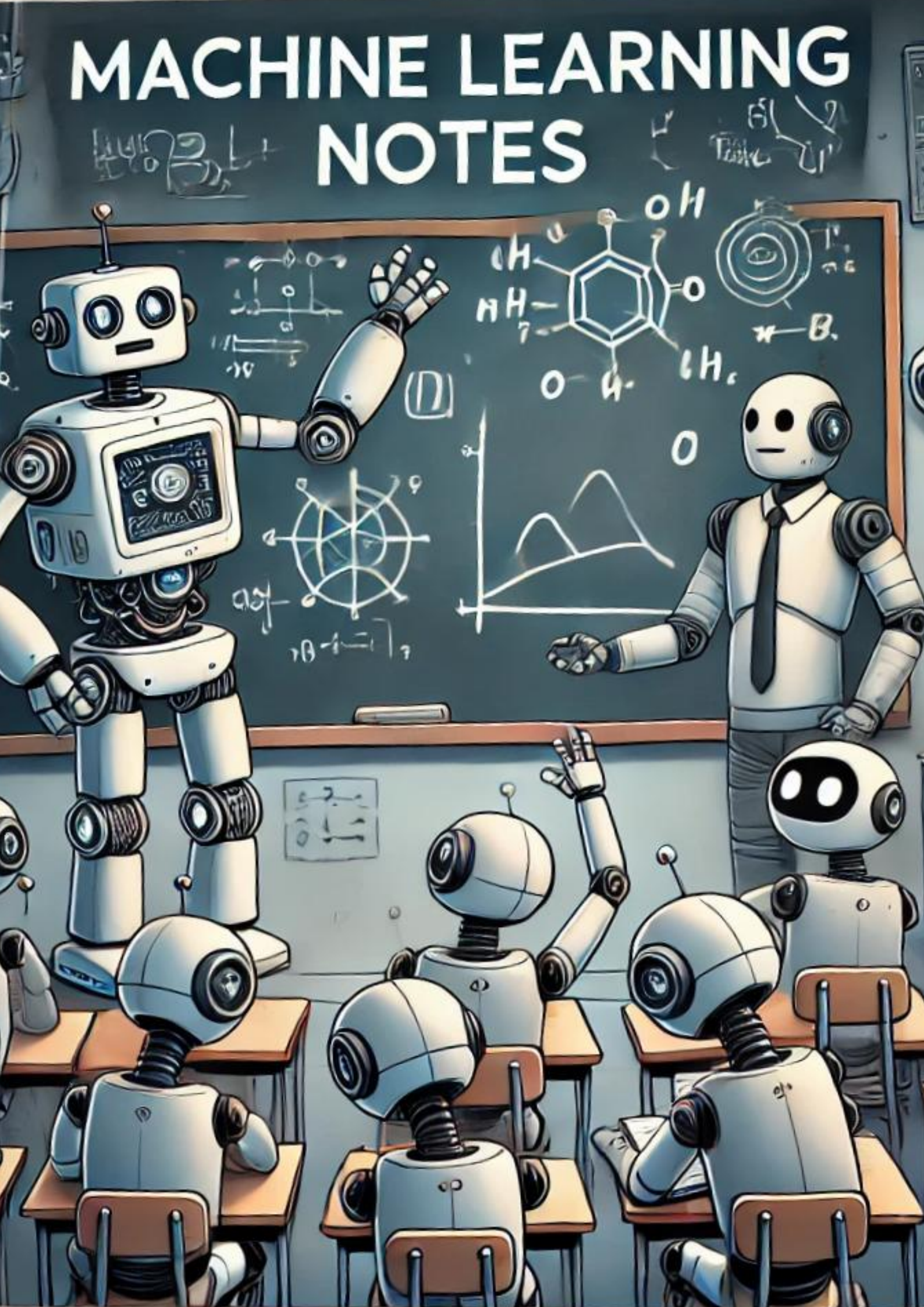
MACHINE LEARNING NOTES



Foundations of Machine Learning

Notes of Luigi Battista

A.Y. 2024/25

Contents

Introduction 6

Introduction to Machine Learning 8

Supervised Learning 10

1 Linear Regression 11

1.1 Simple Linear Regression 11

1.2 Gradient Descent 16

1.3 Multiple Linear Regression 20

1.4 Deriving Explicit form for Partial Derivatives of the MSE cost function w.r.t to parameters in Linear Regression 21

1.5 Batch GD vs Stochastic GD vs Mini Batch GD 23

1.6 Polynomial Regression 26

1.7 Feature Scaling and Normalization Techniques 28

1.8 Normal Equations Method 30

1.9 Probabilistic Interpretation 33

2 Logistic Regression 37

2.1 Classification 37

2.2 Introduction to Logistic Regression 38

2.3 Multiclass Classification (One-vs-All) 41

3 Fitting 42

3.1 Bias and Variance 43

3.2 Bias and Variance Trade-off 44

4 Regularization 49

4.1 Intuition Behind Regularization 49

4.2 Regularization in Linear Regression 50

4.3 Regularization in Logistic Regression 51

4.4 Regularization with L2-norm and L1-norm 52

4.5 Regularization VS Bias/Variance 53

5 How to Build an ML System 55

5.1 Data Analysis and Pre-Processing 55

5.2 Hypothesis Evaluation 57

6 Diagnostic and debugging 62

6.1 Learning Curves 62

6.2 Evaluation Metrics 63
6.3 ROC & AUC 66
6.4 Paired t-Test 69
6.5 Coefficient of Determination - $R2$ 72
6.6 Hyperparameter Tuning 72
7 Neural Networks 75
7.1 Non-Linear Hypothesis 75
7.2 Inspiration from Biological Neuron 78
7.3 Neuron Model: Logistic Unit 79
7.4 Forward Propagation: Vectorized Implementation 83
7.5 Non-Linear Classification Example: XOR 85
7.6 Neural Networks for Multi-Class Classification 89
7.7 Neural Network Cost Function 90
7.8 Neural Network Backpropagation 92
7.9 Parameters Initialization 109
7.10 Activation Functions 110
8 Support Vector Machines 115
8.1 SVM Intuition 115
8.2 Maximum Margin 116
8.3 Lagrange Duality 119
8.4 SVM Optimization 123
8.5 SVM Cost Function 129
8.6 Soft-Margin SVM 132
8.7 Kernel SVM 133
8.8 Generalized Linear Models (GLMs) 138
9 Decision Trees 140
9.1 Regression Tree 141
9.2 Classification Tree 149
9.3 Numerical Values in Classification Trees 155
9.4 Missing Values in Decision Trees 157
9.5 Pruning Regression Trees 158
10 Random Forests 164

10.1 Classify a New Sample with “Bagging” 166
10.2 Evaluate the Accuracy of a Random Forest: Out-of-Bag Error 166
10.3 Missing Values in Random Forest 167
Unsupervised Learning 174
11 Clustering 174
11.1 K-Means 175
11.2 K-Medoids 180
11.3 Gaussian Mixture of Models (GMMs) 182
11.4 Choosing the Value of K 190
11.5 Hierarchical Clustering 202
11.6 DBSCAN 205
11.7 HDBSCAN 208
11.8 GMM Use Case: Anomaly Detection 220
12 Dimensionality Reduction 222
12.1 The Curse of Dimensionality 222
12.2 Introduction to Dimensionality Reduction 223
12.3 Principal component Analysis (PCA) 225
12.4 Kernel PCA 240
12.5 Independent Components Analysis (ICA) 244
12.6 Embeddings 253
12.7 Autoencoders 266
Some Other Important Principles in Machine Learning 271
Occam’s (Ockham’s) Razor 271
No free Lunch (NFL) Theorem 271

Introduction

These notes are enriched with my personal insights and reflections from the ***Foundations of Machine Learning*** course (A.Y. 2024/25), taught by Prof. Tommaso Di Noia as part of the Master's program in Artificial Intelligence and Data Science at PoliBa – Polytechnic University of Bari.

I created these notes with the intention of helping future students gain a deeper understanding of Machine Learning. During discussions with fellow students, I noticed frequent confusion or gaps in comprehension. This often stemmed from certain topics being not thoroughly explained during lectures or in the provided material.

In writing these notes, I aimed to align them as closely as possible with the course slides and lectures. However, since some slides contained errors, I have supplemented them with the necessary details to ensure a comprehensive understanding of the subject matter.

In particular:

- I followed a clear and consistent notation in all chapters.
- Whenever possible, I included sources to allow readers to verify the information independently.
- I adhered to the principle of "**no topic left unexplained**". Every concept is presented with thorough explanations to ensure clarity and understanding.
- I used a color-coded system for emphasis:
  - **Light blue** for simple comments.
  - **Orange** for noteworthy aspects or additional details.
  - **Red** for critical points and important notes that require attention.

I hope these notes will provide a solid foundation in Machine Learning that will serve as a stepping stone for future courses, such as *Deep Learning* or *Information Retrieval and Personalization*.

Sources for These Notes:

The material I used to write these notes includes:

- ***Machine Learning: A Probabilistic Perspective*** **by Kevin P. Murphy** (the primary course textbook). This is an excellent, comprehensive book that covers a wide range of machine learning concepts in great depth. However, it's worth noting that the book's level of detail and mathematical rigor can make it quite challenging. While it isn't perfectly aligned with the course material—since the course takes a more simplified approach—it's a fantastic resource if you want to explore topics more thoroughly and build a solid theoretical foundation.

- ***Pattern Recognition and Machine Learning*** **by Christopher M. Bishop**: I personally prefer the flow of explanations in this book, as it's slightly less formal and easier to follow in some areas. It's particularly helpful for gaining intuition about concepts without being overwhelmed by heavy mathematical formalism. If you're looking for an alternative to Murphy's book, this one is a great option.

- The **CS229 Lecture Notes** **by Andrew Ng**: Much of this course is inspired by Andrew Ng's earlier CS229 course, and his notes are closely aligned with the material we cover. I've drawn heavily from these notes in writing mine, so they're a highly relevant and useful resource for understanding the concepts presented here.

- Additional papers or videos as needed.

Although I didn't directly reference it while writing these notes, I also recommend *An Introduction to Statistical Learning* by Gareth James et al. as a fantastic complementary resource.

Advice on Studying Machine Learning:

I recommend starting with the course notes I've written as your primary resource since they provide a focused foundation for the material covered in the course. At the same time, you can explore the suggested books to dive deeper into specific topics or gain a broader perspective.

After studying a theoretical concept, it's crucial to immediately apply it through coding exercises. Practical implementation not only reinforces your knowledge but also allows you to see how the concepts function in real-world scenarios. To support your learning, I've created a repository on my personal GitHub profile called **ML Course**, where I've rewritten the code covered in the lectures in a clear, organized manner. The repository includes detailed comments and reflections to enhance comprehension. If you find the repository helpful, I'd greatly appreciate it if you could **star it** or share it with others who might benefit! Moreover, in the **Theory** folder, you will find the latest updated version of these notes. Please remember to check for any new updates.

While this course focuses on understanding the principles behind machine learning algorithms, it's important to recognize that machine learning is inherently tied to working with data. Make sure to spend time understanding your data — learn how to visualize it, extract insights, and interpret its characteristics. Developing this intuition for data is as essential as mastering the algorithms themselves.

Disclaimer: These notes are not intended to be a definitive authority on the course material but rather a valuable resource for studying the theoretical aspects. Therefore: **I AM NOT RESPONSIBLE FOR ANY BAD GRADES OR EXAM FAILURES**.

With that said, let's embark on the study of this fascinating subject.

Introduction to Machine Learning

Machine learning (ML) arose from the need to analyze data automatically and make decisions similar to human judgment. At its core, ML refers to a set of methods that enable computers to detect patterns in data and use these patterns to predict future outcomes or assist in decision-making under uncertainty. Given the inherent uncertainty in data, probabilistic and statistical models play a critical role in helping ML algorithms make informed predictions or decisions.

Definitions of Machine Learning:

- **Herbert Alexander Simon**: "Machine Learning is concerned with computer programs that automatically improve their performance through experience".

- **Tom Mitchell (widely considered a comprehensive definition)**: "A computer program is said to learn from experience (E) with some class of tasks (T) and a performance measure (P) if its performance at tasks in T as measured by P improves with E" → In simpler terms, **learning** means improving at a given task (T) based on experience (E), and this improvement is measured by a performance metric (P).

  Example:
  Consider an email filtering system that learns to classify emails as spam or not spam. In this case we will have:

  - **Task (T)**: Classifying emails as spam or not spam.
  - **Experience (E)**: Watching how you label emails as spam or not spam over time.
  - **Performance (P)**: The number (or fraction) of emails correctly classified by the system as spam or not spam.

  Note: It's important to underline that the computer program (that later we will call model) learns to improve to a **specific** class of tasks, not to all tasks! This will be underlined at the end of the course by the **No Free Lunch theorem**.

