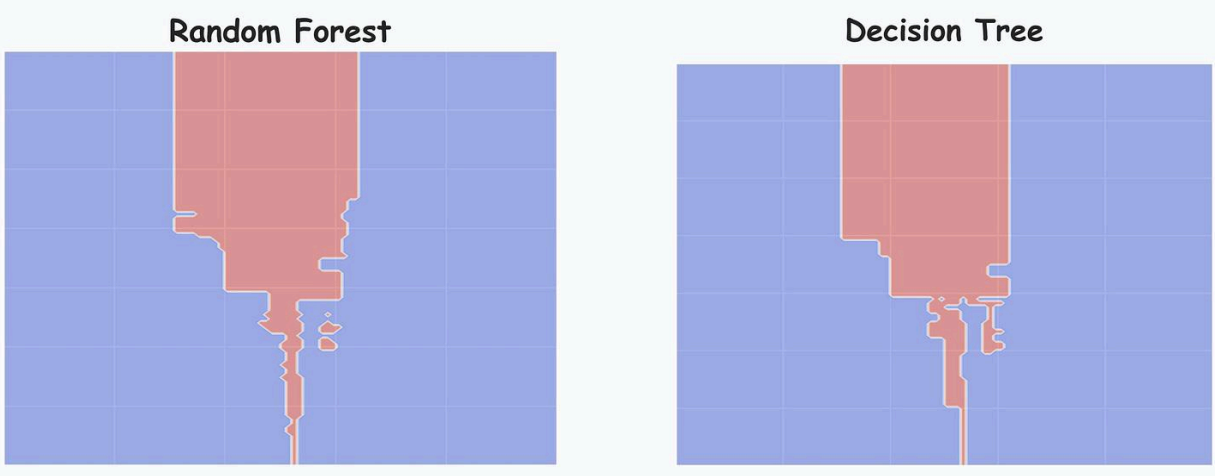
More specifically, given a dataset (X, y):

- Train a random forest model. This will learn some rules from the training set which are expected to generalize on unseen data (due to Bagging).
- Generate predictions on X, which produces the output y'. These predictions will capture the essence of the rules learned by the random forest model.
- Finally, train a decision tree model on (X, y'). Here, we want to intentionally overfit this mapping as this mapping from (X) to (y') is a proxy for the rules learned by the random forest model.

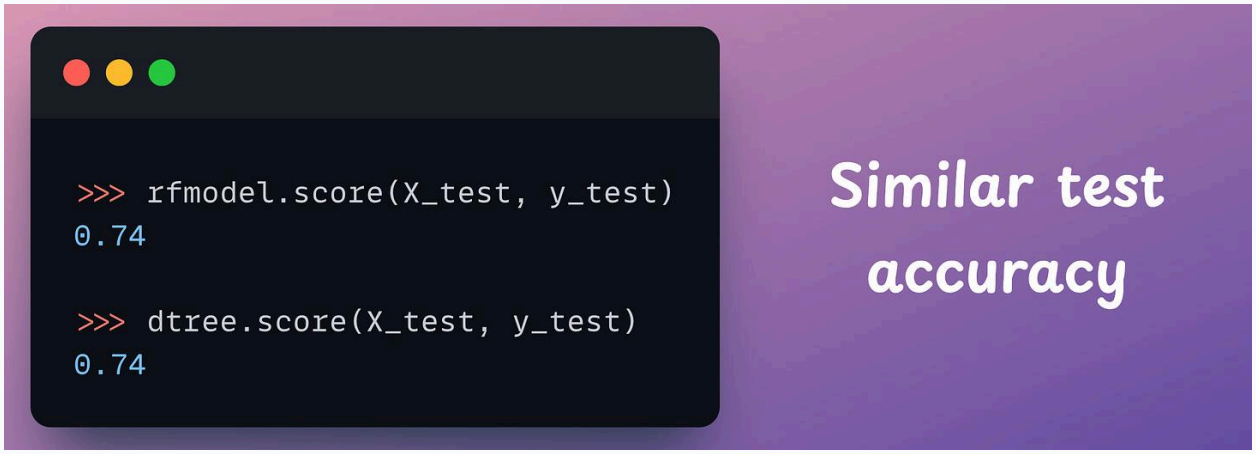
This idea is implemented below:



The decision region plot we get with the new decision tree is pretty similar to what we saw with the random forest earlier:



Measuring the test accuracy of the decision tree and random forest model, we notice them to be similar too:



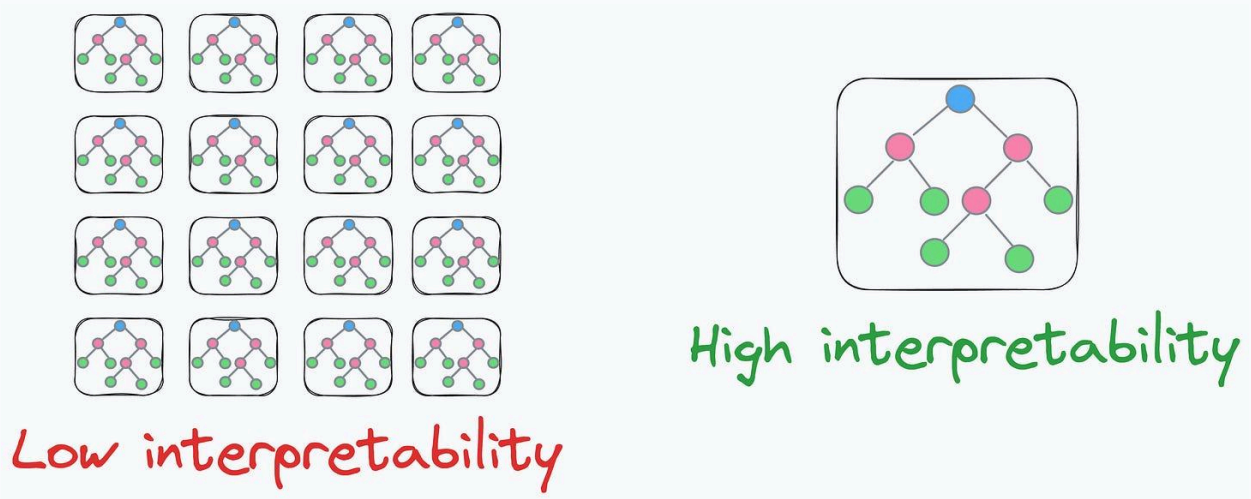
In fact, this approach also significantly reduces the run-time, as depicted below:

Isn't that cool?

Another rationale for considering doing this is that it adds interpretability.

This is because if we have 100 trees in a random forest, there's no way we can interpret them.

However, if we have condensed it to a decision tree, now we can inspect it.



A departing note

I devised this very recently. I also tested this approach on a couple of datasets, and they produced promising results.

But it won't be fair to make any conclusions based on just two instances.

While the idea makes intuitive sense, I understand there could be some potential flaws that are not evident right now.

So, I not saying that you should adopt this technique right away.

Instead, it is advised to test this approach on your random forest use cases. Considering reverting back to me with what you discovered.

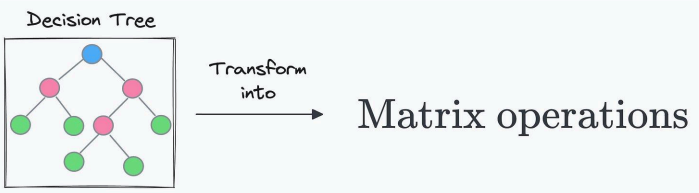
The code for this chapter is available here: https://bit.ly/3XSPejD.

In this next chapter, let's understand a technique to transform a decision tree into matrix operations which can run on GPUs.

Transform Decision Tree into Matrix Operations

Inference using a decision tree is an iterative process. We traverse a decision tree by evaluating the condition at a specific node in a layer until we reach a leaf node.

In this chapter, let's learn a superb technique that to represent inferences from a decision tree in the form of matrix operations.

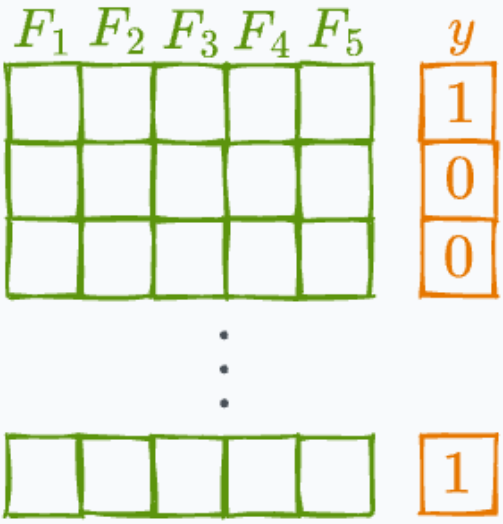


As a result:

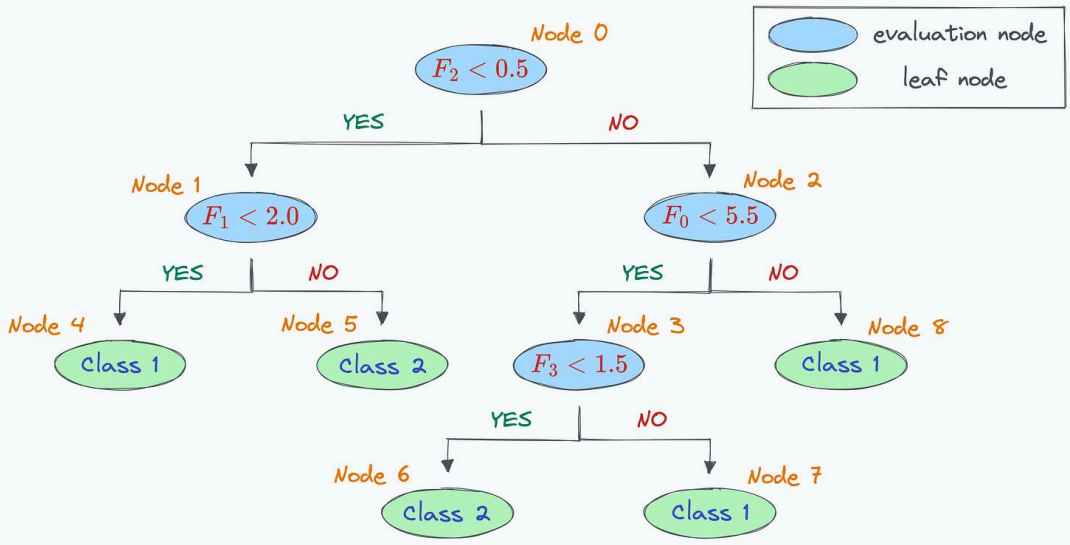
1. It makes inference much faster as matrix operations can be radically parallelized.
2. These operations can be loaded on a GPU for even faster inference, making them more deployment-friendly.

Setup

Consider a binary classification dataset with 5 features.



Let's say we get the following tree structure after fitting a decision tree on the above dataset:



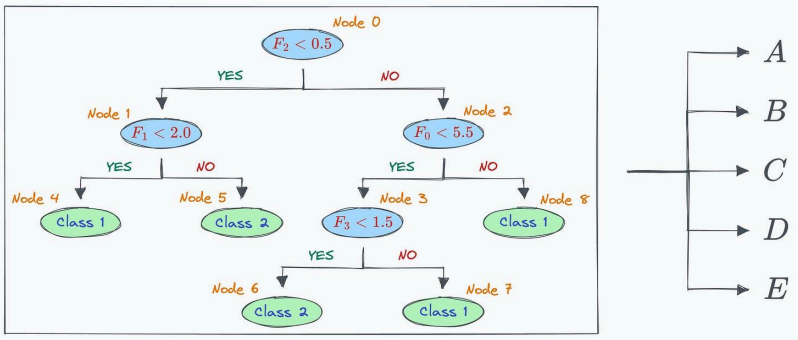
Notations

Before proceeding ahead, let's assume that:

- m → Total features in the dataset (5 in the dataset above).
- e → Total evaluation nodes in the tree (4 blue nodes in the tree above).
- l → Total leaf nodes in the tree (5 green nodes in the tree).
- c → Total classes in the dataset (2 in the dataset above).

Tree to Matrices

The core idea in this conversion is to derive five matrices (A, B, C, D, E) that capture the structure of the decision tree.



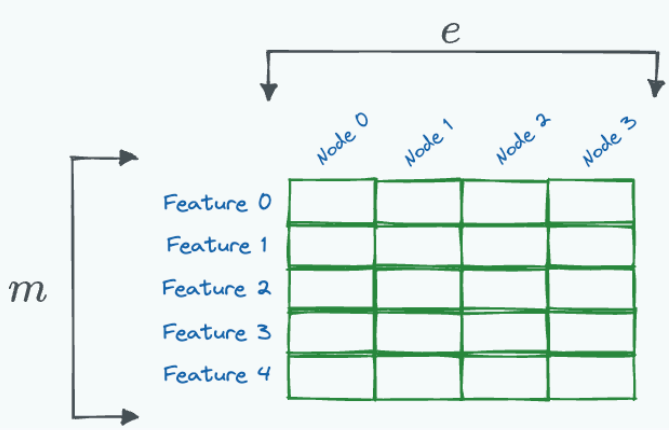
Let's understand them one by one!