- Personally, I also like the definition given by Wikipedia: "Machine learning is a field of study in artificial intelligence concerned with the development and study of **statistical algorithms** that can learn from data and generalize to unseen data, and thus perform tasks without explicit instructions".

Categories of Machine Learning

Machine learning can be categorized based on how data is used during training:

- **Supervised Learning (SL):** In supervised learning, the algorithm learns from a **labeled** training dataset $D = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$ where both the input data and the corresponding correct outputs (labels) are provided. Usually, $x^{(i)}$ is a vector where each component is called **feature**, **attribute,** or **covariate**, and $y^{(i)}$ is the desired output, also known as **target**. In particular, the best-known tasks belonging to this class of problems are:
    - **Regression**: Predicting continuous values (e.g., forecasting temperatures).
    - **Classification**: Assigning discrete labels to inputs (e.g., categorizing emails as spam or not spam).
- **Unsupervised Learning (USL)**: In unsupervised learning, the algorithm is provided with input data but without any labeled output, so $D = \{x^{(i)}\}_{i=1}^{m}$. The objective is to discover patterns or structures in the data, such as identifying groups or clusters of similar items. A common example of unsupervised learning is:
    - **Clustering**: Grouping data points with similar characteristics (e.g., customer segmentation in marketing).

Other fields related to ML are:

- **Deep learning (DL)**: It is a subset of machine learning that uses neural networks with many layers (hence "deep"). It excels in handling large amounts of unstructured data such as images, audio, text.
- **Reinforcement Learning (RL):** In reinforcement learning, an agent learns to interact with an environment and take actions to maximize cumulative rewards over time. The agent receives feedback in the form of rewards or penalties and adjusts its actions accordingly. A classic analogy is training a dog through rewards and corrections (positive or negative reinforcement).

+ Recommender Systems: Recommender systems aim to predict a user's preference for certain items, given historical interactions (e.g., ratings, clicks). The input is often represented as a "user-item" matrix, and the system tries to recommend items that the user may rate highly or enjoy.

ML models can also be classified based on the approach they use to model data relationships:

- **Discriminative Models:** Discriminative models are trained to predict the most likely output given a specific input. In other words, their goal is to find the relationship between the input data (features) and the corresponding output (labels) by estimating the conditional probability $p(y|x)$. This means the model focuses on distinguishing or "discriminating" between different classes or outcomes based on the input.
    - **Example Applications**: Image recognition, text classification, speech recognition.
- **Generative Models:** Generative models, on the other hand, attempt to model the entire distribution of the data, by estimating the joint probability $p(x, y)$, which means they learn how the inputs and outputs are related as well as the underlying patterns in the data itself. After the model has been trained, it is possible to infer the conditional probability, that is, generate new data characterized by a similar distribution.
    - **Example Applications**: Deepfake generation, music composition, and text synthesis.

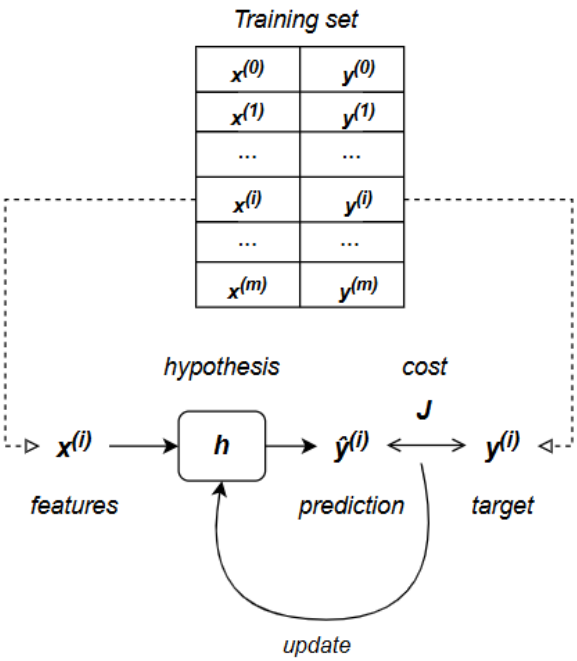
General notation

Let's establish the notation we'll use to maintain consistency:

- $m =$ number of training examples
- n = number of features
- x = "input" variable/ feature
- y = "output" variable / "target" variable
- (x, y) = tuple rappresenting one training example
- $\left(x^{(i)}, y^{(i)}\right)$ = training example i^{th}
- **$x_j^{(i)}$ = the j^{th} feature of the training example i^{th}**

Supervised Learning

In supervised learning, the algorithm learns from a **labeled** training dataset $D = \left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{m}$ where both the input data $x^{(i)}$ and the corresponding correct outputs (labels) $y^{(i)}$ are provided. Usually, $x^{(i)}$ is a vector where each component j ($x_j^{(i)}$) is called **feature**, **attribute,** or **covariate**, and $y^{(i)}$ is the desired output, also known as **target**.



The algorithm is defined by a **hypothesis function $h(x)$** which, given the input data $x^{(i)}$ performs an estimation (prediction) of the target value associated with that input data. To evaluate the quality of these predictions, we use a performance metric in the form of a **cost function** (also known as **loss function**), which measures the error between the predicted value and the actual target value.

To improve the learning algorithm, we "**update**" the hypothesis based on this error, aiming to reduce it in the next iteration. This process is repeated for all m instances $x^{(i)}$ of our training set. Over time, the model should refine itself, hopefully resulting in an algorithm (hypothesis) that can provide estimates (predictions) close to the target values. The algorithm resulting from this learning process is generally called **model**.

The Figure on the side illustrates this process (don't worry, we will explore all of these steps in detail shortly).

This is the foundation of most supervised learning techniques (though not always), where what generally changes is the learning algorithm (defined by its hypothesis function) and the cost function.

To better illustrate this, let's begin by analyzing one of the simplest (and most well-known) hypothesis functions we can construct: **linear regression**.

1 Linear Regression

Linear regression is a statistical model that estimates the **linear** relationship between a **dependent variable** and one or more **independent variables** (also called **regressors**). In our case the independent variables will be the features and the dependent variable will be the target.

A model with exactly one feature is a **simple linear regression** and its hypothesis function is defined as:

$$h_\theta\left(x^{(i)}\right) = \theta_0 + \theta_1 x_1^{(i)}$$

While a model with two or more features is a **multiple linear regression** and its hypothesis function is defined as:

$$h_\theta\left(x^{(i)}\right) = \theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)}$$

where n is the number of features in the training set.

Here, $\theta_0, \theta_1, \ldots, \theta_n$ are the **parameters** (also called **weights**) that define the linear relationship between the input x and the output y.

When correctly trained (i.e., when the optimal parameter values are determined), linear regression allows us to predict the target value based on input features.

A classic example is the Housing price prediction in Ames (Iowa) where the goal is to estimate the price of a house given characteristics such as living area, the number of bedrooms, the number of floors, etc. To achieve this, we "fit" the model, finding the parameters θ that best approximate the relationship between the target (price) and the features (house characteristics).

1.1 Simple Linear Regression

For the moment let's concentrate just on the simple linear regression, we will come back on the multiple linear regression later.

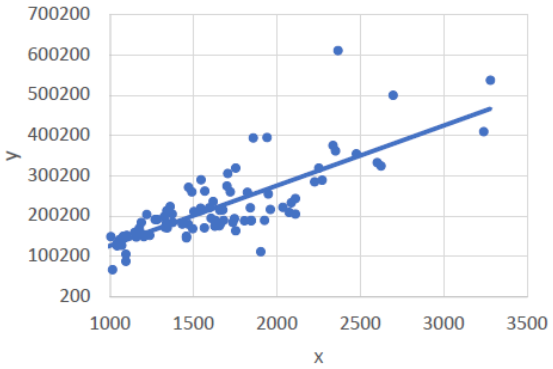
Living area (feet2)	Price (1000$s)
1656	215
896	105
1329	172
2110	244
...	...

Suppose we have a training set with one feature, Living Area, and a target, Price, as shown in the Figure above. In this case, the output of the hypothesis function is a **straight line**:

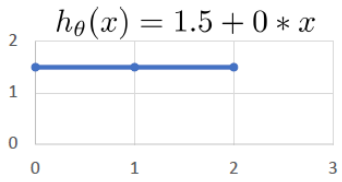
$$h_\theta\left(x^{(i)}\right) = \theta_0 + \theta_1 x_1^{(i)}$$

where θ_0 represents the intercept term and θ_1 represents the slope of the line.

There we can say that our objective is to find the straight line that best aligns with the target prices. In more precise terms, we aim to find the optimal values for θ_0 and θ_1 of our hypothesis function to actually be able to predict prices that are as close as possible to the target values.

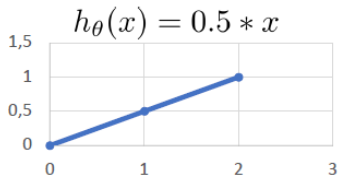


For example, by adjusting the parameters θ_0 and θ_1, we can create straight lines with different slopes and intercepts, leading to different fits:



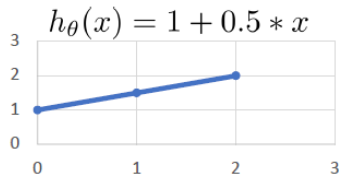
$\theta_0 = 1.5$

$\theta_1 = 0$



$\theta_0 = 0$

$\theta_1 = 0.5$



$\theta_0 = 1$

$\theta_1 = 0.5$

Note: One key aspect of regression (whether linear or other forms like polynomial regression) is that the output is a **continuous** value. This distinguishes regression from classification, where the output is typically a discrete category.

Linear Regression Cost Function

How do we choose the values of our parameters? → We can choose the **parameters** so the **hypothesis is** as **close** as possible **to y for** our **training examples**. Okay said that, now we need a way to assess how close our predictions are to the actual target values. We can therefore use a **cost function** to quantify the error between the predicted values and the target values in our training data.

One commonly used cost function is the **Mean Squared Error (MSE)**, which measures the average of the squared differences between the predicted values and the actual target values. It is defined as:

$$MSE = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

Here, m is the number of training examples in our training set, $h_\theta\left(x^{(i)}\right)$ is the output of the hypothesis function (therefore the predicted value), and $y^{(i)}$ is the actual target value.

Sometimes, for mathematical convenience, a factor of $1/2$ is included in the MSE loss function, resulting in:

$$\boldsymbol{J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2}$$

This does not alter the optimization results but simplifies the derivative calculations.

Our goal will be to find the parameters that give us the lower error as possible, so the ones that minimize the cost function:

$$\theta_{min} = argmin_\theta\left(J(\theta_0, \theta_1)\right) = argmin_\theta(J(\theta))$$

This process is known as **minimizing the cost function**.

+ A related concept is **Ordinary Least Squares (OLS)**, which minimizes the **Sum of Squared Residuals (SSR)**, defined as:

$$\boldsymbol{SSR = \frac{1}{2}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2}$$

where also here, for mathematical convenience, a factor of $1/2$ is included.

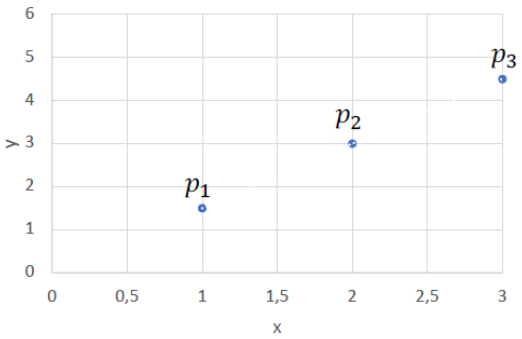
Therefore, OLS is defined as: $argmin_\theta(SSR)$

The MSE is essentially a scaled version of this objective function, with the scaling factor being $1/m$. Mathematically:

$$MSE = \frac{1}{m} SSR$$

Since $1/m$ is simply a constant for a given dataset, **minimizing the SSR in OLS also minimizes the MSE**. Both approaches yield the same optimal parameters θ_{min}. I wanted to underline this aspect because as we will see sometimes, we will operate directly with OLS approach rather than minimizing the MSE (but anyway we end-up with the same optimal parameters). For the moment let's assume to use the MSE.

The hypothesis function parameters influence the cost function, i.e. by changing their values the error can increase or decrease. To see how the parameter values influence the cost function, consider the following example:



Example: Let's say we have three data points:

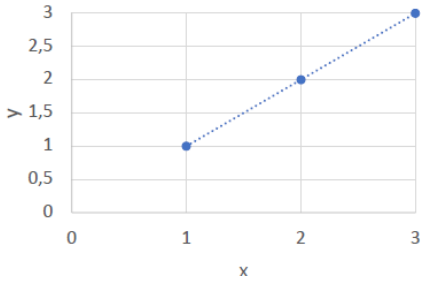
$$p_1 = (1,1)$$
$$p_2 = (2,2)$$
$$p_3 = (3,3)$$

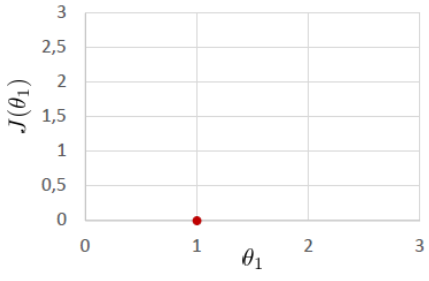
and we adopt a linear function $h_\theta(x) = \theta_0 + \theta_1 x$. The MSE in this case will be given by:

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 = \frac{1}{2*3}[(\theta_0 + 1\cdot\theta_1 - 1)^2 + (\theta_0 + 2\cdot\theta_1 - 2)^2 + (\theta_0 + 3\cdot\theta_1 - 2)^2]$$

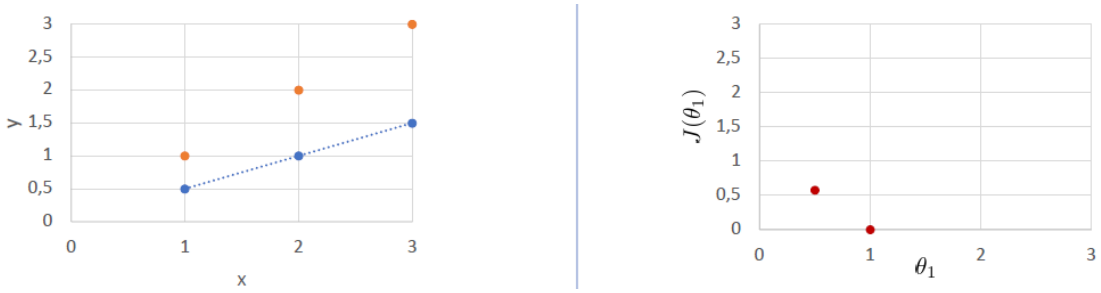
For simply calculation let's consider $h(x) = \theta_1 x$, so basically here we supposed $\theta_0 = 0$. Given so, our MSE now will be $J(\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(\theta_1 x^{(i)} - y^{(i)}\right)^2$.

- If $\theta_1 = 1$, then if we compute the MSE we obtain $J(1) = \frac{1}{2*3}[0^2 + 0^2 + 0^2] = 0$. In this case the estimate is perfect, the error is zero, consequently the cost function takes on a zero value.

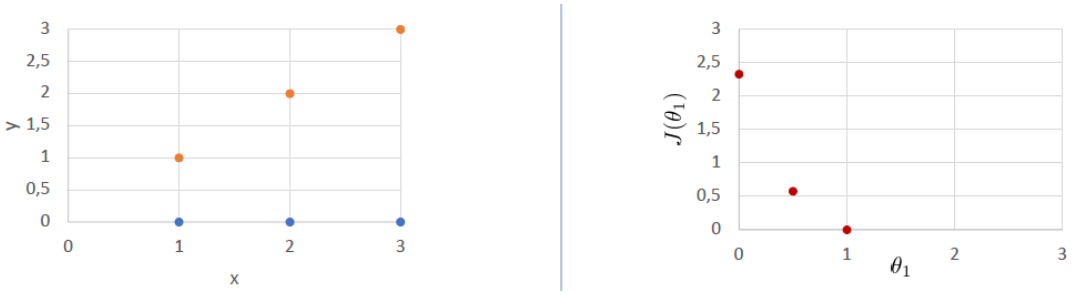




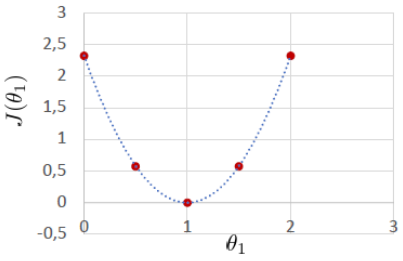
- If $\theta_1 = 0.5$, then MSE will be $J(0.5) = \frac{1}{2*3}[(0.5-1)^2 + (1-2)^2 + (1.5-3)^2] = \frac{3.5}{6} = 0.58$. We can see that by changing the value of the parameter θ_1, the error is no longer negligible, consequently the cost function takes on a non-zero value. More specifically here, the predictions are less accurate, leading to a higher error.



- If $\theta_1 = 0$, then MSE will be $J(0) = \frac{1}{2*3}[(0-1)^2 + (0-2)^2 + (0-3)^2] = \frac{14}{6} = 2.33$. This is the worst case, where the predictions are far from the target values, resulting in the highest error.

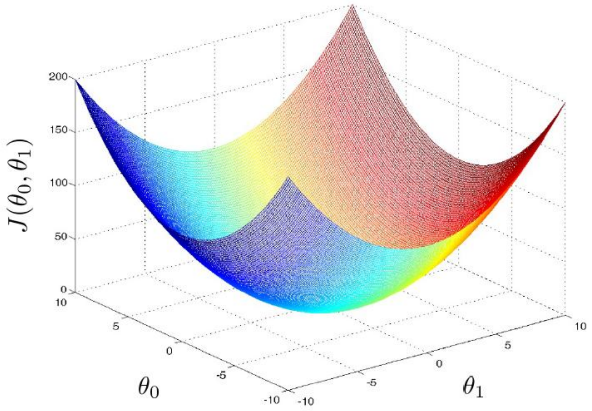


From the example you should also notice that, in this case, the cost function obtained is a parabola, i.e. a **quadratic function** of θ_1 (see Figure). Since quadratic functions are convex, we are sure that the cost function has a single global minimum and it's there that the parameter θ_1 is optimized (i.e. the predictions are as accurate as possible).

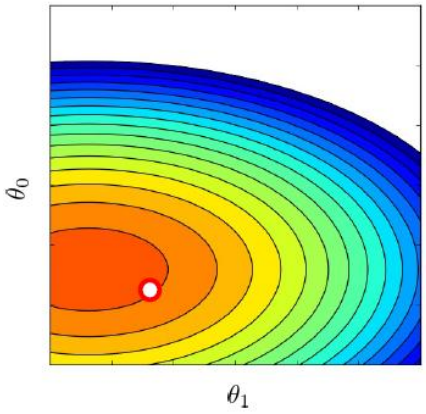


Note: However, this is not always the case as if the convexity doesn't hold (e.g., for more complex data distributions), multiple local minima may exist, and in that case **local mimimum≠global mimum**.

In the above example, we considered a single parameter (θ_1), resulting in a 2D plot of the cost function (a parabola). If we consider both θ_0 and θ_1, the cost function forms a **3D bowl-shaped curve**, as shown in the Figure below.



Additionally, the cost function can be visualized using a **contour plot**, which represents slices of the cost function in 2D. Contour plots show curves where the cost function has the same value, helping us visualize how the parameters affect the cost (see Figure below).



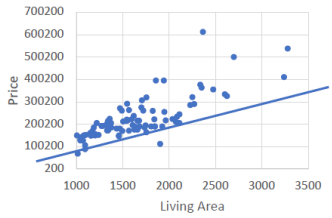
Now, let's return to our primary goal: **minimizing the cost function**, specifically the Mean Squared Error (MSE). At this point we may ask: "how we can minimize it?"

A Naïve Approach: One simple approach could be to **randomly** choose values for θ, plug them into the hypothesis function, make predictions, compute the MSE, and **hope** to find the parameters that yield the lowest error.

The Figures below illustrate how choosing different values for θ_0 and θ_1 results in different MSE values, corresponding to different positions on the cost function's contour plot (which represents levels of error).

$h_\theta(x)$

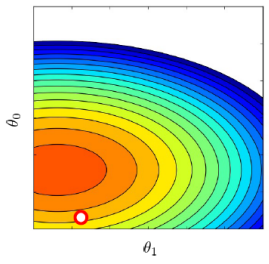
(function of x, with θ_1 fixed)



$$\theta_0 = 100000$$
$$\theta_1 = 66$$
$$h_\theta(x^{(i)}) = 100000 + 66x$$

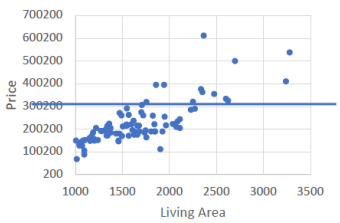
$J(\theta_0, \theta_1)$

(function of θ_0, θ_1, an aggregate function over all x's)



$h_\theta(x)$

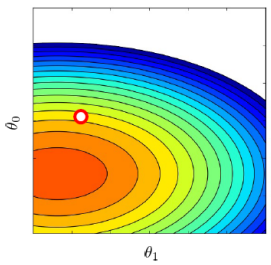
(function of x, with θ_1 fixed)



$$\theta_0 = 300200$$
$$\theta_1 = 0$$
$$h_\theta(x^{(i)}) = 300200 + 0x$$

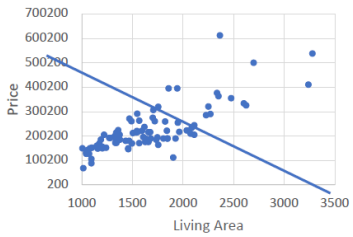
$J(\theta_0, \theta_1)$

(function of θ_0, θ_1, an aggregate function over all x's)



$h_\theta(x)$

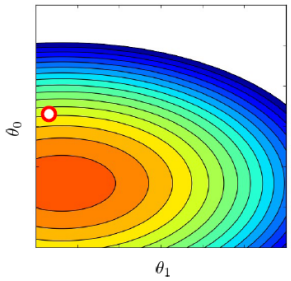
(function of x, with θ_1 fixed)



$$\theta_0 = 600000$$
$$\theta_1 = -90$$
$$h_\theta(x^{(i)}) = 600000 - 90x$$

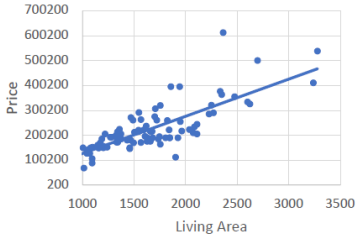
$J(\theta_0, \theta_1)$

(function of θ_0, θ_1, an aggregate function over all x's)



$h_\theta(x)$

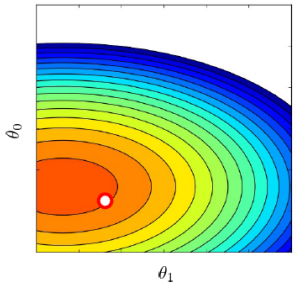
(function of x, with θ_1 fixed)



$$\theta_0 = 200000$$
$$\theta_1 = 70$$
$$h_\theta(x^{(i)}) = 200000 + 70x$$

$J(\theta_0, \theta_1)$

(function of θ_0, θ_1, an aggregate function over all x's)



By tuning these parameters, we can navigate through the surface of the cost function and move closer to the minimum point where the error is the smallest.

A Better Approach: At this point we may ask: "Is there a better approach to reach this minimum rather than simply tries values at random?". The answer is yes, and it's by using an efficient optimization algorithm called **gradient descent**.

1.2 Gradient Descent

Before diving into the details of gradient descent, it's important to note that while linear regression with one feature is simpler, many real-world problems involve multiple features. We can generalize the cost function $J(\theta_0, \theta_1)$ for one variable to handle multiple features:

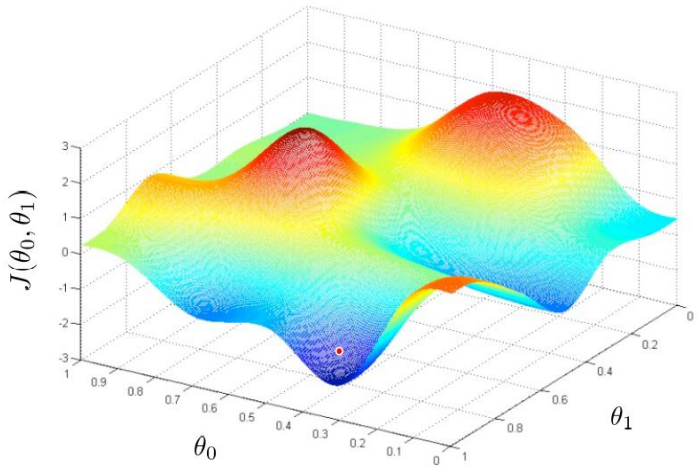
$J(\theta_0, \theta_1)$	→	$J(\theta_0, \theta_1, \dots, \theta_n)$
Goal: $\min_{\theta_0, \theta_1}(J(\theta_0, \theta_1))$		$\min_{\theta_0, \theta_1, \dots, \theta_n}(J(\theta_0, \theta_1, \dots, \theta_n))$

Our goal remains the same: to minimize the cost function. For simplicity, however, we will focus on the case of two parameters θ_0 and θ_1 to explain the concepts, as the approach applies similarly when there are n features.

Said that our goal is to:

1. Initialize the parameters θ_0, θ_1 with some values.
2. Keep changing θ_0, θ_1 to reduce to get closer $J(\theta_0, \theta_1)$ and reach the minimum.