Note: The steps we shall be performing below might not make immediate sense so please keep reading. You'll understand everything once we have derived all 5 matrices.

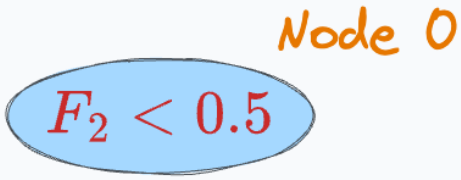
#1) Matrix A

This matrix captures the relationship between input features and evaluation nodes (blue nodes above).

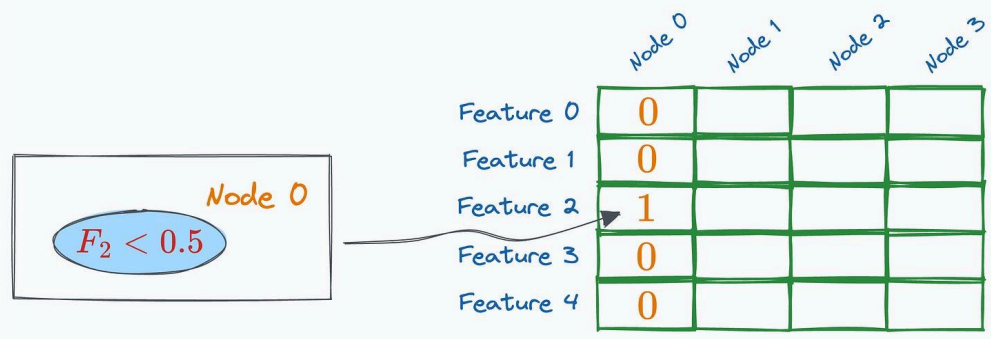
So it's an (m×e) shaped matrix.



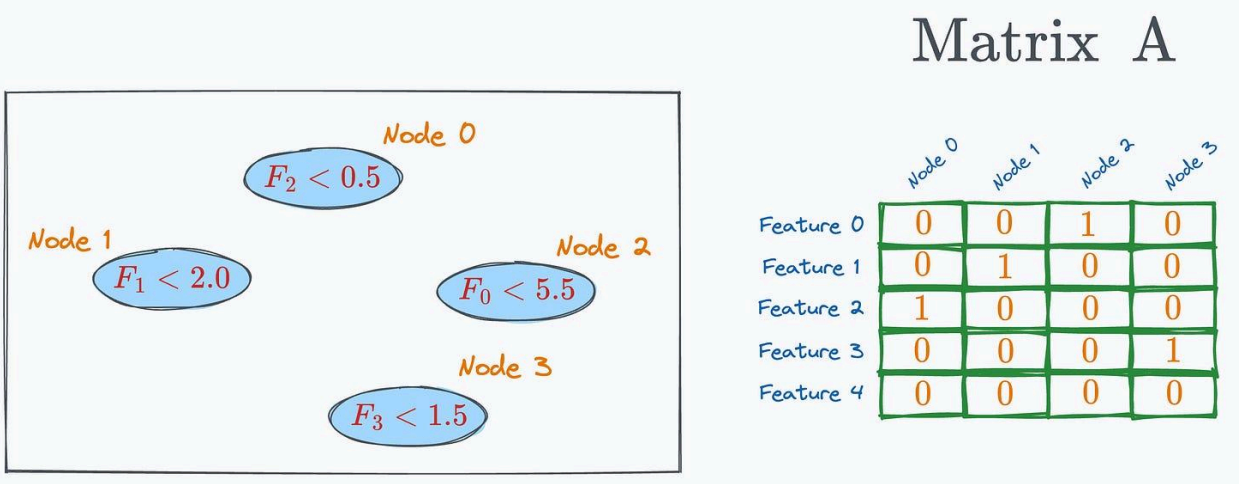
A specific entry is set to "1" if the corresponding node in the column evaluates the corresponding feature in the row. For instance, in our decision tree, "Node 0" evaluates "Feature 2".



Thus, the corresponding entry will be "1" and all other entries will be "0."

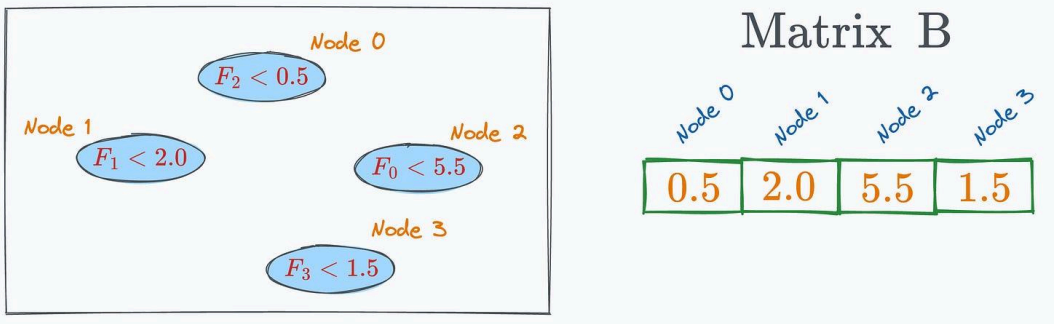


Filling out the entire matrix this way, we get:



#2) Matrix B

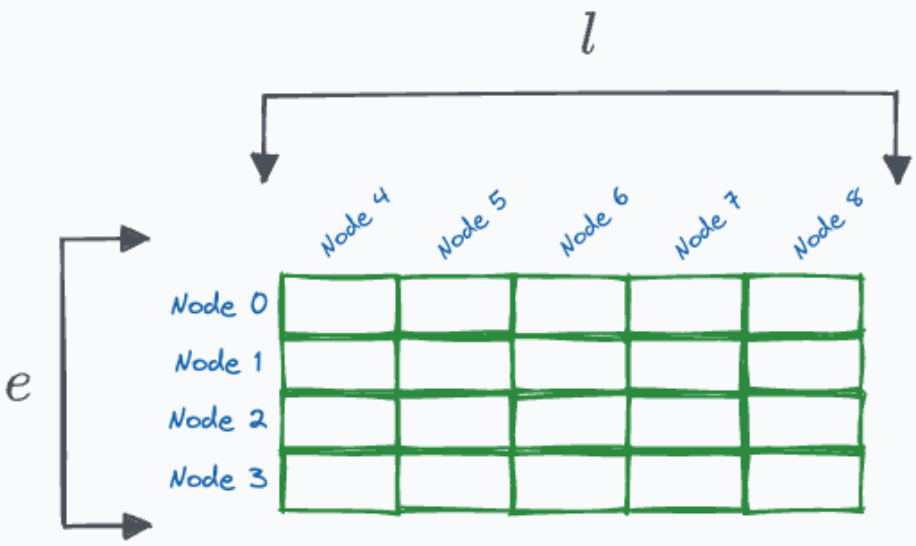
The entries of matrix B are the threshold value at each node. Thus, its shape is 1×e.



This is vector though, but the terminology is not important here.

#3) Matrix C

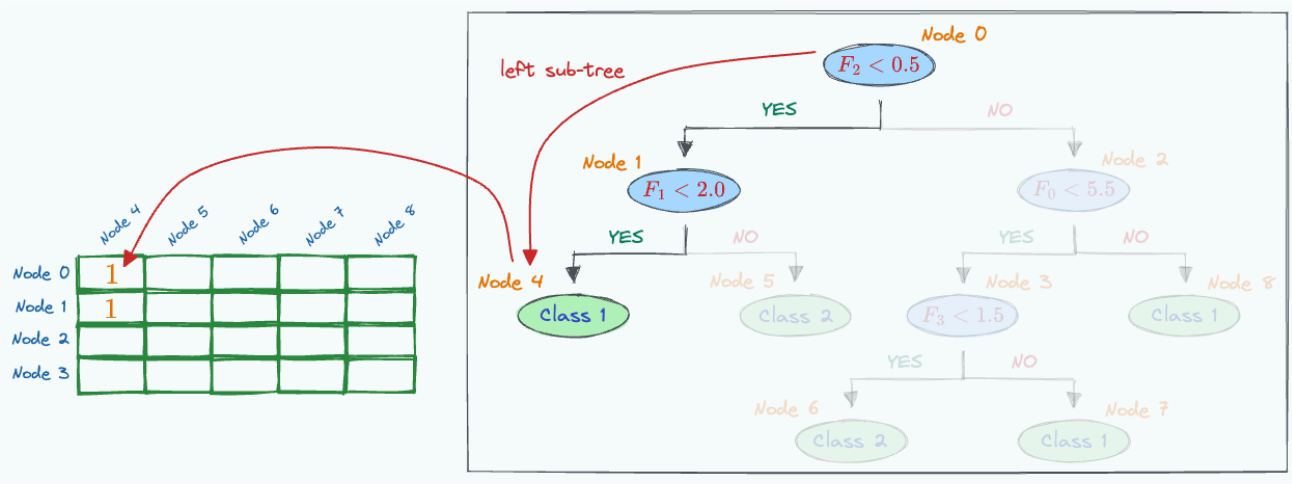
This is a matrix between every pair of leaf nodes and evaluation nodes. Thus, its dimensions are e×l.



A specific entry is set to:

- "1" if the corresponding leaf node in the column lies in the left sub-tree of the corresponding evaluation node in the row.
- "-1" if the corresponding leaf node in the column lies in the right sub-tree of the corresponding evaluation node in the row.
- "0" if the corresponding leaf node and evaluation node have no link.

For instance, in our decision tree, the "leaf node 4" lies on the left sub-tree of both "evaluation node 0" and "evaluation node 1". Thus, the corresponding values will be 1.



Filling out the entire matrix this way, we get:

Matrix C

	Node 4	Node 5	Node 6	Node 7	Node 8
Node 0	1	1	−1	−1	−1
Node 1	1	−1	0	0	0
Node 2	0	0	1	1	−1
Node 3	0	0	1	−1	0

#4) Matrix (or Vector) D

The entries of vector D are the sum of non-negative entries in every column of Matrix C:

Matrix C

	Node 4	Node 5	Node 6	Node 7	Node 8
Node 0	1	1	−1	−1	−1
Node 1	1	−1	0	0	0
Node 2	0	0	1	1	−1
Node 3	0	0	1	−1	0

Matrix D

Node 4	Node 5	Node 6	Node 7	Node 8
2	1	2	1	0

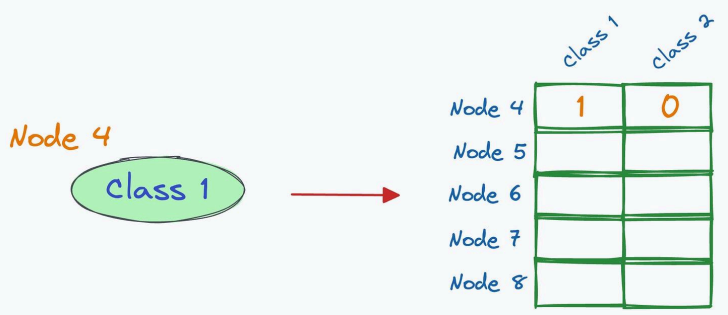
#5) Matrix E

Finally, this matrix holds the mapping between leaf nodes and their corresponding output labels. Thus, its dimensions are $l \times c$.

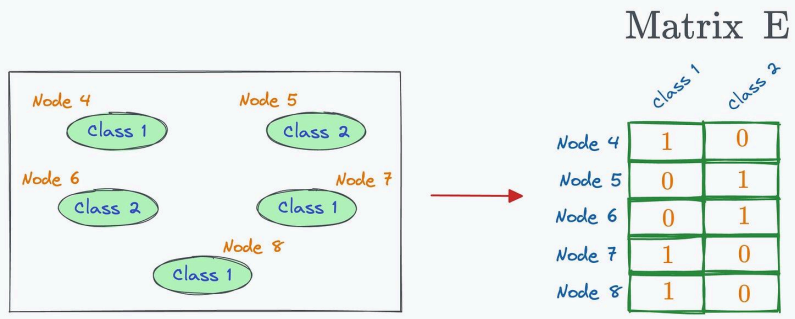
	Class 1	Class 2
Node 4		
Node 5		
Node 6		
Node 7		
Node 8		

If a leaf node classifies a sample to "Class 1", the corresponding entry will be 1, and the other cell entry will be 0.

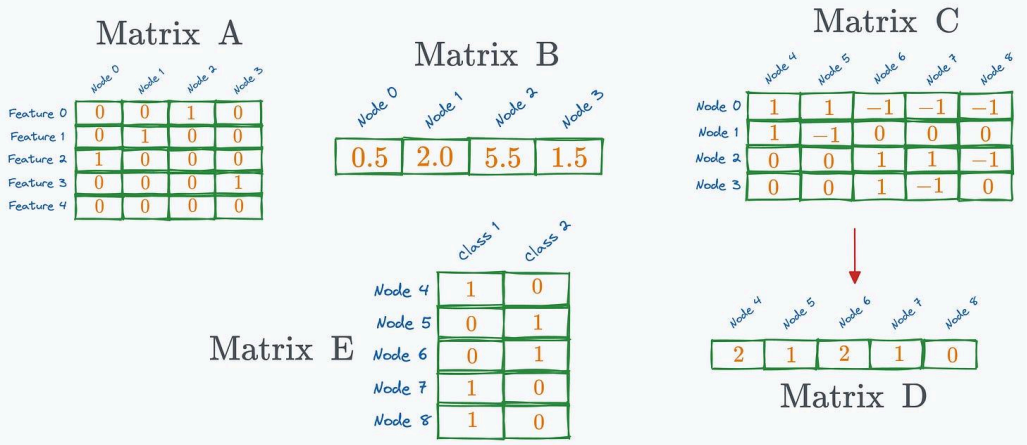
For instance, "lead node 4" outputs "Class 1", thus the corresponding entries for the first row will be (1,0):



We repeat this for all other leaf nodes to get the following matrix as Matrix E:



With this, we have compiled our decision tree into matrices. To recall, these are the five matrices we have created so far:



- Matrix A captures which input feature was used at each evaluation node.
- Matrix B stores the threshold of each evaluation node.
- Matrix C captures whether a leaf node lies on the left or right sub-tree of a specific evaluation node or has no relation to it.
- Matrix D stores the sum of non-negative entries in every column of Matrix C.
- Finally, Matrix E maps from leaf nodes to their class labels.

Inference using matrices

Say this is our input feature vector X (5 dimensions):

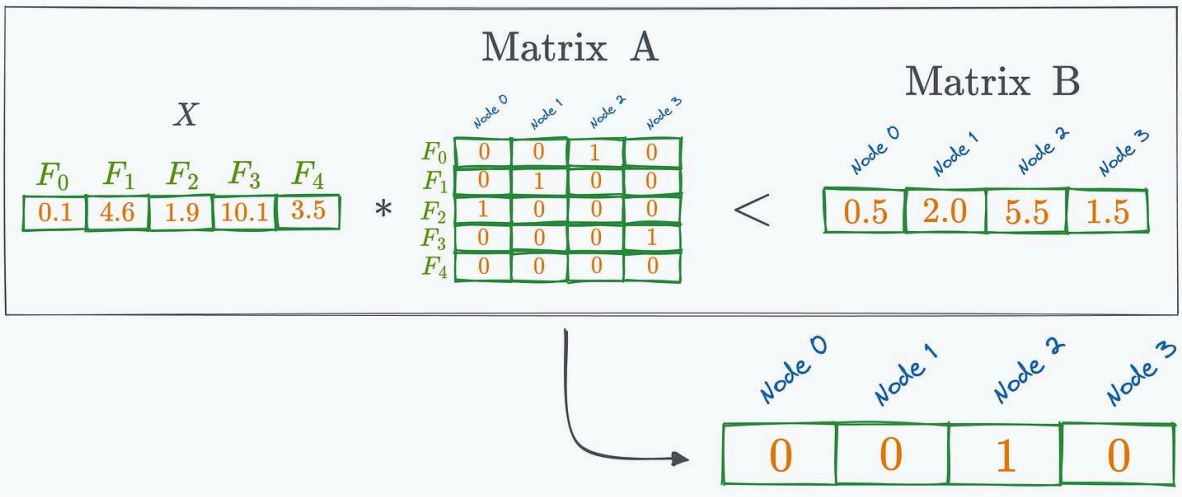
X

F_0	F_1	F_2	F_3	F_4
0.1	4.6	1.9	10.1	3.5

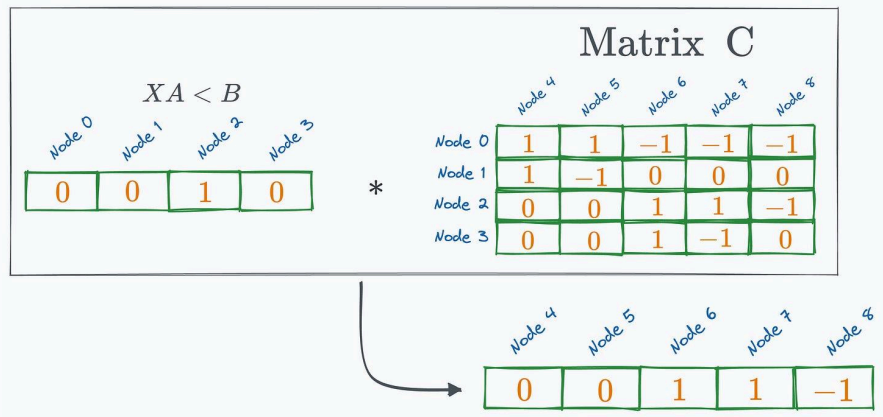
The whole inference can now be done using just these matrix operations:

$$(((XA < B) * C) == D) * E$$

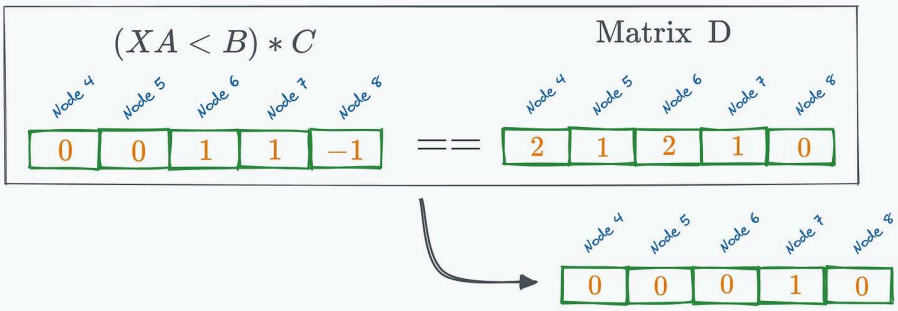
- XA < B gives:



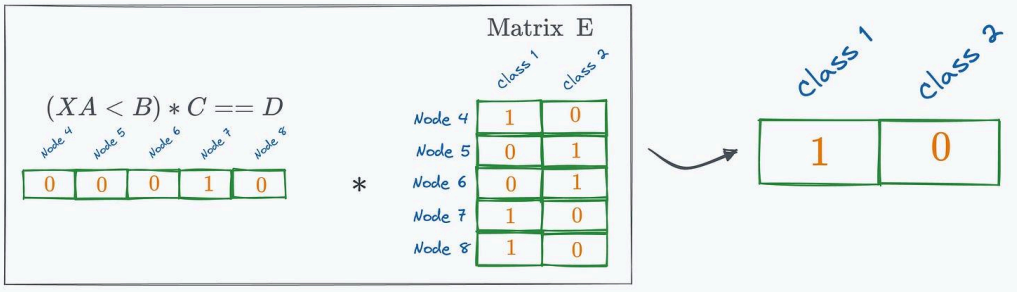
- The above result multiplied by C gives:



- The above result, when matched to D, gives:



- Finally, multiplying with E, we get:

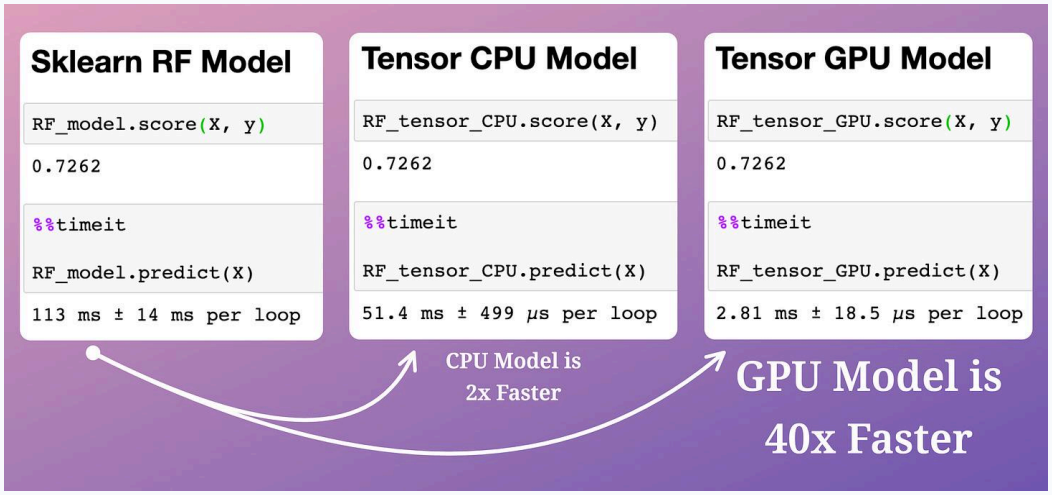


The final prediction comes out to be "Class 1," which is indeed correct! Notice that we carried out the entire inference process using only matrix operations:

$$(((XA < B) * C) == D) * E$$

As a result, the inference operation can largely benefit from parallelization and GPU capabilities.

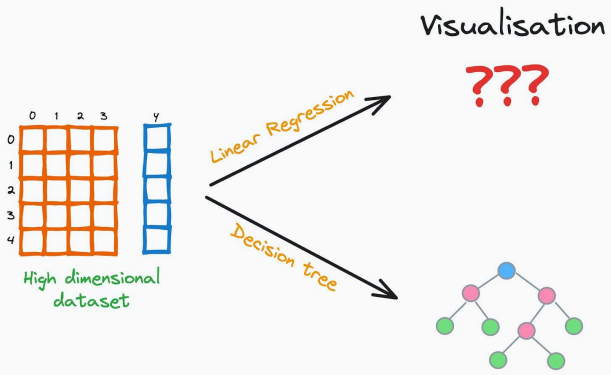
The run-time efficacy of this technique is evident from the image below:



- Here, we have trained a random forest model.
- The compiled model runs:
    - Over twice as fast on a CPU (Tensor CPU Model).
    - ~40 times faster on a GPU, which is huge (Tensor GPU Model).
- All models have the same accuracy — indicating no loss of information during compilation.

Interactively Prune a Decision Tree

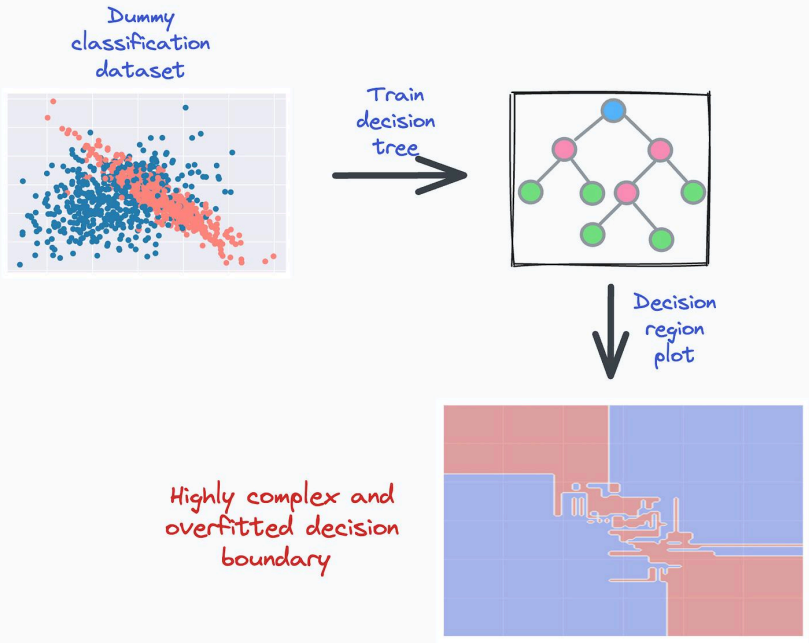
One thing I always appreciate about decision trees is their ease of visual interpretability. No matter how many features our dataset has, we can ALWAYS visualize and interpret a decision tree.



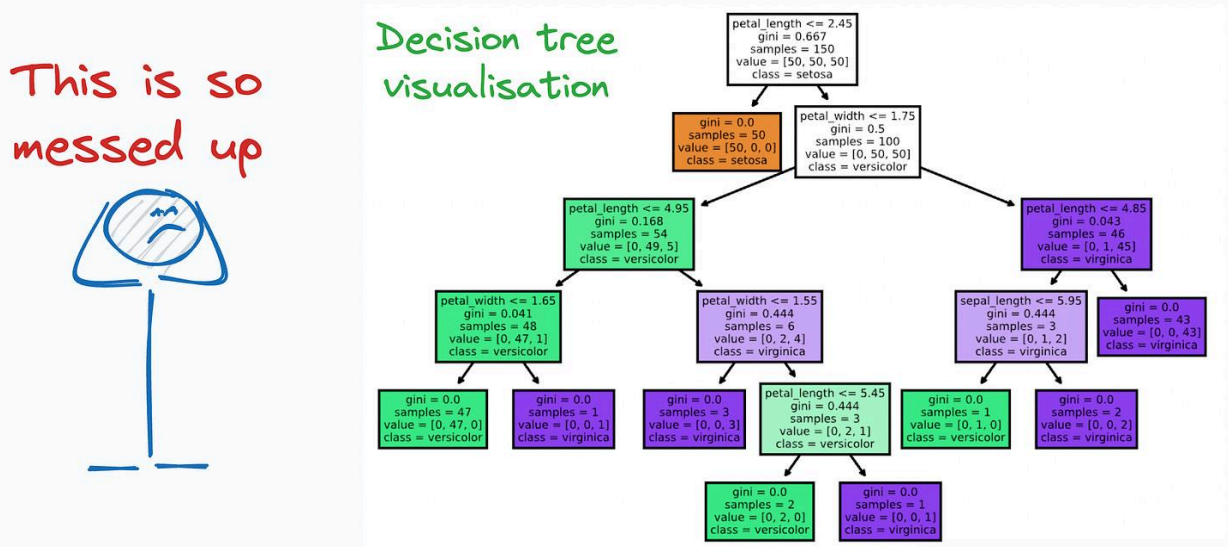
This is not always possible with other intuitive and simple models like linear regression. But decision trees stand out in this respect. Nonetheless, one thing I often find a bit time-consuming and somewhat hit-and-trial-driven is pruning a decision tree.