To achieve this, we need a method to move along the cost function's contour (which represents error) towards the minimum.

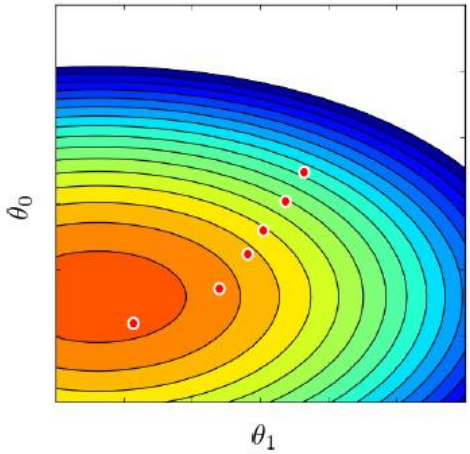


This is where Gradient Descent comes in. **Gradient Descent (GD)** is an optimization technique that helps us **iteratively** adjust the parameters to find the minimum of a function, in our case will be the cost function. It does so by following the **direction of the *steepest descent*** (where the cost decreases most rapidly) which is determined by the **gradient** $\nabla J(\theta)$ (the vector of partial derivatives) of the cost function (which points us towards the nearest minimum):

$$\nabla J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_j} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix}$$

The idea is simple: at each step, we adjust the parameters slightly in the direction indicated by the gradient (this means computing the derivative of the cost function with respect to each parameter (partial derivatives)), gradually moving closer to the minimum.

By following the gradient, we efficiently converge in a more direct way towards the minimum of the cost function (see Figure) without wasting time exploring irrelevant regions of the parameter space (as were doing in the naïve approach).



Gradient Descent Algorithm

The steps to perform gradient descent are as follows:

1. **Initialize the Parameters**: We randomly initialize the values of θ_0, θ_1. In cases where the cost function is convex (bowl-shaped), any starting point will eventually converge to the **global minimum**. If the function is **not convex**, choosing the starting point becomes crucial because different starting points may lead to different minimums (local minima).

2. **Update the Parameters**: Each parameter is updated based on the partial derivative of the cost function w.r.t. to that parameter:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \mid \alpha > 0$$

where α is the **learning rate**, a hyperparameter that controls the step size taken in each iteration towards the minimum.

The process is repeated iteratively until convergence.

Note: **Hyperparameter** = a parameter that is not calculated but set manually.

Important:

- All parameters θ_j are **updated simultaneously** at each step to ensure a coordinated descent. This means that before changing the values of θ_j through this formula, we must first calculate the new values for all parameters using the update rule, then update them simultaneously.

Correct: Simultaneous update

$$\text{Temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$
$$\text{Temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$
$$\theta_0 := \text{temp0}$$
$$\theta_1 := \text{temp1}$$

Incorrect:

$$\text{Temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$
$$\theta_0 := \text{temp0}$$
$$\text{Temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$
$$\theta_1 := \text{temp1}$$

- Or without needing to keep in memory a temporary value for the new θs, what we can do is first compute all the partial derivates and then once computed all we can proceed with the updates:
  - First we compute:

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

  - and then we perform the updates:

$$\theta_0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

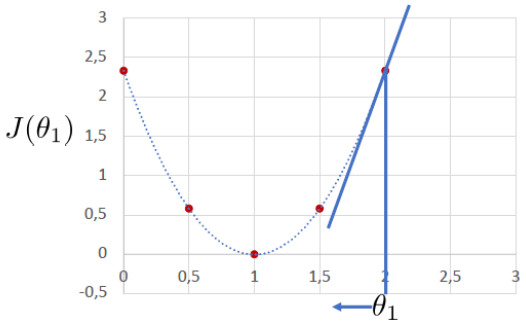
Note: This is the same as the one seen above, the important thing to understand is that we need to perform update only after have computed the new parameters value for all of them (I wanted to underline this approach since in practical exercises this is easier to do).

Gradient Descent Intuition

The derivative $\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ gives us the slope of the cost function at the current point:

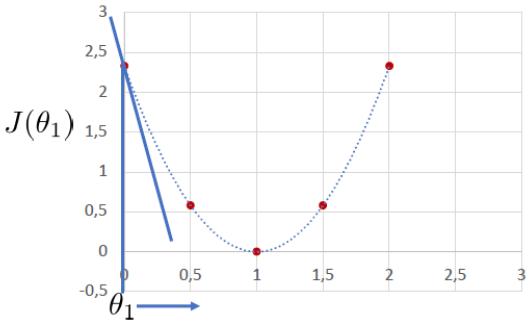
- If the derivative is **positive**, it means the cost function is increasing, so we need to move in the **opposite direction** by reducing θ_j.
- If the derivative is **negative**, it means the cost function is decreasing, so we should increase θ_j.

In essence, the gradient points in the direction where the cost function increases the fastest. So, we need to move in the **opposite direction** to head toward the minimum, that's why we have the **negative sign** in the formula. We can easily constate it by looking at the example in Figure where we consider the cost function for a parameter θ_1:



$$\theta_1 := \theta_1 - \alpha \underbrace{\frac{\partial}{\partial \theta_1} J(\theta_1)}_{\geq 0}$$

$$\theta_1 := \theta_1 - \alpha * (positive\ number) \rightarrow we\ decrease\ \theta_1$$

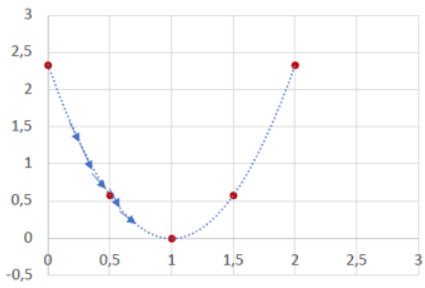


$$\theta_1 := \theta_1 - \alpha \underbrace{\frac{\partial}{\partial \theta_1} J(\theta_1)}_{\leq 0}$$

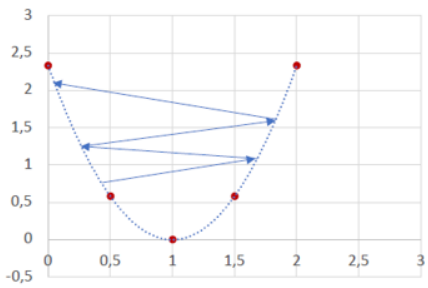
$$\theta_1 := \theta_1 - \alpha * (negative\ number) \rightarrow we\ increse\ \theta_1$$

The learning rate α controls how large each step is:

- If α is too **small**, gradient descent will be **slow** to converge, taking many iterations to reach the minimum.
- If α is too **large**, gradient descent may **diverge**, overshooting the minimum.

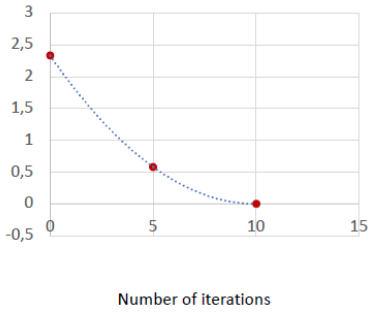


α too **small**

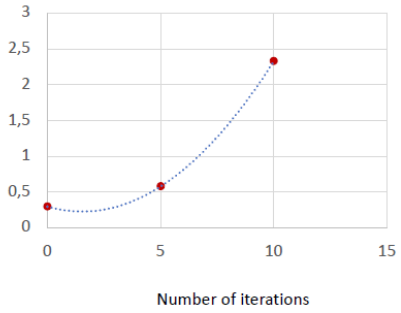


α too **large**

Finding the right learning rate is crucial for efficient convergence. Bad values of α can lead to oscillations, divergence or even bouncing (like a sinusoidal function).

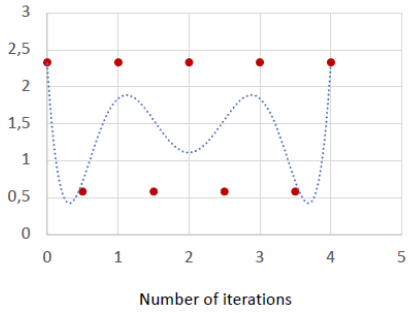


Gradient descent is working



Gradient descent is **NOT** working

Use smaller alpha



Gradient descent is **NOT** working

use smaller alpha

A last consideration to make is that in theory, gradient descent converges to the minimum after an **infinite number of iterations**. In practice, we define a **stopping criterion** to decide when to terminate the algorithm. A common stopping criterion is the use of a **fixed number of iterations** after which we stop the algorithm. Later we will see also others.

1.3 Multiple Linear Regression

As we said, we often deal with multiple features($n \geq 1$). This will be for example the case of Housing price prediction that we saw before when start to consider more features such as living area, the number of bedrooms and the number of floors. In this case our hypothesis function will change from a simple linear regression to a multiple linear regression (in order to handle these additional features):

Living area ($feet^2$)	Number of bedrooms	Number of floors	Price (1000$)
1656	5	1	215
896	3	2	105
1329	4	2	172
2110	2	1	244
…	…	…	…

$Before$: $h_\theta(x^{(i)}) = \theta_0 + \theta_1 x_1^{(i)}$

$After$: $h_\theta(x^{(i)}) = \theta_0 + \theta_1 x_1^{(i)} + \theta_1 x_2^{(i)} + \theta_1 x_3^{(i)}$

In this case our goal will always be to predict the price (target), but in this case we will do it considering all these features.

Considering the general case of have n features, then the multiple linear regression can be defined as:

$$h_\theta(x^{(i)}) = \theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)}$$

If for convenience we set $x_0^{(i)} = 1$ for every training sample, $i = 1, \ldots, m$ then we can compact the formula even more:

$$h_\theta(x^{(i)}) = \theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)} = \sum_{j=0}^{n} \theta_j x_j^{(i)}$$

In matrix form, we can express the feature vector $x^{(i)}$ and parameter vector θ as:

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} = 1 \\ x_1^{(i)} \\ \dots \\ x_n^{(i)} \end{bmatrix} \in \mathbb{R}^{n+1}, \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \dots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

Thus, the hypothesis $h_\theta(x^{(i)})$ is expressed as a matrix product:

$$h_\theta(x^{(i)}) = [\theta_0 \quad \theta_1 \quad \dots \quad \theta_n] \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \dots \\ x_n^{(i)} \end{bmatrix} = \boldsymbol{\theta^T x^{(i)}}$$

Dimensionality: It's important to have a look at the dimensionality: $\theta^T \in \mathbb{R}^{1\times(n+1)}$ and $x^{(i)} \in \mathbb{R}^{(n+1)\times 1}$, so $h_\theta(x^{(i)}) \in \mathbb{R}^{1\times 1}$ for the matrix's product rule. So, we end up with a result that is $\mathbb{R}^{1\times 1}$, so a **single** real number.

! Note: Basically, above we assumed that each training sample $x^{(i)}$ was represented as a column vector of n features, and so following this convention the same has been for y and θ. However, we may have used different assumptions for representation. For instance:

- We may have represented $x^{(i)}$, y and θ as a row vector, and according to this we will end up with $h_\theta(x^{(i)}) = \boldsymbol{\theta(x^{(i)})^T}$.
- Or moreover we may have represented $x^{(i)}$ as row vector of n column features, while y and θ as column vectors, and in this case we end up with $h_\theta(x^{(i)}) = \boldsymbol{x^{(i)}\theta}$.

I prefer the last one since it generally follows the structure of how a dataset is organized where the training sample are the rows while the columns are the features. However, a lot of texts use the first assumption, so I will try to follow that one, but sometimes we will need to change this notation, so be careful about it (it's not my fault if each one uses a different assumption 😟)!

→ When working with vectors or matrices it is important to have a look at their **dimensionality**!

! Note: Moreover, don't confuse multiple linear regression with **multivariate linear regression**:

- multiple linear regression predicts a single dependent variable using more than two dependent variables.
- multivariate linear regression which predicts **multiple dependent variables** (responses) simultaneously, not just one.

1.4 Deriving Explicit form for Partial Derivatives of the MSE cost function w.r.t to parameters in Linear Regression

1. Simple linear regression (one feature)

For simple linear regression, the hypothesis function is defined as:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

and the cost function is given:

$$J(\theta_0, \theta_1) = J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

Now, let's derive an explicit form for the partial derivatives of the cost function (MSE) w.r.t to each parameter, required for updating the parameters in GD.

Since we need the partial derivatives of the cost function with respect to θ_0 and θ_1, we differentiate as follows:

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} = \frac{\partial}{\partial \theta_j}\left(\frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2\right)$$

Thanks to the linearity of summation, we can differentiate the expression inside the summation:

$$\frac{\partial}{\partial \theta_j}\left(\frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2\right) = \frac{1}{2m}\sum_{i=1}^{m}\frac{\partial}{\partial \theta_j}\left(\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2\right) =$$

$$= \frac{1}{\cancel{2}m}\sum_{i=1}^{m}\cancel{2}\left(h_\theta(x^{(i)}) - y^{(i)}\right)\frac{\partial h_\theta(x^{(i)})}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)\frac{\partial h_\theta(x^{(i)})}{\partial \theta_j}$$

Here you can see that the factor $1/2$ come in help simplifying the derivative.

Next, we handle the cases for $j = 0$ and $j = 1$:

For $j = 0$, we compute the derivative of the hypothesis with respect to θ_0:

$$j = 0 \rightarrow \frac{\partial h_\theta(x^{(i)})}{\partial \theta_0} = \frac{\partial(\theta_0 + \theta_1 x^{(i)})}{\partial \theta_0} = 1$$

thus, the partial derivate for θ_0 becomes:

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)1$$

For $j = 1$, we compute the derivative of the hypothesis with respect to θ_1:

$$j = 1 \rightarrow \frac{\partial h_\theta(x^{(i)})}{\partial \theta_1} = \frac{\partial(\theta_0 + \theta_1 x^{(i)})}{\partial \theta_1} = x^{(i)}$$

thus, the partial derivative for θ_1 becomes:

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x^{(i)}$$

2. Multiple linear regression (multiple features)

In the case of multiple linear regression, the hypothesis function is defined as:

$$h_\theta(x^{(i)}) = \theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)}$$

where $x_0^{(i)} = 1$. Then for each $j = 0, 1, \ldots, n$ we will obtain the partial derivatives as follows:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{\partial J(\theta_0, \theta_1, \ldots, \theta_n)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)\frac{\partial h_\theta(x^{(i)})}{\partial \theta_j}$$

where:

$$\frac{\partial h_\theta(x^{(i)})}{\partial \theta_j} = \frac{\partial\left(\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \cdots + \theta_j x_j^{(i)} + \cdots + \theta_n x_n^{(i)}\right)}{\partial \theta_j} = x_j^{(i)}$$

Note: If we follow the convention for which we set $x_0^{(i)} = 1$ also for the simple linear regression, then for θ_0 we will end up with:

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \boldsymbol{x_0^{(i)}} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) 1$$

→ Thus, in general in linear regression (for both simple and multiple linear regression) the explicit form of a partial derivative of the MSE cost function w.r.t to **any $\boldsymbol{\theta_j}$** becomes:

$$\boldsymbol{\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}}$$

and you can then use it within the GD update.

1.5 Batch GD vs Stochastic GD vs Mini Batch GD

What we have done so far was use **all** training samples in the dataset to calculate the gradient and perform a single update to the parameters. This technique is known as **Batch Gradient Descent (BGD)** and the updates are done such that:

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

However, since datasets often contain a very large number of training samples (m) and relatively fewer features (n), processing all samples before every update can be computationally expensive and time-consuming. The computational cost of BGD per iteration is $O(m)$, which grows linearly with the number of samples. Therefore, in such cases adopting BGD would take too long since we have to process all the data before performing an update.

To address the inefficiency of BGD, an alternative method called **Stochastic Gradient Descent (SGD)** is used. In SGD, a **single** randomly chosen training sample is used to compute the gradient and update the parameters:

$$\theta_j := \theta_j - \alpha\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

Unlike BGD, where the gradient is computed using all m samples, SGD updates the parameters based on a single sample that is selected randomly ***with replacement*** from the dataset at each iteration. To ensure this provides an unbiased estimate of the gradient, the sample is drawn **uniformly** at random (from a uniform distribution), giving each training example an equal probability of $1/m$ of being selected.

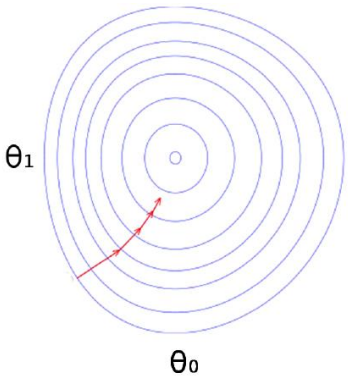
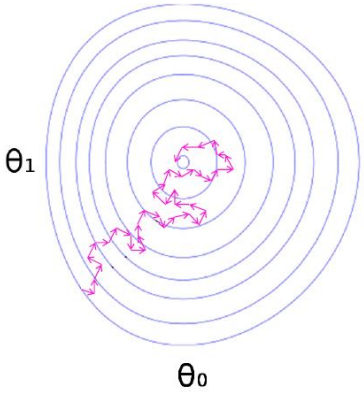
It can be demonstrated that the gradient computed in this way (i.e. considering a single sample at each iteration) is a **stochastic approximation** of the full gradient:

$$\mathbb{E}\left[\nabla J\left(\theta|x^{(i)}\right)\right] = \frac{1}{m}\sum_{i=1}^{m}\nabla J\left(\theta|x^{(i)}\right) = \nabla J(\theta|X)$$

This means that, **on average**, the stochastic gradient is a good estimate of the gradient.

SGD significantly reduces the computational cost per iteration from $O(m)$ to $O(1)$, making SGD much more efficient for large datasets. However, it introduces noise into the optimization process because the gradient is calculated using only one sample. This noise can cause the algorithm to oscillate around the minimum (a "**zig-zag**" effect), potentially preventing it from settling exactly at the global minimum.

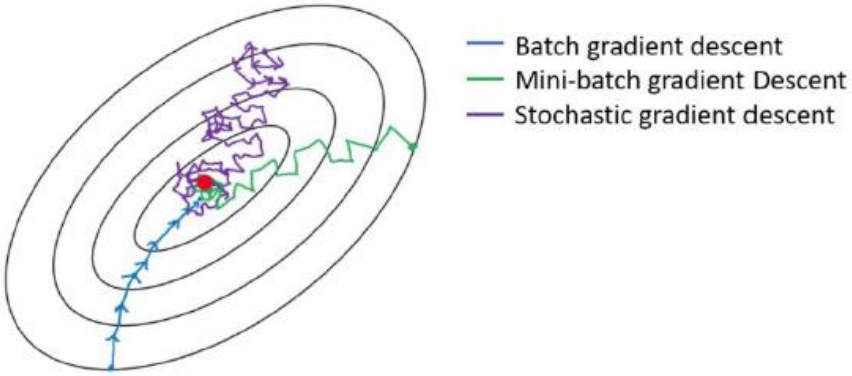
In practice, SGD is often preferred for large datasets as it quickly finds a solution close to optimal, even though it may not precisely reach the minimum.

Batch GD	Stochastic GD
Each "jump" is directed towards the minimum, but the relative update step goes through all training examples. For this reason, it is slower, but convergence is assured. 	Each "jump" may potentially go anywhere but is relative to only one training sample. For this reason, it is faster, but convergence is not sure. 

To achieve a balance between the stability of BGD and the efficiency of SGD, we use **Mini-Batch Gradient Descent (MBGD)**. Instead of computing the gradient using all the training samples (as in BGD) or a single sample (as in SGD), MBGD uses a **small batch of b randomly chosen training samples** $(1 < b \ll m)$ to compute the gradient and update the parameters. In MBGD the parameters are updated as:

$$\theta_j := \theta_j - \alpha \frac{1}{b} \sum_{i=1}^{b} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)}$$

Similar to SGD, each small batch of b samples (called **mini-batch**) is selected randomly with replacement. Using mini-batches reduces the variance in the updates compared to SGD, resulting in more stable convergence, while also being computationally more efficient and faster than BGD. The cost per iteration for MBGD is $O(b)$, which depends on the **mini-batch size** b.



! In both SGD and MBGD, using random sampling with replacement can introduce two issues. First, the same sample may be selected multiple times during iterations, which may bias the gradient estimate. Second, some samples may never be selected during iterations, leading to under-utilization of the dataset. To mitigate these issues, a more common practice is to iterate through the entire training set without replacement. In the machine learning context, we refer to a complete pass over the training dataset without replacement as an **epoch**.

Therefore, in practical implementations, SGD is often performed by iterating through all training samples during each epoch, updating the parameters one sample at a time. Similarly, MBGD iterates over the training samples in mini-batches during each epoch, updating the parameters each batch at time. These approaches ensure all samples are utilized and are the most commonly used variations in practice, as they strike a balance between computational efficiency and stability in convergence.

Note: When using this last version of MBGD we can notice that when $b = 1$ this algorithm responds like an **SGD** and when $b = m$ like a **BGD**. Considering this, a lot of implementations realize a unique code for Gradient Descent leaving the m used to discriminate between these approaches.

Note: Another thing you should notice is that:

- In SGD we consider just a single example to compute the gradient.
- In BGD we are **averaging** $(1/m)$ the partial derivatives computed considering each training example to then update the overall gradient.
- In MBGD we are **averaging** $(1/b)$ the partial derivatives computed considering each batch of training example to then update the overall gradient.

BGD ***vs*** ***SGD***

```
theta = rand();
while (not convergence){
 # iteration over theta parameters
  for( j from 1 to  n){
    update = 0;
    # iteration over training samples
    for( i from 1 to  m)}
      update = update + (h(x[i]) - y[i]) x[i];
    }
    theta_new[j] = theta[j] – alpha * update / m;
  }
  theta = theta_new;
}
```

```
theta = rand();
while (not convergence){
 # iteration over training samples
  for(i from 1 to  m){
    # iteration over theta parameters
    for(j from 1 to  n){
      theta_new[j] = theta[j] – alpha * (h(x[i]) - y[i]) x[i];
    }
    theta = theta_new;
  }
}
```

Note: The "for" loops in BGD and in SGD are exchanged.

MBGD

```
b = 50;
theta = rand();
while (not convergence) {
    # iteration over training samples
    while (i < m) {
        # iteration over theta parameters
        for (j from 1 to n) {
            update = 0;
            # iteration over the b samples
            for (z from i to i  + b) {
                update = update + (h(x[z]) - y[z]) x[z];
            }
            theta_new[j] = theta[j] – alpha * update / b;
        }
        theta = theta_new;
        i = i + b;
    }
}
```

Stopping Conditions

Gradient descent algorithm **stopping conditions**:

- **Max iteration**: the algorithm stops after a fixed number of iterations.
- **Absolute tolerance**: the algorithm stops when a fixed value of the cost function is reached.
- **Relative tolerance**: the algorithm stops when the decreasing of the cost function w.r.t. the previous step is below a fixed rate.
- **Gradient Norm tolerance**: the algorithm stops when the norm of the gradient is lower than a fixed value.

1.6 Polynomial Regression

Adding Features

In linear regression, we typically start with a set of features that describe our input data. For instance, if we have a training sample represented as $x^{(i)}$, our prediction function can be defined as:

$$h_\theta(x^{(i)}) = \theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)}$$

However, nothing stops us from creating **new features** which are derived from the original ones, which can enhance our model's predictive power.

Example: House Price Prediction

Suppose we are trying to predict the price of a house, and our input features are defined as:

$$x^{(i)} = [front^{(i)} \quad side^{(i)}]$$

At first glance we may think to predict the price with our function:

$$h_\theta(x^{(i)}) = \theta_0 + \theta_1 front^{(i)} + \theta_2 side^{(i)}$$

However, individual dimensions like "front" and "side" might not provide sufficient information to predict the price effectively. A more informative feature could be the **area** of the house, which can be calculated as:

$$area^{(i)} = front^{(i)} * side^{(i)}$$

Using this area feature, we can redefine our hypothesis to:

$$h_\theta(x^{(i)}) = \theta_0 + \theta_1 area^{(i)}$$

By focusing on the area, we simplify our model, reducing the number of features while still capturing the essential information that contributes to the house's price.

Note: We could substitute (as we did above) the new feature with the previous features, but if we want, we could also add this new feature to our previous ones.

Generally speaking, for any input $x^{(i)}$, we can introduce a new feature defined as a transformation of the existing features:

$$f(x^{(i)}) = x_{n+(k+1)}^{(i)} \; where \; k \in N$$

This transformation enables us to define a new hypothesis function $h'_\theta(x)$ which incorporates the new feature.

Introduction to Polynomial Regression

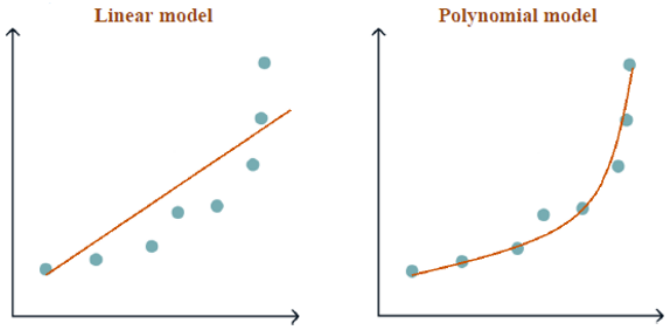
Polynomial regression extends the idea of linear regression by allowing for **non-linear** relationships between the input features and the target variable. It does this by introducing polynomial terms (that actually acts as non-linear transformations) of the existing features, which can help capture more complex patterns in the data.