Why prune?

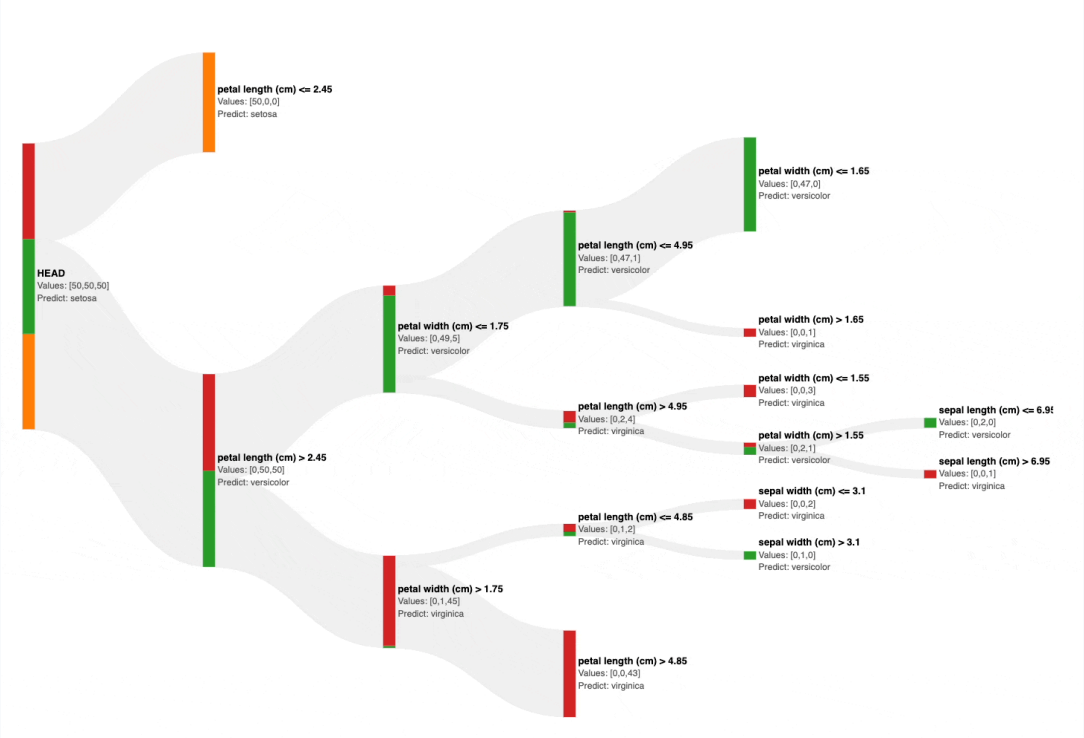
The problem is that under default conditions, decision trees ALWAYS 100% overfit the dataset, as depicted in this image:



Thus, pruning is ALWAYS necessary to reduce model variance. Scikit-learn already provides a method to visualize them as shown below:

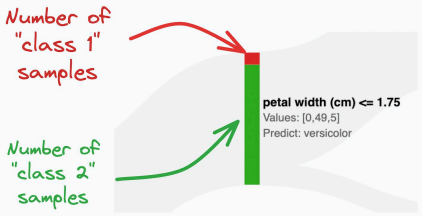


But the above visualisation is pretty non-elegant, tedious, messy, and static (or non-interactive). I recommend using an interactive Sankey diagram to prune decision trees. This is depicted below:

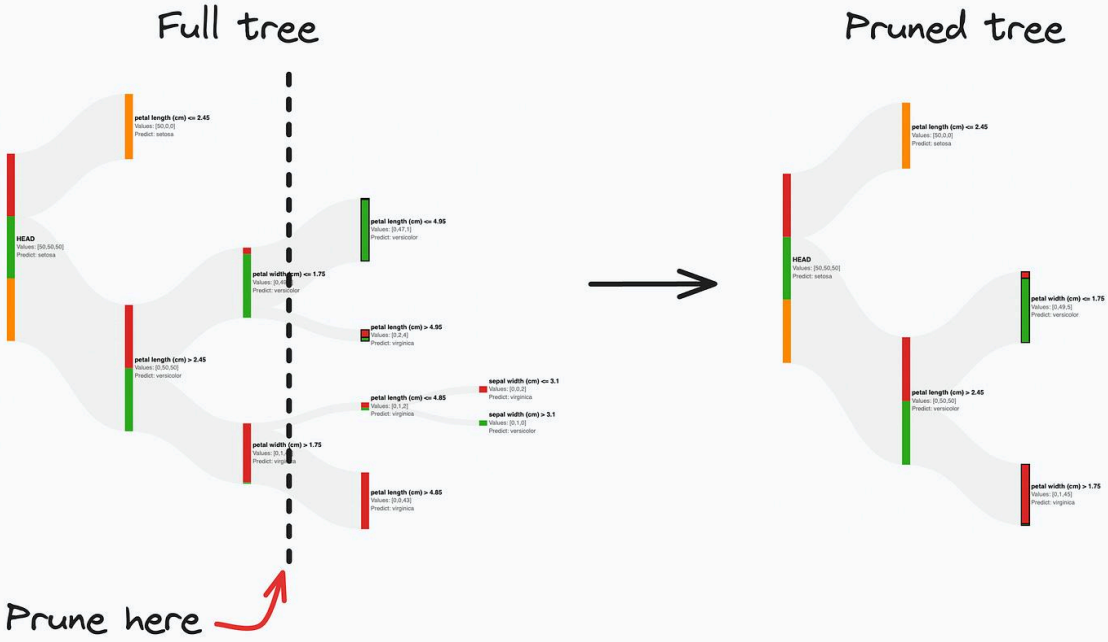


As shown above, the Sankey diagram allows us to interactively visualize and prune a decision tree by collapsing its nodes.

Also, the number of data points from each class is size and color-encoded in each node, as shown below.



This instantly gives an estimate of the node's impurity, based on which, we can visually and interactively prune the tree in seconds. For instance, in the full decision tree shown below, pruning the tree at a depth of two appears reasonable:



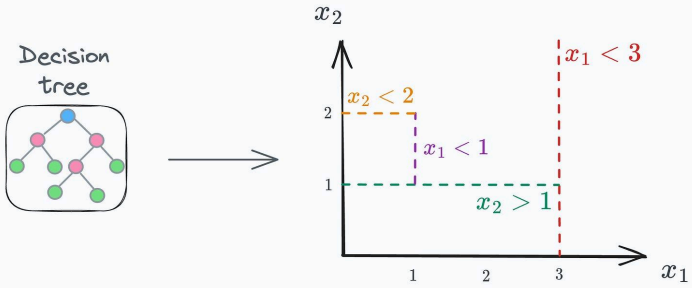
Next, we can train a new decision tree after obtaining an estimate for hyperparameter values. This will help us reduce the variance of the decision tree.

You can download the code notebook for the interactive decision tree here: https://bit.ly/4bBwY1p. Instructions are available in the notebook.

Next, let's understand a point of caution when using decision trees.

Why Decision Trees Must Be Thoroughly Inspected After Training

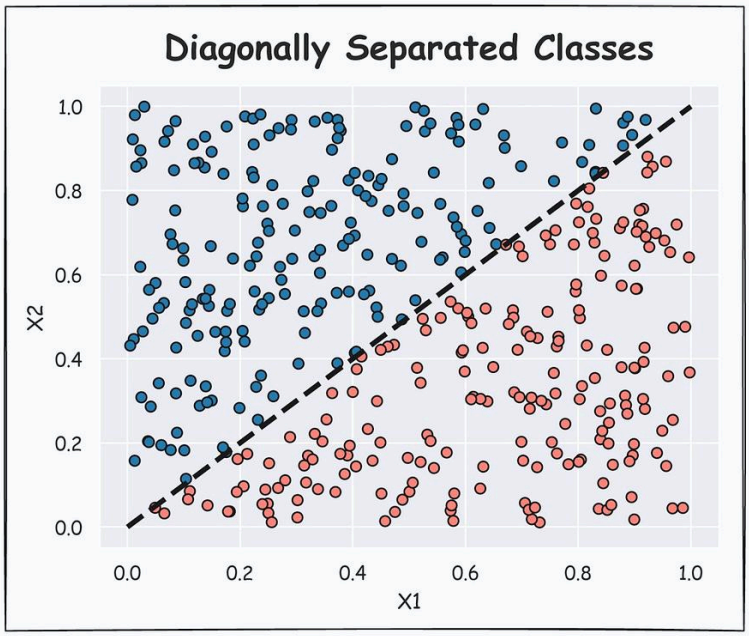
If we were to visualize the decision rules (the conditions evaluated at every node) of ANY decision tree, we would ALWAYS find them to be perpendicular to the feature axes, as depicted in the image.



In other words, every decision tree progressively segregates feature space based on such perpendicular boundaries to split the data.

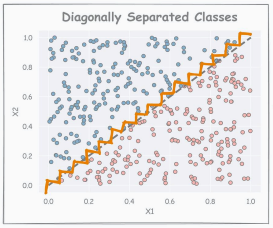
Of course, this is not a "problem" per se.

In fact, this perpendicular splitting is what makes it so powerful to perfectly overfit any dataset. However, this also brings up a pretty interesting point that is often overlooked when fitting decision trees. More specifically, what would happen if our dataset had a diagonal decision boundary, as depicted below:

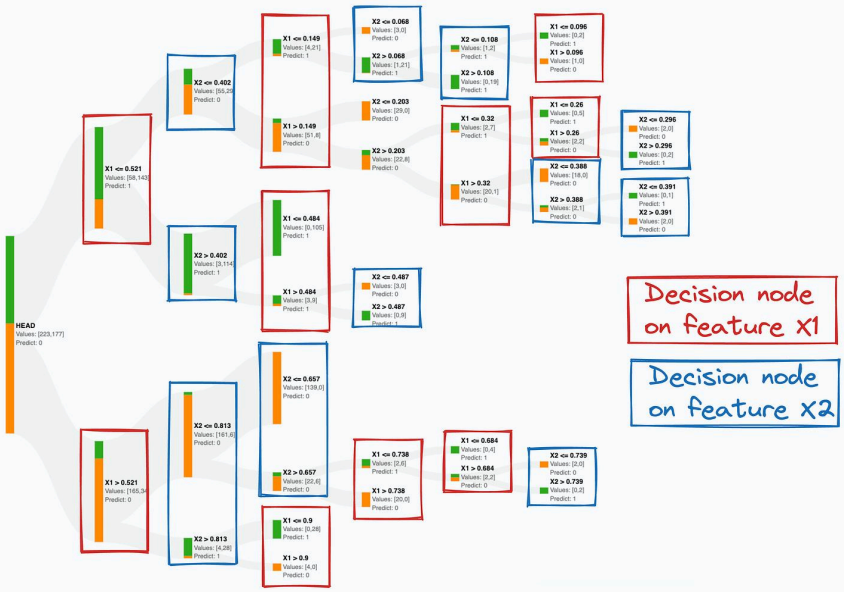


Dummy dataset with diagonal decision boundary

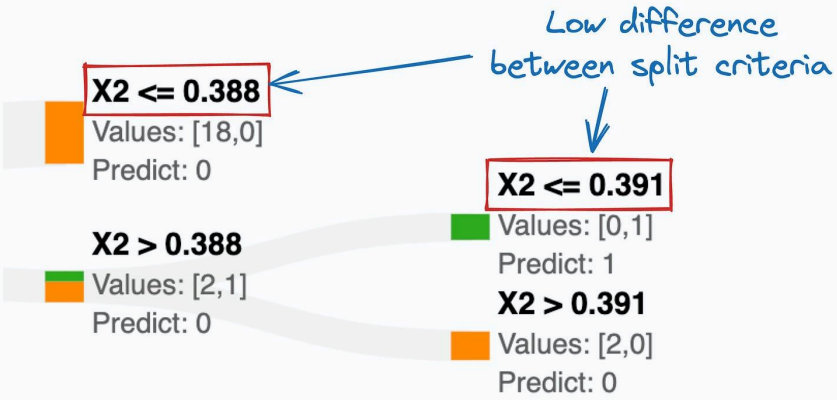
It is easy to guess that in such a case, the decision boundary learned by a decision tree is expected to appear as follows:



In fact, if we plot this decision tree, we notice that it creates so many splits just to fit this easily separable dataset, which a model like logistic regression, support vector machine (SVM), or even a small neural network can easily handle:



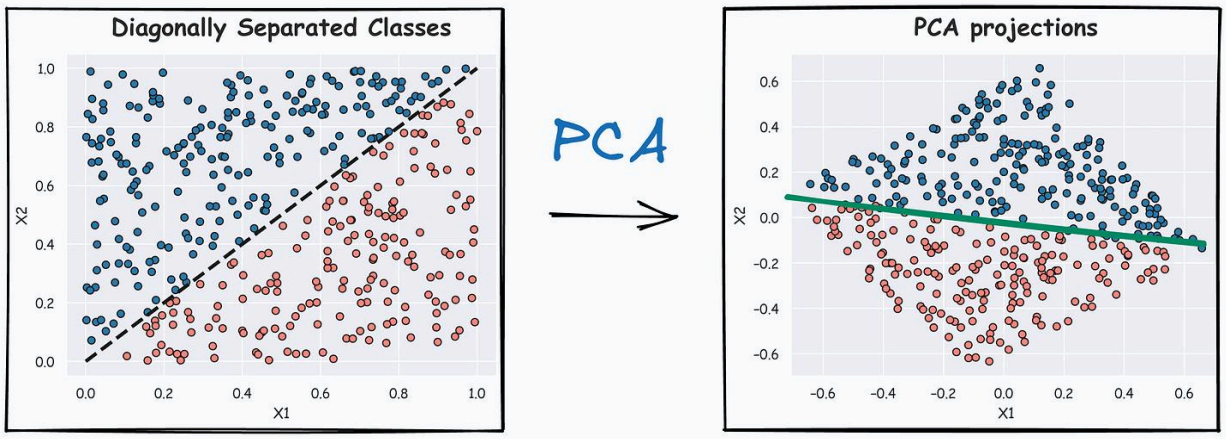
It becomes more evident if we zoom into this decision tree and notice how close the thresholds of its split conditions are:



This is a bit concerning because it clearly shows that the decision tree is meticulously trying to mimic a diagonal decision boundary, which hints that it might not be the best model to proceed with. To double-check this, I often do the following:

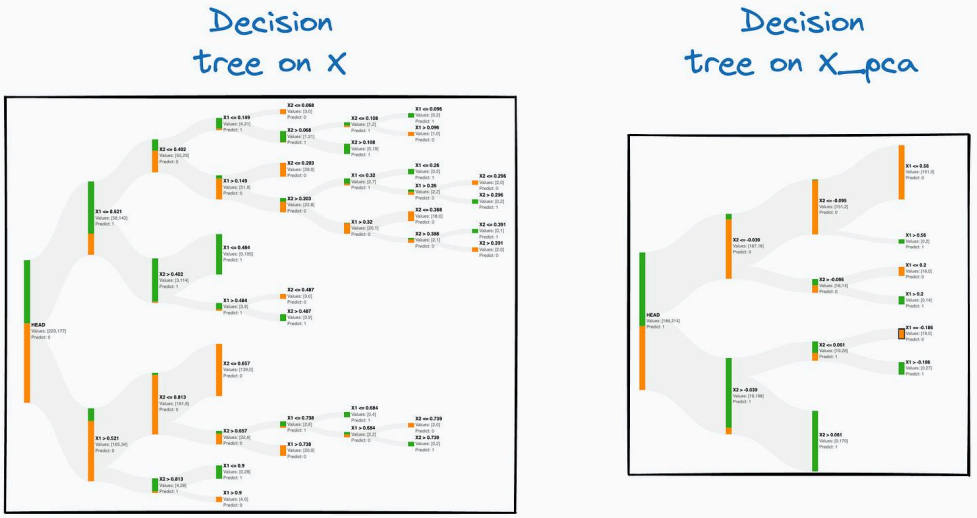
- Take the training data (X, y);
    - Shape of X: (n, m).
    - Shape of y: (n, 1).
- Run PCA on X to project data into an orthogonal space of m dimensions. This will give X_pca, whose shape will also be (n, m).
- Fit a decision tree on X_pca and visualize it (thankfully, decision trees are always visualizable).
- If the decision tree depth is significantly smaller in this case, it validates that there is a diagonal separation.

For instance, the PCA projections on the above dataset are shown below:



It is clear that the decision boundary on PCA projections is almost perpendicular to the X2` feature (the 2nd principal component).

Fitting a decision tree on this X_pca drastically reduces its depth, as depicted below:



This lets us determine that we might be better off using some other algorithm instead.

Or, we can spend some time engineering better features that the decision tree model can easily work with using its perpendicular data splits.

At this point, if you are thinking, why can't we use the decision tree trained on X_pca?

While nothing stops us from doing that, do note that PCA components are not interpretable, and maintaining feature interpretability can be important at times. Thus, whenever you train your next decision tree model, consider spending some time inspecting what it's doing.

Before ending this chapter

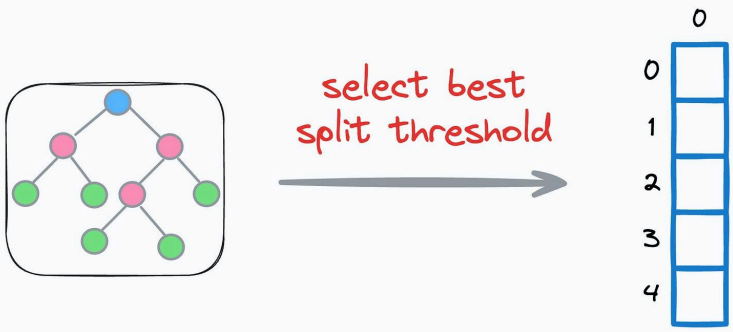
I don't intend to discourage the use of decision trees. They are the building blocks of some of the most powerful ensemble models we use today.

My point is to bring forward the structural formulation of decision trees and why/when they might not be an ideal algorithm to work with.

Decision Trees ALWAYS Overfit!

In addition to the above inspection, there's one more thing you need to be careful of when using decision trees. This is about overfitting.

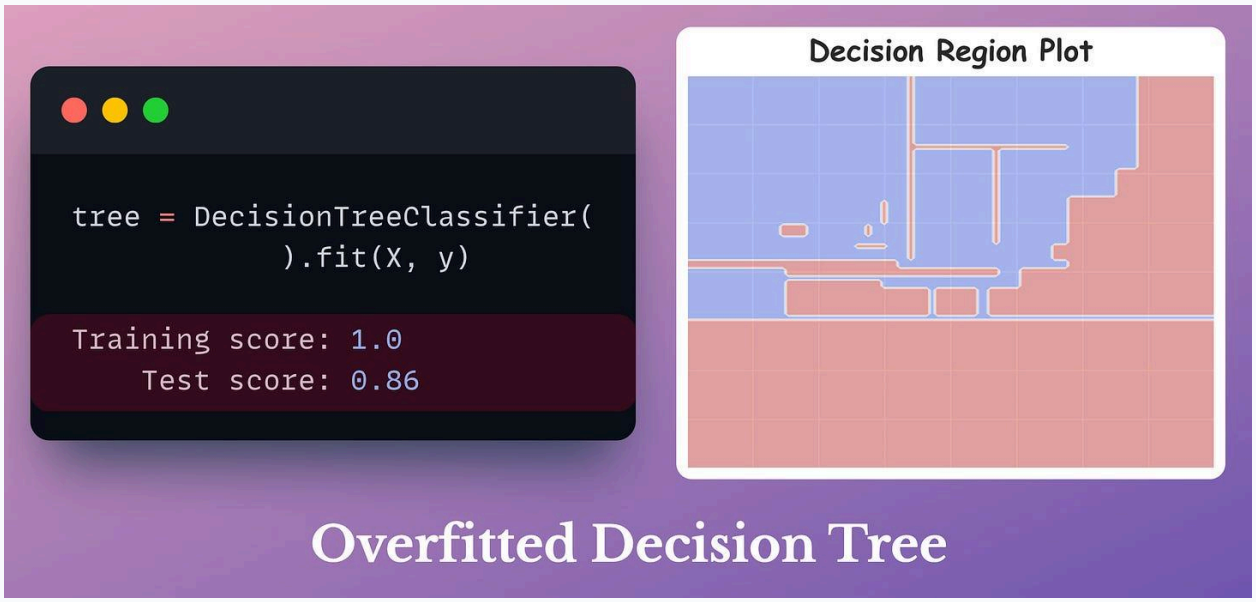
The thing is that, by default, a decision tree (in sklearn's implementation, for instance), is allowed to grow until all leaves are pure. This happens because a standard decision tree algorithm greedily selects the best split at each node.



This makes its nodes more and more pure as we traverse down the tree. As the model correctly classifies ALL training instances, it leads to 100% overfitting, and poor generalization.

For instance, consider this dummy dataset:

Fitting a decision tree on this dataset gives us the following decision region plot:



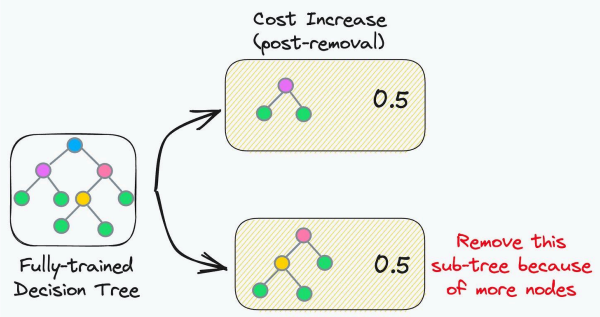
It is pretty evident from the decision region plot, the training and test accuracy that the model has entirely overfitted our dataset.

Cost-complexity-pruning (CCP) is an effective technique to prevent this.

CCP considers a combination of two factors for pruning a decision tree:

- Cost (C): Number of misclassifications
- Complexity (C): Number of nodes

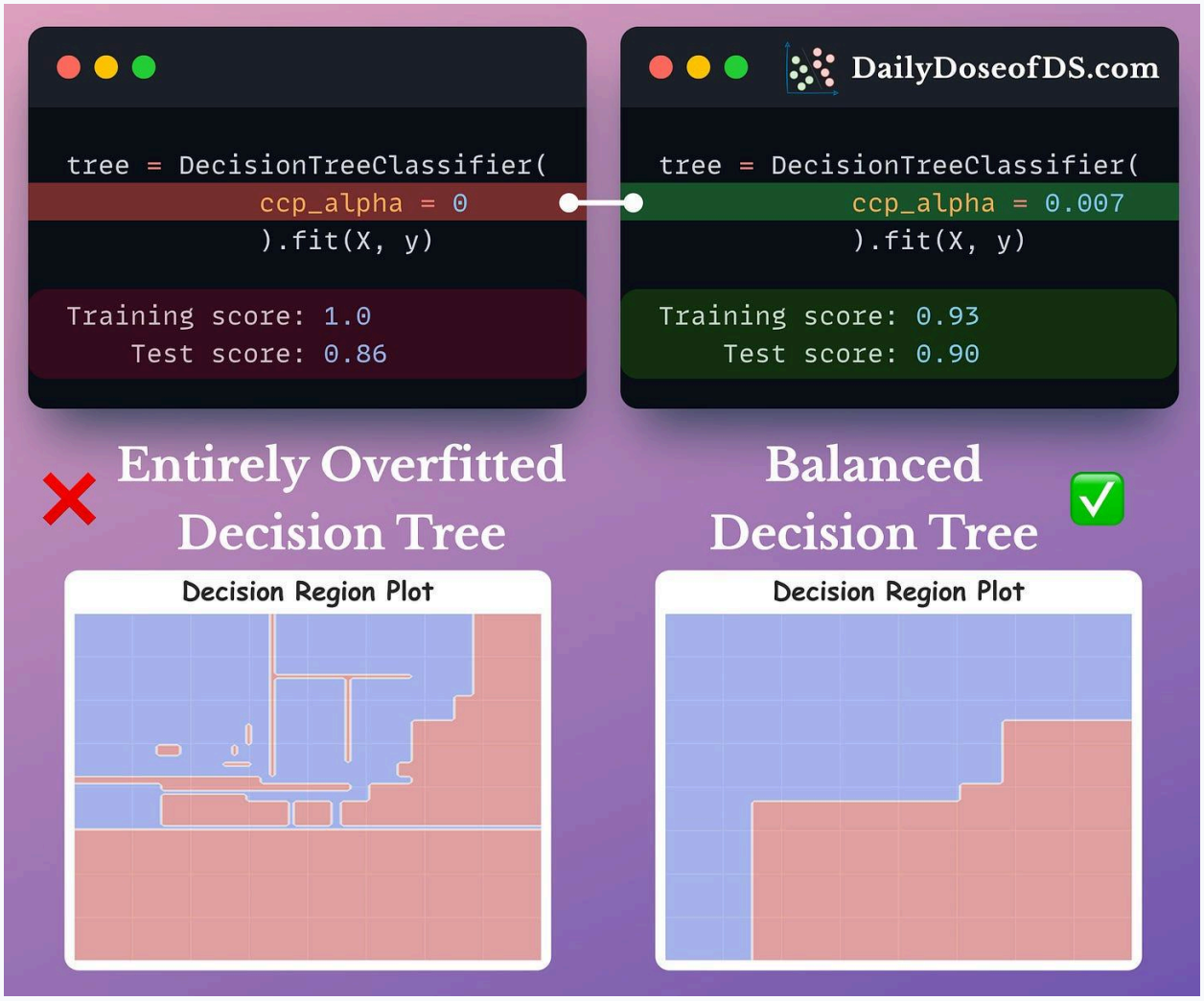
The core idea is to iteratively drop sub-trees, which, after removal, lead to a minimal increase in classification cost AND a maximum reduction of complexity (or nodes). In other words, if two sub-trees lead to a similar increase in classification cost, then it is wise to remove the sub-tree with more nodes.