For instance, consider the simple linear regression case, after having added new polynomial terms our hypothesis will be:

$$h_\theta(x) = \theta_0 x^0 + \theta_1 x^1 + \theta_2 x^2 + \cdots + \theta_t x^t$$

We refer to this as a polynomial regression model of degree $\boldsymbol{t}$.

Our goal remains consistent with that of linear regression—predicting the target value—but now we have additional higher-degree terms (e.g., x^2, x^3 and so on) that enables our model to capture non-linear relationships in the data that a simple linear regression can't handle (see Figure on side).



Considering our previous example the relationship between the side feature and the house price might not be linear, instead, it could be quadratic, cubic, or even higher-degree polynomial. By introducing polynomial terms of the area, we can capture these more complex patterns in the data.

For instance, if we adopt a **quadratic hypothesis function** we will have:

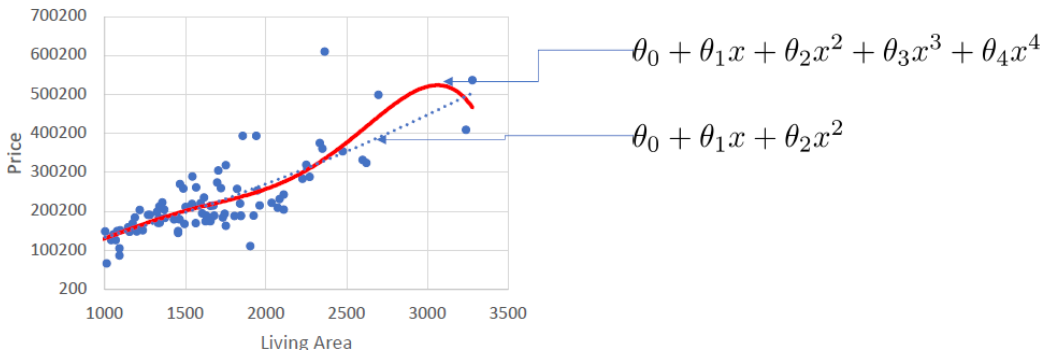
$$\theta_0 + \theta_1 x^1 + \theta_2 x^2 = \theta_0 + \theta_1 side + \theta_2 area^2$$

This type of function allows us to model a **U-shaped (quadratic)** relationship between area and price. If θ_2 is positive and significant, it suggests that larger houses increase in price at an accelerating rate (quadratic rate), better capturing market trends where larger homes might be disproportionately more expensive.

Similarly, we can extend this to **higher-degree polynomials** for even more flexibility in capturing complex relationships. For example:

$$\theta_0 + \theta_1 x^1 + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4 = \theta_0 + \theta_1 area + \theta_2 area^2 + \theta_3 area^3 + \theta_4 area^4$$

If θ_3 is negative and θ_4 is positive, this suggests that beyond a certain house area, prices might initially rise more slowly (due to market saturation) before increasing rapidly again for larger homes. This reflects scenarios where extremely large homes belong to niche segments of the market and might command significantly higher prices.



In general, by incorporating higher-degree polynomial terms, our model becomes more capable of fitting the data accurately, as we're no longer limited to assuming a strictly linear relationship. This can result in better predictive accuracy, especially when dealing with non-linear data.

However, there are important considerations when using polynomial regression:

1. **Overfitting**: Increasing the degree of the polynomial may lead to overfitting, where the model fits the training data perfectly but performs poorly on new, unseen data.
2. **Computational Complexity**: As the degree of the polynomial increases, the model becomes more complex, both in terms of computation and interpretation.

1.7 Feature Scaling and Normalization Techniques

Feature Scaling

When dealing with features that have vastly different ranges of values, it's important to **scale** them to comparable ranges. This avoids problems such as **zig-zagging** during gradient descent, where the algorithm struggles to converge because the features have different scales. The goal is to scale the features so that they fall within a similar range, improving computational efficiency and convergence.

Example:

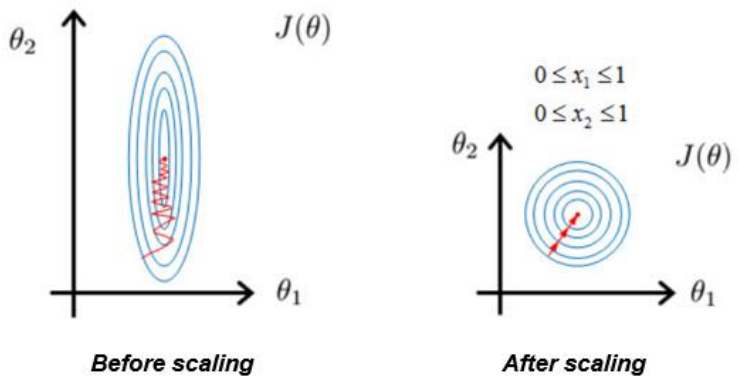
Consider a dataset where we have two features, the **area** of a house and the number of **bedrooms**, having the following ranges:

- $x_1 = area, x_1 \in [0,2110]$
- $x_2 = bedrooms, x_1 \in [1,5]$

As anticipated above, since these features differ significantly in range we have a zig-zag convergence problem (see Figure on the left). We need to scale them. We know the max area and the max number of bedrooms therefore we can scale these two factors as:

$$x_1 = \frac{area}{2110}, \qquad x_2 = \frac{bedrooms}{5}$$

By doing so we resized both our features values in a bounding region of $[0,1]$, letting them comparable (see Figure on the right).



Before scaling ***After scaling***

Normalization

Normalization is a type of feature scaling that adjusts the values of features to a standard range or distribution. Two common techniques for normalization are: **Min-Max Normalization** and **Z-Score Normalization**.

Min-Max Normalization

Min-max normalization scales features to a fixed range $[a, b]$:

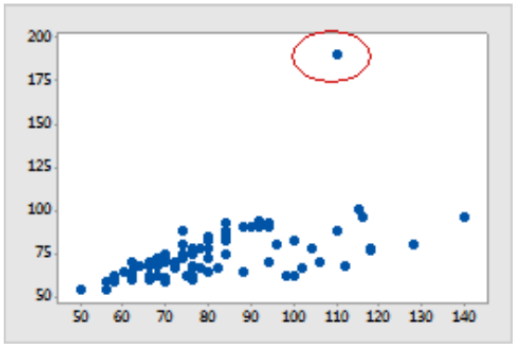
$$x' = \frac{x - min(x)}{max(x) - min(x)} \times (b - a) + a$$

This ensures that the transformed feature x'_j falls between a and b.

Basically:

- first applying $\frac{x-min(x)}{max(x)-min(x)}$ we **scale** x so that it falls between 0 and 1
- then multiplying it for $(b - a) + a$ we **shift** it to the desired range $[a, b]$.

Advantages: Min-max normalization preserves the relationships of the original data by just scaling and shifting the values to a desired (generally smaller) range.



Disadvantages:

- Min-max normalization is highly sensitive to **outliers**, which are extreme or isolated values in the dataset. Outliers can distort this normalization, as they may be treated as the minimum or maximum values, affecting all other normalized values.
- When new data points fall outside the original $[min, max]$ range (e.g., during the prediction phase), they will not conform to the normalized range leading to inconsistencies.

Z-Score Normalization (Standardization)

Z-score normalization, also known as **standardization**, transforms features based on their mean and standard deviation. The formula is:

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

where μ is the mean of x and σ is the standard deviation of x.

Z-score normalization ensures that the transformed features have a **mean of 0** and a **standard deviation of 1**. This method creates a standardized distribution (more compact), which can be beneficial for algorithms that assume normality in the data (e.g., linear regression, logistic regression).

Note: Here is like we scaled by σ and shifted by μ.

Advantages:

- Z-score normalization is less affected by outliers compared to min-max normalization.
- It also doesn't require defining a minimum and maximum value, making it more robust when new data points fall outside the original range (e.g. during the prediction phase).

Disadvantages: This method assumes that the data follows a normal distribution, which may not always be the case. If the data is highly skewed or has heavy tails, the Z-score normalization may produce misleading results, as it does not account for **skewness** or **kurtosis** (the shape of the distribution).

1.8 Normal Equations Method

In linear regression, we said that the goal is to find the optimal parameters θ that minimize the cost function $J(\theta)$. Previously, we introduced Gradient Descent, an iterative method to achieve this. However, there is an alternative method to minimize $J(\theta)$ explicitly without relying on iteration. This method, known as the **Normal Equations**, directly computes the optimal parameters by solving a system of equations derived in closed-form by setting the gradient of $J(\theta)$ to zero.

Note: We can see it as **a one-step learning algorithm** (as opposed to GD that requires more iterative steps).

It's important to note that the Normal Equations are specifically suited for solving the Ordinary Least Squares (OLS) problem in linear regression. However, as we already discussed previously, since the OLS method inherently minimizes the **Sum of Squared Residuals (SSR)** it aligns with minimizing the MSE, as MSE is nothing more than a scaled version of SSR (scaled by a factor of $1/m$).

Said that, following the third convention of representation (so here we change representation assumption), we can define:

$$X = \begin{bmatrix} x^{(1)T} \\ x^{(2)T} \\ \vdots \\ x^{(m)T} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}, \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}, \quad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

where $X \in \mathbb{R}^{m \times n}$ is the input matrix, $\theta \in \mathbb{R}^{n \times 1}$ is the parameters column vector and $y \in \mathbb{R}^{m \times 1}$ is the target column vector.

Note: Rember that we set all $x_0^{(i)} = 1$.

Under this convention of representation, the cost function $J(\theta)$ (i.e. the SSR) in matrix form becomes:

$$J(\theta) = \frac{1}{2}(X\theta - y)^T(X\theta - y)$$

It's trivial to show it. In fact, we can rewrite the difference vector $X\theta - y$ (also called residual vector) in explicit form as:

$$X\theta - y = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}$$

So, it follows that the dot product $(X\theta - y)^T(X\theta - y)$ can be written as:

$$(X\theta - y)^T(X\theta - y) = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} & h_\theta(x^{(2)}) - y^{(2)} & \dots & h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix} \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}$$

$$= \left(h_\theta(x^{(1)}) - y^{(1)}\right)^2 + \left(h_\theta(x^{(2)}) - y^{(2)}\right)^2 + \cdots + \left(h_\theta(x^{(m)}) - y^{(m)}\right)^2$$

which is just the original formulation of the SSR:

$$\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)^2=\left(h_\theta\left(x^{(1)}\right)-y^{(1)}\right)^2+\left(h_\theta\left(x^{(2)}\right)-y^{(2)}\right)^2+\cdots+\left(h_\theta\left(x^{(m)}\right)-y^{(m)}\right)^2$$

Therefore, we have constated that the cost function $J(\theta)$ in matrix form is given by:

$$J(\theta)=\frac{1}{2}(X\theta-y)^T(X\theta-y)$$

Now, we want to find the minimum of the cost function $J(\theta)$, i.e. the point in which the gradient of $J(\theta)$ has null value. Recalling that the gradient is the vector of partial derivatives of $J(\theta)$ with respect to each parameter θ_j, what we want to do is:

$$\nabla J(\theta)=\begin{bmatrix}\frac{\partial J(\theta)}{\partial\theta_0}\\ \vdots\\ \frac{\partial J(\theta)}{\partial\theta_j}\\ \vdots\\ \frac{\partial J(\theta)}{\partial\theta_n}\end{bmatrix}=0$$

Previously, we derived an explicit form for the partial derivatives of $J(\theta)$ when using the MSE. Specifically, each partial derivative was given by:

$$\frac{\partial J(\theta)}{\partial\theta_j}=\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)x_j^{(i)}$$

Since the MSE includes the scaling factor $1/m$, we drop this factor here to align with the formulation SSR. Therefore:

$$\frac{\partial J(\theta)}{\partial\theta_j}=\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)x_j^{(i)}$$

Now, re-writing explicitly this summation for all parameters θ_j where $j=0,..,n$, the gradient vector $\nabla J(\theta)$ becomes:

$$\nabla J(\theta)=\begin{bmatrix}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)\\ \sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)x_1^{(i)}\\ \vdots\\ \sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)x_n^{(i)}\end{bmatrix}=0$$

At this point, we can notice that we can re-express the same with the following dot product:

$$\nabla J(\theta) = \begin{bmatrix} 1 & 1 & \dots & 1 \\ x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ x_n^{(1)} & x_n^{(2)} & \dots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}$$

By performing the dot product, you can verify by yourself that this formulation is equivalent to the explicit summation above. Looking at what we got we can see that basically this is the dot product between the transpose of the input matrix X and the residual vector. Thus, the gradient becomes:

$$\boldsymbol{\nabla J(\theta) = X^T(X\theta - y)}$$

Note: You can find a better and more formal derivation in Andrew Ng notes, however it requires some additional knowledge of Linear Algebra.

Further expanding this:

$$= X^T X\theta - X^T y$$

Okay, we said that to minimize the cost function $J(\theta)$, we set the gradient to zero:

$$\nabla J(\theta) = X^T X\theta - X^T y = 0$$

Rearranging the terms, we get:

$$X^T X\theta - X^T y = 0 \quad \rightarrow \quad X^T X\theta = X^T y$$

Assuming $X^T X$ is invertible, we can solve for θ by multiplying left side both by $(X^T X)^{-1}$:

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

The result is the **Normal Equations** formulation given by:

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

This provides a closed-form solution for the optimal parameters $\hat{\theta}$ that minimize the cost function $J(\theta)$.

+ Considerations (extra)

Normal Equation method provides an elegant, closed-form solution and is a direct, one-step computation, unlike iterative methods such as GD. However, this approach presents several **drawbacks** that must be considered:

1. **Computational Complexity**: Computing the inverse of the matrix $X^T X$ has a time complexity of $O(n^3)$, where n is the number of features. This can become computationally expensive as the number of features increases.
2. **Matrix Inversion Required**: For the Normal Equation to work, $X^T X$ must be **invertible**. If the matrix is not invertible (also called singular), the equation will fail.

There are several reasons why $X^T X$ might not be invertible, but one of the most important that you should concern is the presence of **linearly dependent** features.

When two or more features are linearly dependent, it means that one feature can be expressed as a linear combination of the others. This results in $X^T X$ becoming singular because it lacks full rank. In such cases, the inverse does not exist, and the Normal Equation cannot solve for the parameters.

→ Let's better understand this. The matrix X having dimensions $m \times n$ where m is the number of data examples and n is the number of features, therefore $X^T X$ will be a matrix of dimensions $n \times n$. Now, if some columns of X are linearly dependent, the rank of X (i.e., the number of linearly independent columns) is less

than n. Consequently, X^TX, which depends on the columns of X, will also have rank less than n. By definition a matrix is invertible only if it is **full rank** (i.e., rank = number of columns = n). If X^TX is not full rank, it will be singular and therefore not invertible.

Moreover, this linear dependence introduces redundancy in the features, meaning that some variables are not adding new information but are instead combinations of other variables. Such redundancy can cause issues like **multicollinearity**, making it difficult to estimate unique parameters or solutions in regression problems.

Example: This issue is particularly common in polynomial regression because the newly added polynomial features are combinations of the original features. As a result, they can become highly correlated or nearly collinear.

To address this issue, a simple and effective approach is to use a **correlation matrix** to assess the degree of linear dependence among the features. By examining the pairwise correlation coefficients of the correlation matrix, you can identify features with high correlations (typically those with values greater than 0.8 or 0.9, which may indicate multicollinearity) and consider removing or modifying these redundant ones to mitigate the problem.

+ GD Update Rule in Matrix form: Previously, we derived an explicit form expression for the partial derivatives of the MSE cost function $J(\theta)$. Specifically, in case of BGD each partial derivative was given by:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

If we would like to express the gradient (i.e. the vector of all partial derivates) in matrix form (following the third convention of representation) we will obtain:

$$\boldsymbol{\nabla J(\theta) = \frac{1}{m} X^T(X\theta - y)}$$

This is the expression you will actually find in the code. Obviously, this also extends with the necessary modifications to the SGD and the MBGD.

1.9 Probabilistic Interpretation

We have taken it as a given that solving the OLS, or alternatively minimizing the MSE cost function, was the right thing to do to reduce error within the linear regression. But how can you be sure that the cost function adopted was a reasonable choice?

In this section, we will give a set of **probabilistic assumptions**, under which the least-squares cost function for minimizing linear regression is derived as a very natural algorithm.

Let us assume that the target variables and the inputs are related via the equation

$$y^{(i)} = \theta^T x^{(i)} + e^{(i)}$$

where $e^{(i)}$ is an error term that captures either unmodeled effects or random noise. Therefore, it is expressed as a **predicted value** (calculated by the hypothesis function $\theta^T x^{(i)}$) **plus** an **error**.

Let us further assume that the $e^{(i)}$ are independently and identically distributed (**i.i.d.**) according to a Gaussian distribution (also called a Normal distribution) with mean zero and some variance σ^2. We can write this assumption as $e^{(i)} \sim \mathcal{N}(0, \sigma^2)$, i.e. the density (pdf) of $e^{(i)}$ is given by:

$$p\left(e^{(i)}\right) = \mathcal{N}(0, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(e^{(i)}\right)^2}{2\sigma^2}\right) \qquad i = 1, \dots, m$$

At this point, we assume that the structure of the model (its hypothesis) and the parameters θ are fixed. Under this assumption, the probability of observing a specific output $y^{(i)}$, given an input $x^{(i)}$, depends entirely on the error between the predicted and actual values (the error $e^{(i)}$ captures any deviations or randomness in the relationship between $y^{(i)}$ and $x^{(i)}$). In other words, this implies that the probability that the input produces the output has the same probability of the error. Substituting

$$e^{(i)} = y^{(i)} - \theta^T x^{(i)}$$

in the expression above we get:

$$p\left(y^{(i)} | x^{(i)}; \theta\right) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right)$$

This represents the probability of observing the output value $y^{(i)}$ given the input feature $x^{(i)}$ and the fixed model parameters θ.

Note: You should note that $p\left(y^{(i)} | x^{(i)}; \theta\right)$ means that $y^{(i)}$ is conditioned only by $x^{(i)}$, and not by θ which is fixed.

Now, since our goal is always to maximize the probability that the predicted outputs are as close as possible to the actual outputs, it is necessary to find a function (of the θ parameters) that, given the input, returns the closest probable output for each training example. This particular function is called "**Likelihood**":

$$L(\theta) = L(\theta; X; y) = p(y | X; \theta)$$

More formally, the likelihood function $L(\theta)$ represents the probability of observing the output data y given the input data X and our actual parameters θ (fixed) of the model.

Moreover, note that $p(y | X; \theta)$ is a joint probability distribution.

Note that by the independence assumption on the $e^{(i)}$'s (and hence also the $y^{(i)}$'s given the $x^{(i)}$'s), this joint probability is given by the product of the individual probabilities for each data point:

$$L(\theta) = p(y | X; \theta) = \prod_{i=1}^{m} p\left(y^{(i)} | x^{(i)}; \theta\right) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right)$$

Now, given this probabilistic model relating the $y^{(i)}$'s and the $x^{(i)}$'s, what is a reasonable way of choosing our best guess of the parameters θ? The principal of **maximum likelihood estimation (MLE)** says that we should choose θ so as to make the data as high probability as possible. I.e., we should choose the parameters θ that maximize this likelihood $L(\theta)$:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\, L(\theta)$$

Instead of maximizing $L(\theta)$, we can also maximize any strictly increasing function of $L(\theta)$. In particular, the derivations will be a bit simpler if we instead maximize the **log likelihood $\boldsymbol{l}(\boldsymbol{\theta})$**:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \log L(\theta) = \underset{\theta}{\operatorname{argmax}}\, l(\theta)$$

where $l(\theta) = \log L(\theta)$ is the **log-likelihood**.

Therefore, now we aim to:

$$\boldsymbol{\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\, l(\theta)}$$

Logarithm operator introduced by the log-likelihood let us transform the production into a summatory:

$$l(\theta) = \log \prod_{i=1}^{m} p\left(y^{(i)} | x^{(i)}; \theta\right) = \sum_{i=1}^{m} \log p\left(y^{(i)} | x^{(i)}; \theta\right)$$

$$= \sum_{i=1}^{m} \log \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right)$$

Further simplifying:

$$= \sum_{i=1}^{m} \left(\log \frac{1}{\sqrt{2\pi}\sigma} + \log \exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right)\right) =$$

$$= \sum_{i=1}^{m} \left(\log \frac{1}{\sqrt{2\pi}\sigma} - \frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right) =$$

$$= m \log \frac{1}{\sqrt{2\pi}\sigma} - \sum_{i=1}^{m} \frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2} =$$

$$= m \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2} \sum_{i=1}^{m} \left(y^{(i)} - \theta^T x^{(i)}\right)^2$$

So, moving back to the maximization problem, we obtain:

$$\hat{\theta} = \max_{\theta} l(\theta) = \max_{\theta} \left(m \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2} \sum_{i=1}^{m} \left(y^{(i)} - \theta^T x^{(i)}\right)^2\right)$$

The term $m \log \frac{1}{\sqrt{2\pi}\sigma}$ can be ignored as it does not depend on θ:

$$= \max_{\theta} \left(-\frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^{m} \left(y^{(i)} - \theta^T x^{(i)}\right)^2\right)$$

and the same can be said for the term $1/\sigma^2$:

$$= \max_{\theta} \left(-\frac{1}{2} \sum_{i=1}^{m} \left(y^{(i)} - \theta^T x^{(i)} \right)^2 \right)$$

Ultimately, we move from a maximization problem to a minimization problem by changing the sign:

$$= \min_{\theta} \left(\frac{1}{2} \sum_{i=1}^{m} \left(y^{(i)} - \theta^T x^{(i)} \right)^2 \right)$$

Now, comparing this last formulation obtained with the one of the OLS:

$$\underbrace{\min_{\theta} \left(\frac{1}{2} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \right)}_{OLS} = \min_{\theta} \left(\frac{1}{2} \sum_{i=1}^{m} \left(y^{(i)} - \theta^T x^{(i)} \right)^2 \right)$$

and recalling that squaring the difference $(a - b)^2$ is the same as squaring the flipped difference $(b - a)^2$, we can say that these expressions are identical (or if we consider the MSE they are the same except for the scaling factor $1/m$ which as we said doesn't have influence on the minimization problem).

→ Thus, under the previous probabilistic assumptions on the data, we can say that **solving the OLS problem** (or alternatively minimizing the MSE) in linear regression is equivalent to **maximizing the log-likelihood**. This is thus one set of assumptions under which least-squares regression can be justified as a very natural method that's just doing maximum likelihood estimation (MLE).

2 Logistic Regression

2.1 Classification

Classification refers to the task of predicting a discrete category or class as the output, rather than a continuous value (as in regression). In classification, the goal is to assign each input to one of several predefined classes. This can involve:

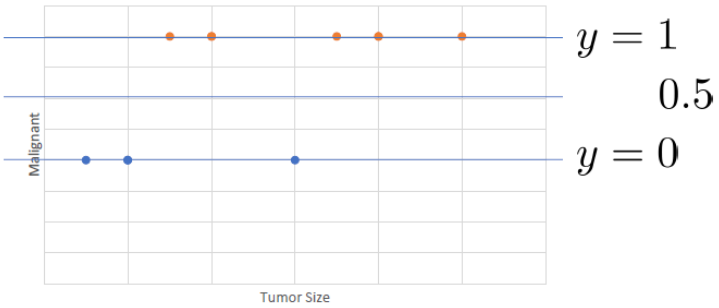
- **Binary classification**: where the output can belong to one of two possible classes, typically represented as $y = \{0,1\}$, with 1 often representing the positive class (e.g., malignant tumor) and 0 representing the negative class (e.g., benign tumor).
- **Multiclass classification**: where the output can belong to one of several classes, for example $y = \{0,1,2,3,4,5\}$, with each number representing a different class.

In general, for classification, we use a model that can classify new examples into these predefined classes. A naive method for doing this involves setting a threshold on the basis of which, depending on whether the values are below/above it, it is possible to divide them into different classes.

Example: Given input data x like tumor size, we want to predict whether it is benign or malignant (i.e., $y \in \{0,1\}$). We can set up a threshold equal to 0.5 where:

- If $h(x) \geq 0.5$, we predict class 1 (malignant tumor).
- If $h(x) \leq 0.5$, we predict class 0 (benign tumor).

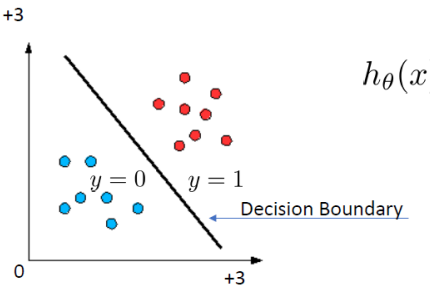
So here the threshold serves as the **decision boundary** between the two classes.



Decision Boundary Intuition: In general, we need a hypothesis function g that, given an input, is able to predict its class, assigning 0 to one side of the decision boundary and 1 to the other:

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

Is this simple enough? Not quite. We want a more reliable and rigorous approach that reflects the inherent uncertainty in classification and perhaps it is linked to a probabilistic formulation of the type:



$$h_\theta(x) = g(\underbrace{\theta_0}_{-3} + \underbrace{\theta_1}_{+1} x_1 + \underbrace{\theta_2}_{+1} x_2)$$

Predict $y = 1$ if $-3 + x_1 + x_2 \geq 0$

- The "closer" you are to 0 the greater the possibility of belonging to class 0
- The "closer" you are to 1 the greater the possibility of belonging to class 1

To this end, logistic regression is used.