In sklearn, you can control cost-complexity-pruning using the ccp_alpha parameter:

- large value of ccp_alpha → results in underfitting
- small value of ccp_alpha → results in overfitting

The objective is to determine the optimal value of ccp_alpha, which gives a better model. The effectiveness of cost-complexity-pruning is evident from the image below:



As depicted above, CCP results in a much simpler and acceptable decision region plot.

OOB Validation in Random Forest

After training an ML model on a training set, we always keep a held-out validation/test set for evaluation. I am sure you already know the purpose, so we won't discuss that.

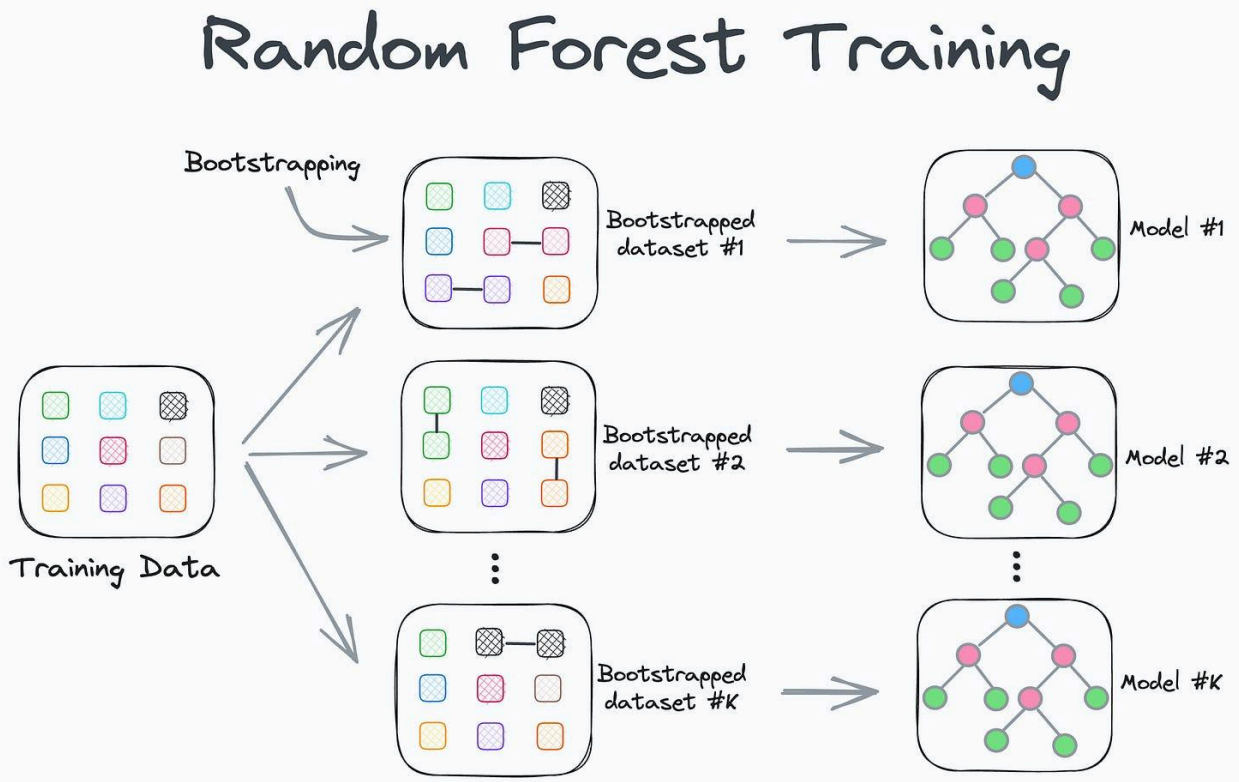
But do you know that random forests are an exception to that? In other words, one can somewhat "evaluate" a random forest using the training set itself.

Let's understand how.

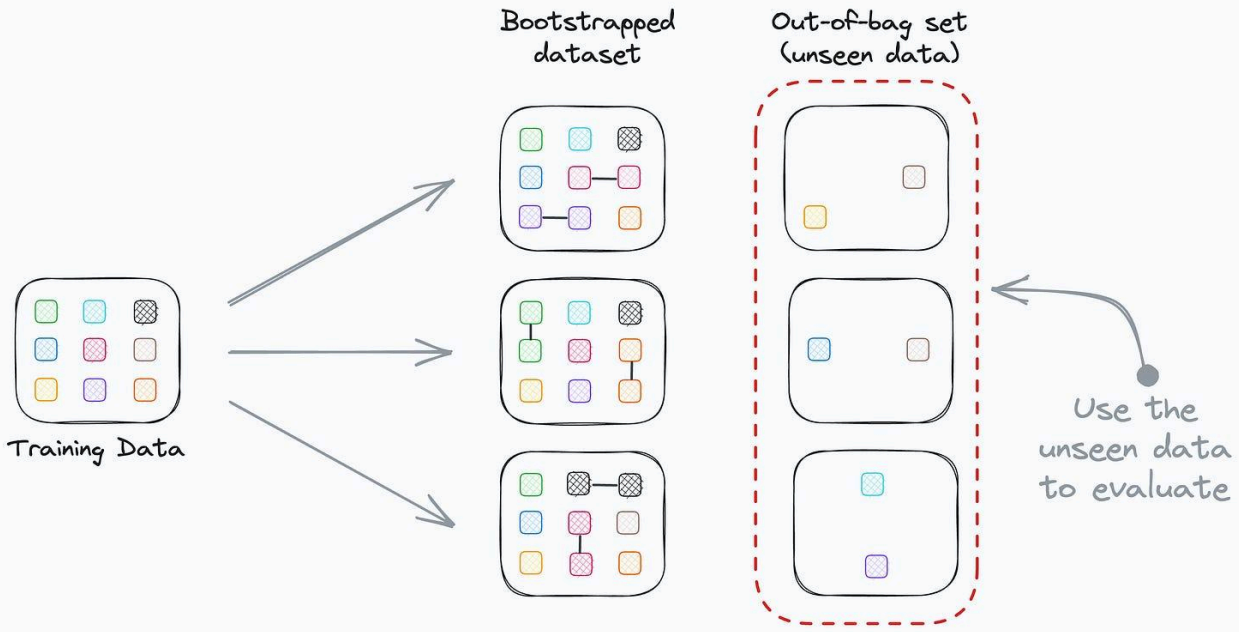
To recap, a random forest is trained as follows:

- First, we create different subsets of data with replacement (this process is called bootstrapping).
- Next, we train one decision tree per subset.
- Finally, we aggregate all predictions to get the final prediction.

This process is depicted below:

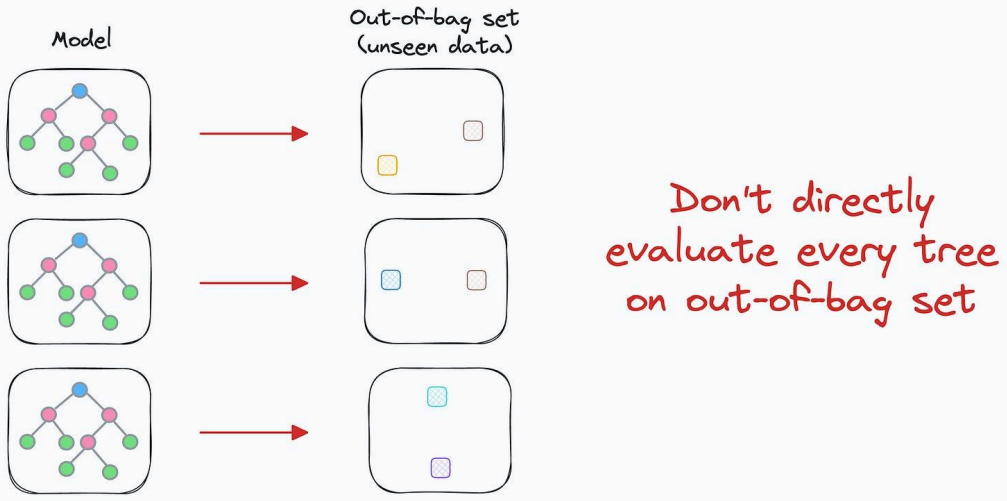


If we look closely above, every subset has some missing data points from the original training set.



We can use these observations to validate the model. This is also called out-of-bag validation. Calculating the out-of-bag score for the whole random forest is simple too.

But one thing to remember is that we CAN NOT evaluate individual decision trees on their specific out-of-bag sample and generate some sort of "aggregated score" for the entire random forest model.

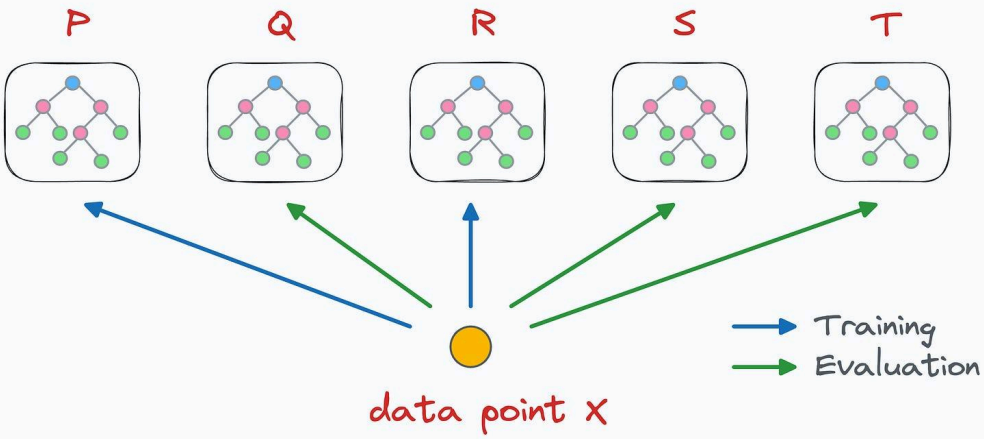


This is because a random forest is not about what a decision tree says individually. Instead, it's about what all decision trees say collectively.

So here's how we can generate the out-of-bag score for the random forest model. For every data point in the training set:

- Gather predictions from all decision trees that did not use it as a training data point.
- Aggregate predictions to get the final prediction.

For instance, consider a RF model with 5 decision trees → (P, Q, R, S, T). Say a specific data point X was used as a training data in decision trees P and R.



So we shall gather the out-of-bag prediction for data point X from decision trees Q, S and T.

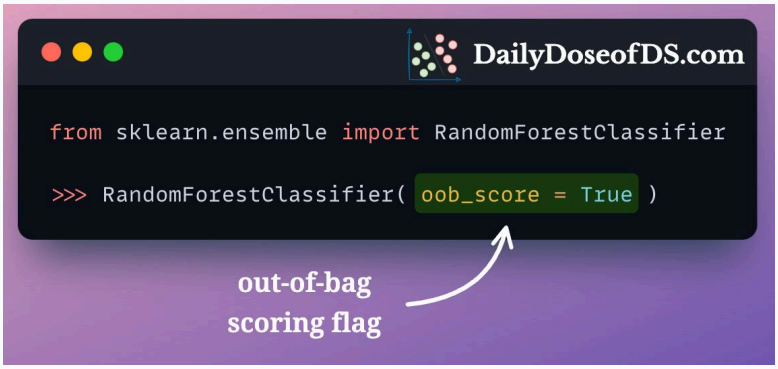
After obtaining out-of-bag predictions for all samples, we score them to get the out-of-bag score.

See...this technique allowed us to evaluate a random forest model on the training set. Of course, I don't want you to blindly adopt out-of-bag validation without understanding some of its advantages and considerations.

I have found out-of-bag validation to be particularly useful in the following situations:

- In low-data situations, out-of-bag validation prevents data splitting whilst obtaining a good proxy for model validation.
- In large-data situations, traditional cross-validation techniques are computationally expensive. Here, out-of-bag validation provides an efficient alternative. This is because, by its very nature, even cross-validation provides an out-of-fold metric. Out-of-bag validation is also based on a similar principle.

And, of course, an inherent advantage of out-of-bag validation is that it guarantees no data leakage. Luckily, out-of-bag validation is also neatly tied in sklearn's random forest implementation.



The most significant consideration about out-of-bag score is to use it with caution for model selection, model improvement, etc.

This is because if we do, we typically tend to overfit the out-of-bag score as the model is essentially being tuned to perform well on the data points that were left out during its training. And if we consistently improve the model based on the out-of-bag score, we obtain an overly optimistic evaluation of its generalization performance.

If I were to share just one lesson here based on my experience, it would be that if we don't have a true (and entirely different) held-out set for validation, we will overfit to some extent.

The decisions made may be too specific to the out-of-bag sample and may not generalize well to new data.

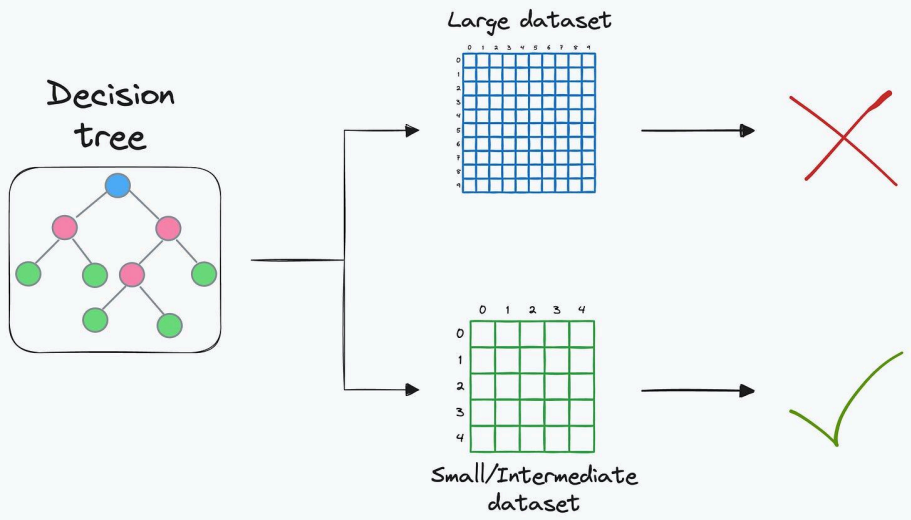
Train Random Forest on Large Datasets

Most classical ML algorithms cannot be trained with a batch implementation. This limits their usage to only small/intermediate datasets. For instance, this is the list of sklearn implementations that support a batch API:

It's pretty small, isn't it?

This is concerning because, in the enterprise space, the data is primarily tabular. Classical ML techniques, such as tree-based ensemble methods, are frequently used for modeling.

However, typical implementations of these models are not "big-data-friendly" because they require the entire dataset to be present in memory.



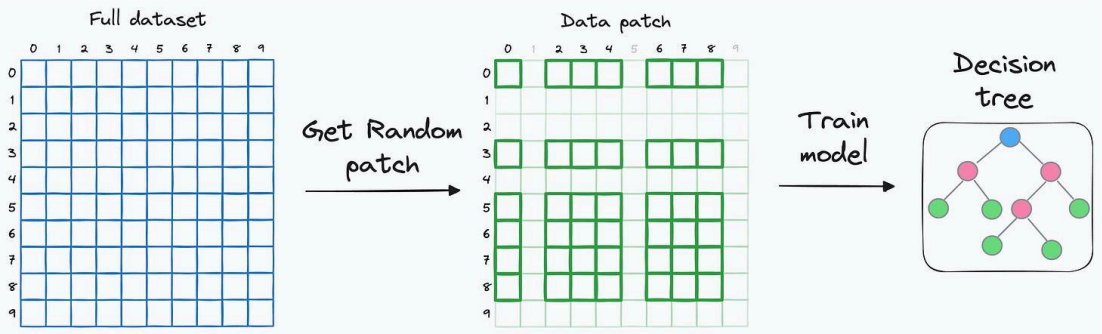
There are two ways to approach this:

1. Use big-data frameworks like Spark MLlib to train them.
2. There's one more way, which Dr. Gilles Louppe discussed in his PhD thesis — Understanding Random Forests.

Here's what he proposed.

Random Patches

Before explaining, note that this approach will only work in an ensemble setting. So, you would have to train multiple models. The idea is to sample random data patches (rows and columns) and train a tree model on them.



Repeat this step multiple times by generating different patches of data randomly to obtain the entire random forest model.

The efficacy?

The thesis presents many benchmarks (check pages 174 and 178 if you need more details) on 13 datasets, and the results are shown below:

From left to right → cifar10, mnist3v8, mnist4v9, mnist, isolet, arcene, breast2, madelon, marti, reged, second, this, and sido.

From the above image, it is clear that in most cases, the random patches approach performs better than the traditional random forest.

In other cases, there wasn't a significant difference in performance.

And this is how we can train a random forest model on large datasets that do not fit into memory.

Why does it work?

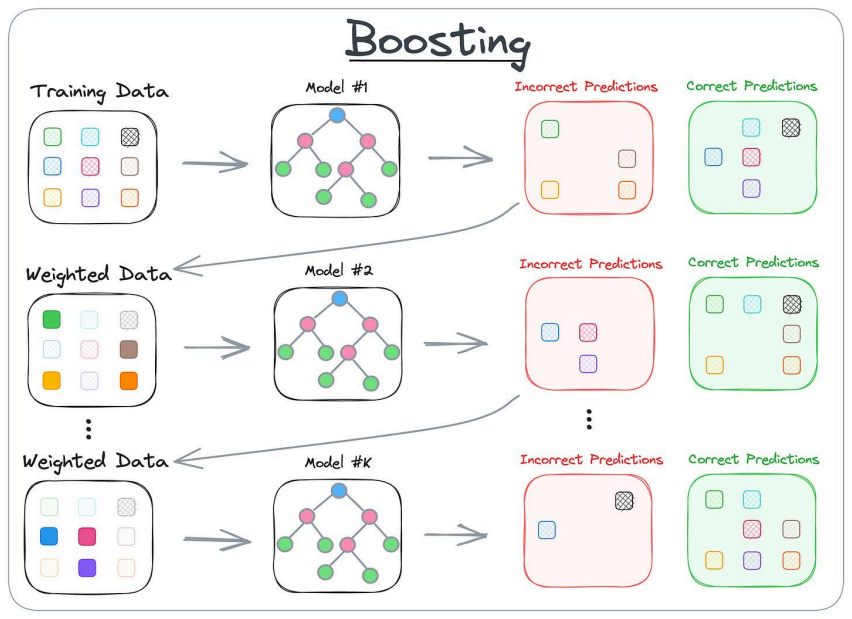
While the thesis did not provide a clear intuition behind this, I can understand why such an approach would still be as effective as random forest.

In a gist, building trees that are as different as possible guarantees a greater reduction in variance.

In this case, the dataset overlap between any two trees is NOT expected to be huge compared to the typical random forest. This aids in the Bagging objective.

A Visual Guide to AdaBoost

The following visual summarizes how Boosting models work:



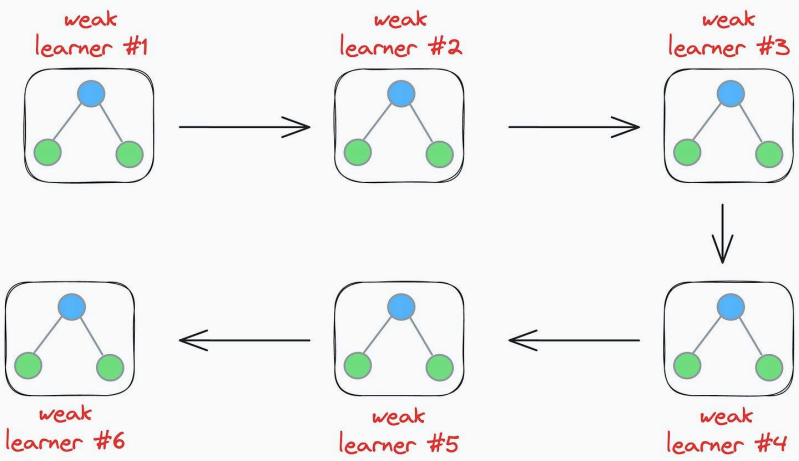
- Boosting is an iterative training process.
- The subsequent model puts more focus on misclassified samples from the previous model.
- The final prediction is a weighted combination of all predictions

However, many find it difficult to understand how this model is precisely trained and how instances are reweighed for subsequent models. AdaBoost is a common Boosting model, so in this chapter, let's understand how it works.

AdaBoost internal working

The core idea behind Adaboost is to train many weak learners to build a more powerful model. This technique is also called ensembling.

Specifically talking about Adaboost, the weak classifiers progressively learn from the previous model's mistakes, creating a powerful model when considered as a whole.



These weak learners are usually decision trees.

Let me make it more clear by implementing AdaBoost using the `DecisionTreeClassifier` class from sklearn.

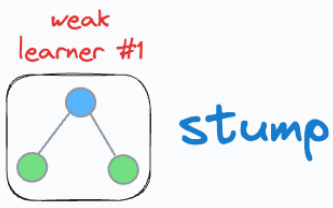
Consider we have the following classification dataset:

feature 1	feature 2	feature 3	target	weight
1.2	2.3	True	class A	0.2
-0.3	0.6	False	class B	0.2
0.7	1.8	False	class A	0.2
4.5	-3.5	True	class B	0.2
-0.4	0.4	True	class A	0.2

To begin, every row has an equal weight, and it is equal to (1/n), where n is the number of training instances.

Step 1: Train a weak learner

In Adaboost, every decision tree has a unit depth, and they are also called stumps.



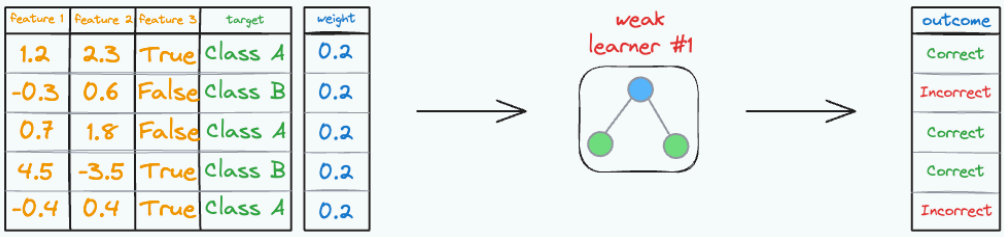
Thus, we define DecisionTreeClassifier with a max_depth of 1, and train it on the above dataset.

Custom-Adaboost.py

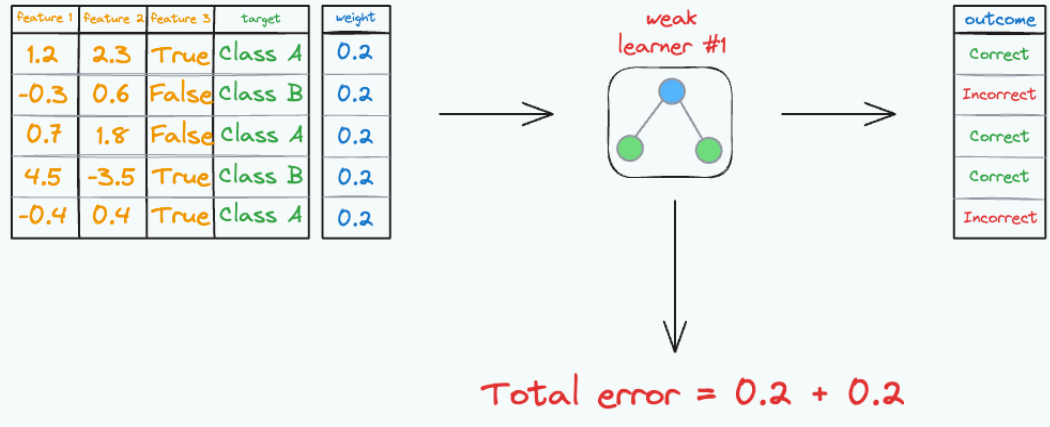
```
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier(max_depth=1)

tree.fit(X, y)
```

Step 2: Calculate the learner's cost

Of course, there will be some correct and incorrect predictions.



The total cost (or error/loss) of this specific weak learner is the sum of the weights of the incorrect predictions. In our case, we have two errors, so the total error is:



Now, as discussed above, the idea is to let the weak learn progressively learn from previous learner's mistakes. So, going ahead, we want the subsequent model to focus more on the incorrect predictions produced earlier.

Here's how we do this:

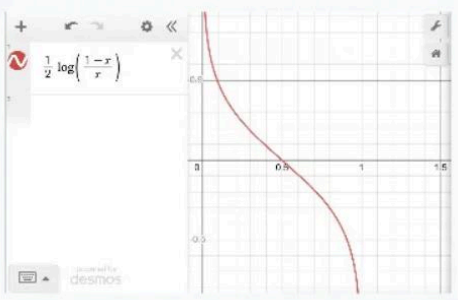
Step 3: Calculate the learner's importance

First, we determine the importance of the weak learner. Quite intuitively, we want the importance to be inversely related to the above error.

- If the weak learner has a high error, it must have a lower importance.
- If the weak learner has a low error, it must have a higher importance.

One choice is the following function:

$$\text{Importance} = \frac{1}{2} \cdot log(\frac{1 - \text{Total error}}{\text{Total error}})$$



- This function is only defined between [0,1].
- When the error is high (~1), this means there were no correct predictions → This gives a negative importance to the weak learner.
- When the error is low (~0), this means there were no incorrect predictions → This gives a positive importance to the weak learner.

If you feel there could be a better function to use here, you are free to use that and call it your own Boosting algorithm.

Now, we have the importance of the learner.

$$\text{Importance} = \frac{1}{2} \cdot log(\frac{1 - \text{Total error}}{\text{Total error}})$$

The importance value is used during model inference to weigh the predictions from the weak learners. So the next step is to...

Step 4: Reweigh the training instances

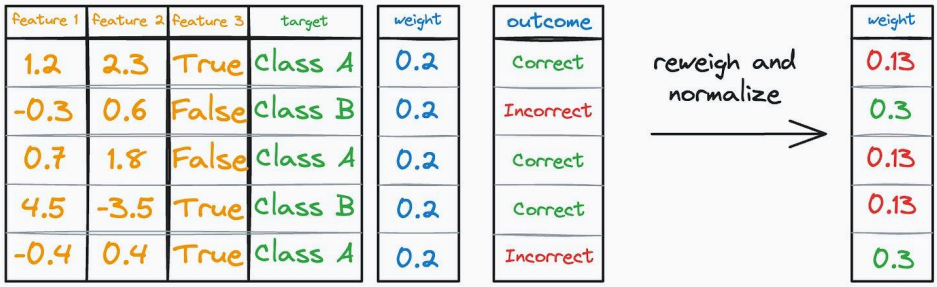
All the correct predictions are weighed down as follows:

$$w := w * e^{\text{-Importance}}$$

And all the incorrect predictions are weighed up as follows:

$$w := w * e^{\text{Importance}}$$

Once done, we normalize the new weights to add up to one.



That's it!

Step 5: Sample from reweighed dataset

From step 4, we have the reweighed dataset.

We sample instances from this dataset in proportion to the new weights to create a new dataset.

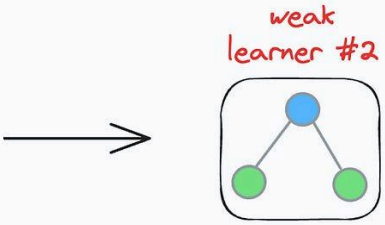
feature 1	feature 2	feature 3	target	weight
1.2	2.3	True	Class A	0.13
-0.3	0.6	False	Class B	0.3
-0.3	0.6	False	Class B	0.3
-0.4	0.4	True	Class A	0.3
-0.4	0.4	True	Class A	0.3

Repeated

Repeated

Next, go back to step 1 — Train the next weak learner.

feature 1	feature 2	feature 3	target	weight
1.2	2.3	True	Class A	0.13
-0.3	0.6	False	Class B	0.3
-0.3	0.6	False	Class B	0.3
-0.4	0.4	True	Class A	0.3
-0.4	0.4	True	Class A	0.3



And repeat the above process over and over for some pre-defined max iterations.

That's how we build the AdaBoost model.

All we have to do is consider the errors from the previous model, reweigh and sample the training instances for the next model, and repeat.

Dimensionality Reduction

The Utility of 'Variance' in PCA

The core objective of PCA is to retain the maximum variance of the original data while reducing the dimensionality. The rationale is that if we retain variance, we will retain maximum information.

But why?

Many people struggle to intuitively understand the motivation for using "variance" here. In other words:

Why retaining maximum variance is an indicator for retaining maximum information?

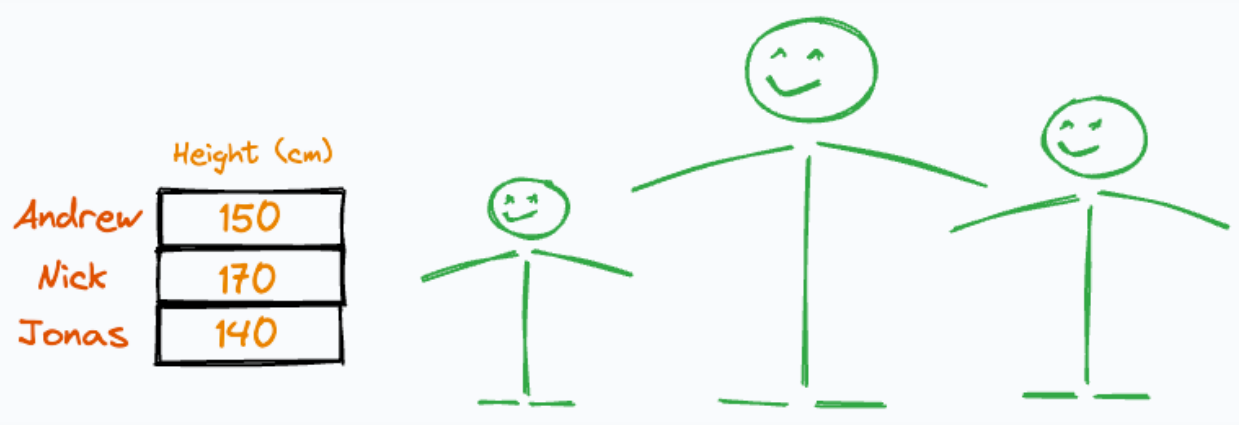
This chapter provides an intuitive explanation of this.

Imagine someone gave us the following weight and height information about three individuals:

	Height (cm)	Weight (kg)
Andrew	150	71
Nick	170	73
Jonas	140	70

It's clear that the height column has more variation than weight.

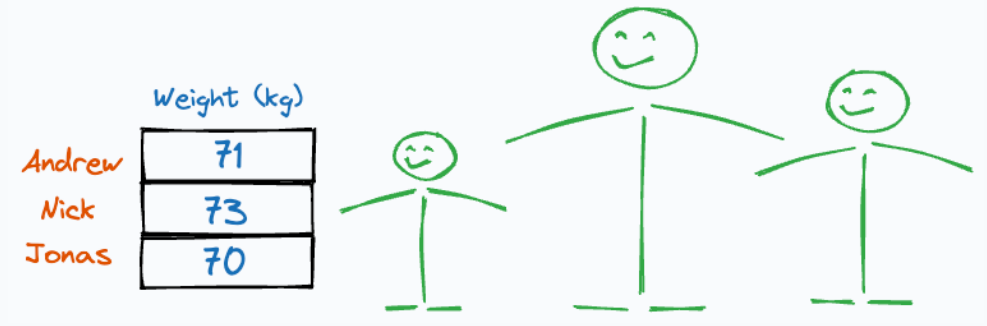
Thus, even if we discard the weight column, we can still identify these people solely based on their heights.



Dropping the weight column

- The one in the middle is Nick.
- The leftmost person is Jonas.
- The rightmost one is Andrew.

That was super simple. But what if we discarded the height column instead?



Can you identify them now?

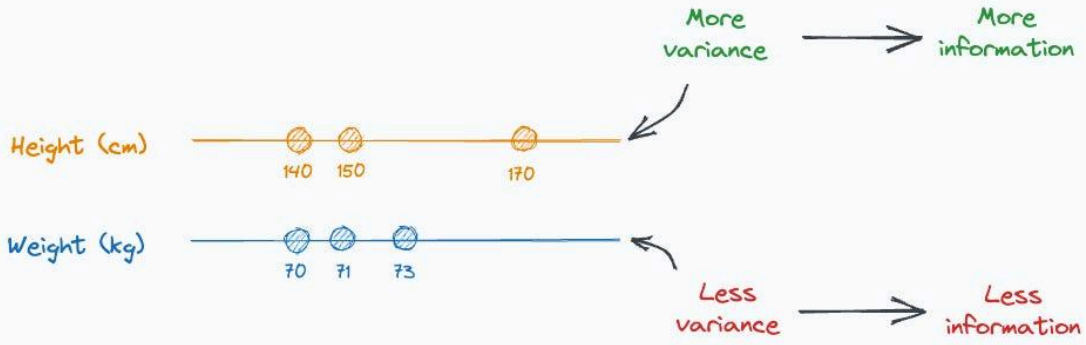
No, right?

And why?

This is because their heights have more variation than their weights.

And it's clear from the above example that, typically, if a column has more variation, it holds more information.

That is the core idea PCA is built around, and that is why it tries to retain maximum data variance. Simply put, PCA is devised on the premise that more variance means more information.

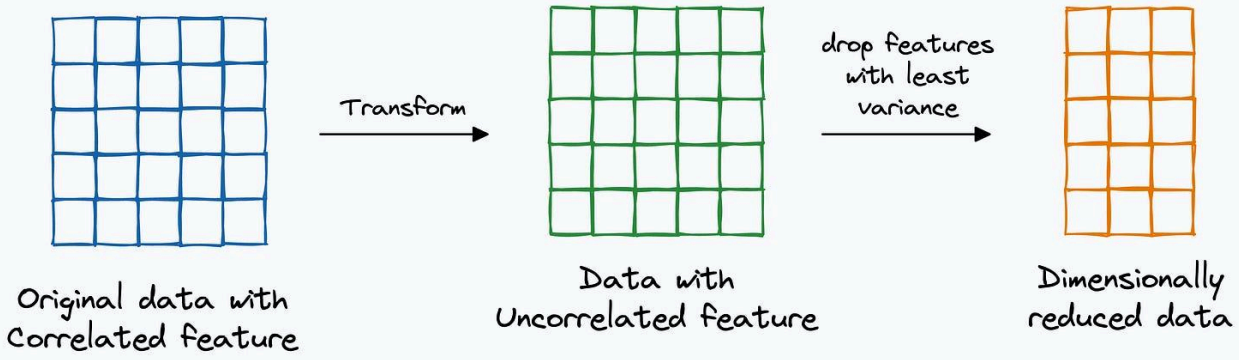


More variance means more information and less variance means less information Thus, during dimensionality reduction, we can (somewhat) say that we are retaining maximum information if we retain maximum variance.

Of course, as we are using variance, this also means that it can be easily influenced by outliers. That is why we say that PCA is influenced by outliers.

As a concluding note, always remember that when using PCA, we don't just measure column-wise variance and drop the columns with the least variance.

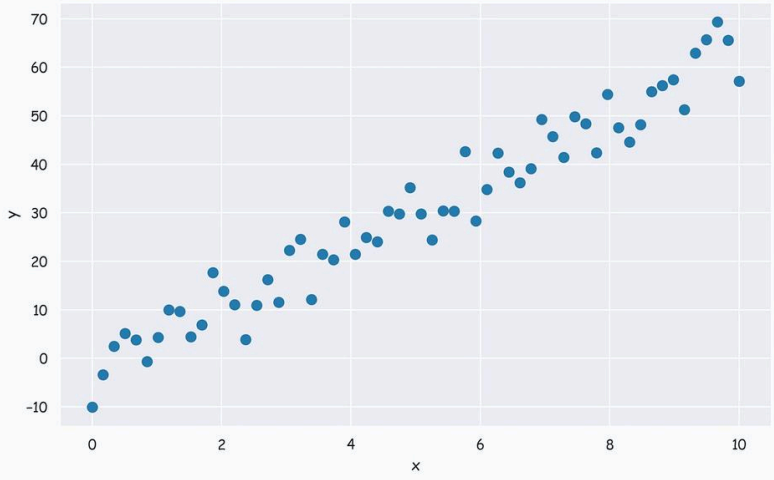
Instead, we must first transform the data to create uncorrelated features. After that, we drop the new features based on their variance.



KernelPCA vs. PCA

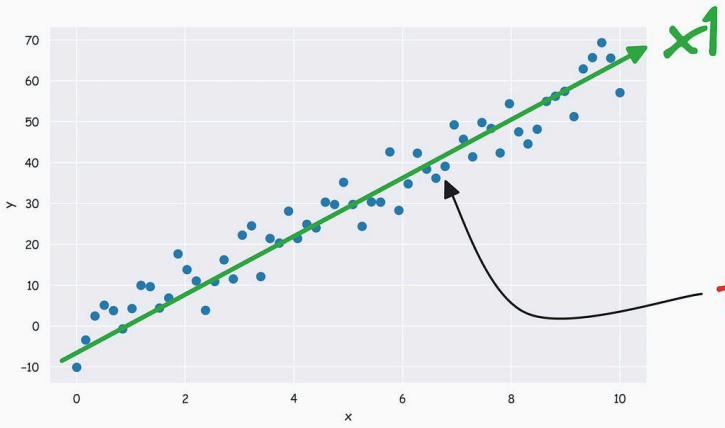
During dimensionality reduction, principal component analysis (PCA) tries to find a low-dimensional linear subspace that the given data conforms to.

For instance, consider the following dummy dataset:



Dummy dataset

It's pretty clear from the above visual that there is a linear subspace along which the data could be represented while retaining maximum data variance. This is shown below:



A LINEAR subspace to represent the data

But what if our data conforms to a low-dimensional yet non-linear subspace.