2.2 Introduction to Logistic Regression

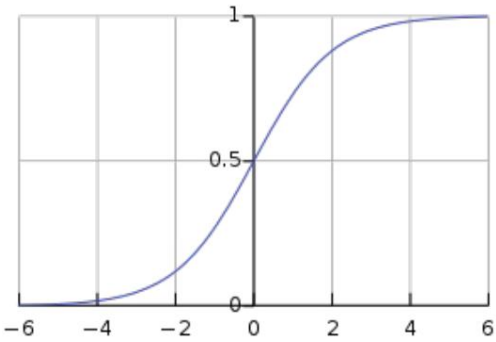
Logistic Regression is a model that aims to establish the probability with which an observation can generate one or another value of the dependent variable. This probability is then used to make a classification, which is why it is important not to be confused with the regression activity, even though the model includes the latter in the name (historical problem).

In Logistic Regression we want the output of the hypothesis function to be constrained within a certain range:

$$0 \leq h_\theta(x) \leq 1$$

To achieve this, the **logistic** (or **sigmoid**) function is used as hypothesis function. This function is defined as:

$$\boldsymbol{g(z) = \frac{1}{1 + e^{-z}}}$$



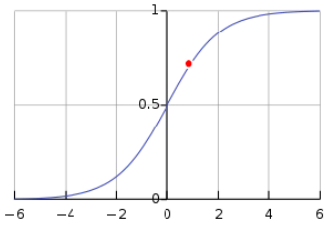
So, in the end, the hypothesis function becomes:

$$\boldsymbol{h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}}$$

The output of this function is always between 0 and 1, where the minimum value approaches 0 as z approaches $-\infty$, and the maximum value approaches 1 as z approaches $+\infty$. This ensures that every input produces a bounded output, $h_\theta(x) \in (0,1)$.

We can notice that the logistic regression model basically estimates the **probability** that $y = 1$.

For example, if $h_\theta(x) = 0.7$, it tells us that for that input there is a 70% probability that the output corresponds to the positive class ($y = 1$).



Therefore, the logistic regression model estimates the probability that $y = 1$, given input x and parameters θ, which can be written as:

$$h_\theta(x) = p(y = 1|x; \theta)$$

Since there are only two possible classes, we have:

$$p(y = 1|x; \theta) + p(y = 0|x; \theta) = 1$$

Given that, we can rewrite the second probability in function of the first as:

$$p(y = 0|x; \theta) = 1 - p(y = 1|x; \theta)$$

Logistic Regression - Cost Function

As with linear regression, also in logistic regression we want to solve a cost function minimization problem using the **Maximum Likelihood Estimate** (**MLE**), i.e. find the best parameters θ that produce the most probable output given the input → maximize the likelihood.

We can recap the same formula we have seen before, under the same assumptions (the training samples are independent and with identical distribution):

$$L(\theta) = L(y|X;\theta) = p(y|X;\theta) = \prod_{i=1}^{m} p\left(y^{(i)}|x^{(i)};\theta\right)$$

Since y can have only two possible outputs (0 and 1)

$$\begin{cases} p\left(y^{(i)} = 1|x^{(i)};\theta\right) = h_\theta\left(x^{(i)}\right) \\ p\left(y^{(i)} = 0|x^{(i)};\theta\right) = 1 - h_\theta\left(x^{(i)}\right) \end{cases}$$

we are facing a **Bernoulli distribution**, which its probability mass function (pmf) is given by:

$$p\left(y^{(i)}|x^{(i)};\theta\right) = h_\theta\left(x^{(i)}\right)^{y^{(i)}} \left(1 - h_\theta\left(x^{(i)}\right)\right)^{1-y^{(i)}}$$

Note: You can verify the probability of each outcome by yourself trying the two cases where $y(i)$ is equal to 0 or 1:

- If $y(i) = 1$ → $p\left(y^{(i)} = 1|x^{(i)};\theta\right) = h_\theta\left(x^{(i)}\right)^1 \left(1 - h_\theta\left(x^{(i)}\right)\right)^0 = h_\theta\left(x^{(i)}\right)$
- If $y(i) = 0$ → $p\left(y^{(i)} = 0|x^{(i)};\theta\right) = h_\theta\left(x^{(i)}\right)^0 \left(1 - h_\theta\left(x^{(i)}\right)\right)^1 = 1 - h_\theta\left(x^{(i)}\right)$

We can substitute the pmf in the expression of the cost function seen above, ending up with:

$$L(\theta) = \prod_{i=1}^{m} h_\theta\left(x^{(i)}\right)^{y^{(i)}} \left(1 - h_\theta\left(x^{(i)}\right)\right)^{1-y^{(i)}}$$

Now we can move to logarithmic form:

$$l(\theta) = \log \prod_{i=1}^{m} h_\theta\left(x^{(i)}\right)^{y^{(i)}} \left(1 - h_\theta\left(x^{(i)}\right)\right)^{1-y^{(i)}} = \sum_{i=1}^{m} \left(y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right)$$

At this point we can scale it by a factor of $1/m$ since it doesn't change the optimal parameters (as we already discussed in the previous chapter), but it helps us in maintaining consistency whenever the training dataset dimension m is. So, we get that our MLE is :

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right)\right]$$

We can move from maximization to the minimization problem by changing the sign:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[-\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right)\right]$$

Therefore, we got that the loss function for our logistic regression is given by:

$$\boldsymbol{J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right)}$$

This loss is known as **Binary Cross Entropy (BCE)** loss or **Logistic Loss**.

Error Contribution in Logistic Regression: The logistic function cannot be easily studied analytically, but we can get an intuition using the error contribution:

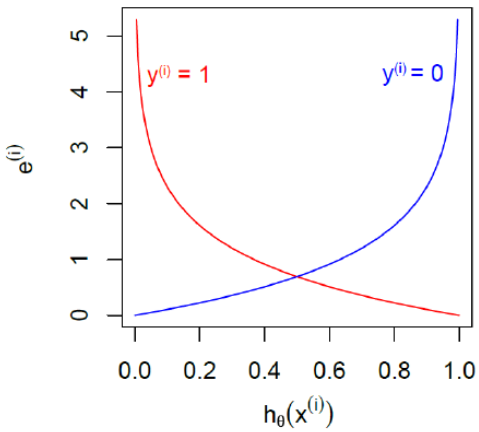
$$e^{(i)} = -y^{(i)} \log h_\theta\left(x^{(i)}\right) - \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)$$

If $\boldsymbol{y^{(i)} = 1}$ the error has value: $e^{(i)} = -\log h_\theta\left(x^{(i)}\right)$:

- If h(x) is low the error is high: $h_\theta\left(x^{(i)}\right) = 0 \Rightarrow e^{(i)} \to \infty$
- If h(x) is high the error is low: $h_\theta\left(x^{(i)}\right) = 1 \Rightarrow e^{(i)} \to 0$

If $\boldsymbol{y^{(i)} = 0}$ the error has value: $e^{(i)} = -\log\left(1 - h_\theta\left(x^{(i)}\right)\right)$:

- If h(x) is low the error is low: $h_\theta\left(x^{(i)}\right) = 0 \Rightarrow e^{(i)} \to 0$
- If h(x) is high the error is high: $h_\theta\left(x^{(i)}\right) = 1 \Rightarrow e^{(i)} \to \infty$



Logistic Regression – GD Update

To solve the problem of determining the values of the parameters such as minimizing the cost function, it is possible to use again the Gradient Descent (GD) method:

$$\theta_k = \theta_k - \alpha \frac{\partial J(\theta)}{\theta_j}$$

So, again, let's derive the explicit form of partial derivates required by the GD update.

To do it we calculate the partial derivative of the cost function with respect to any parameter θ_j:

$$\frac{\partial J(\theta)}{\theta_j} = \frac{\partial}{\theta_j}\left[-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right)\right] =$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} \frac{\partial}{\theta_j} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \frac{\partial}{\theta_j} \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right)$$

Since we don't know the direct derivative of it, it's better to derive the various parts that compose it.

Let's first derive the logistic function:

$$g'(z) = \frac{\partial}{\partial z}\left(\frac{1}{1 + e^{-z}}\right) = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1}{1 + e^{-z}}\left(1 - \frac{1}{1 + e^{-z}}\right) = g(z)(1 - g(z))$$

So, from it we can compute the derivate of $h_\theta(x)$ with respect to θ_j as:

$$\frac{\partial h_\theta(x)}{\partial \theta_j} = h_\theta(x)(1 - h_\theta(x)) \frac{\partial \theta^T x}{\partial \theta_j}$$

When we derived the GD update rule for the linear regression, we saw that $\frac{\partial \theta^T x}{\partial \theta_j} = x_j$, hence:

$$\boldsymbol{\frac{\partial h_\theta(x)}{\partial \theta_j} = h_\theta(x)(1 - h_\theta(x))x_j}$$

Using this last expression in the calculation of the two contributions relating to the starting partial derivative, we have:

$$\frac{\partial}{\partial \theta_j} \log h_\theta\left(x^{(i)}\right) = \frac{1}{h_\theta(x)} h_\theta(x)\left(1 - h_\theta(x)\right)x_j = \left(1 - h_\theta(x)\right)x_j$$

$$\frac{\partial}{\partial \theta_j}\log\left(1-h_\theta\left(x^{(i)}\right)\right)=\frac{1}{1-h_\theta(x)}\left(-h_\theta(x)\right)\left(1-h_\theta(x)\right)x_j=-h_\theta(x)x_j$$

From which, substituting the two developments just obtained in the original equation, we reach the last derivative step:

$$\frac{\partial J(\theta)}{\theta_j}=-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\frac{\partial}{\theta_j}\log h_\theta\left(x^{(i)}\right)+\left(1-y^{(i)}\right)\frac{\partial}{\theta_j}\log\left(1-h_\theta\left(x^{(i)}\right)\right)\right)$$

$$=-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\left(1-h_\theta\left(x^{(i)}\right)\right)x_j^{(i)}+\left(1-y^{(i)}\right)\left(-h_\theta\left(x^{(i)}\right)x_j^{(i)}\right)\right)$$

$$=-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\left(1-h_\theta\left(x^{(i)}\right)\right)+\left(1-y^{(i)}\right)\left(-h_\theta\left(x^{(i)}\right)\right)\right)x_j^{(i)}=$$

$$=-\frac{1}{m}\left(y^{(i)}-y^{(i)}h_\theta\left(x^{(i)}\right)-h_\theta\left(x^{(i)}\right)+y^{(i)}h_\theta\left(x^{(i)}\right)\right)x_j^{(i)}=$$

$$=\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x)-y^{(i)}\right)x_j^{(i)}$$

Therefore:

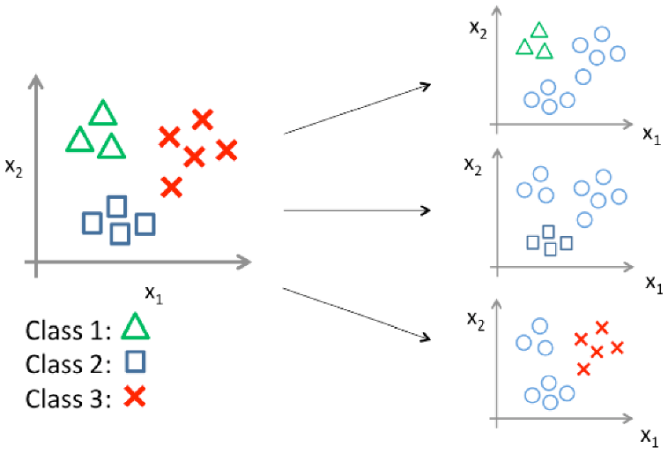
$$\boldsymbol{\frac{\partial J(\theta)}{\theta_j}=\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x)-y^{(i)}\right)x_j^{(i)}}$$

! We can notice this explicit form expression is the **same** as the one we obtained for linear regression.

2.3 Multiclass Classification (One-vs-All)

Every multiclass classification problem can be reconducted to a binary one. Basically, we can reduce the multiclass classification task to a series of binary classification problems using the **One-vs-All (OvA)** approach (sometimes it is also known as **One-vs-Rest**). This strategy involves training a separate binary classifier for each class, where each classifier distinguishes one class from all the others. For example, if we want to classify between triangles, squares, and crosses, we would create three binary classifiers:

1. **Classify triangles vs. non-triangles**
2. **Classify squares vs. non-squares**
3. **Classify crosses vs. non-crosses**



Each binary classifier will output whether the input belongs to its class or not, and the class with the highest confidence will be chosen as the final prediction. This way, we break down the multiclass problem into several binary classification problems.

$$h_\theta^{(i)}(x)=p(y=i|x;\theta)\quad(i=1,2,3)$$

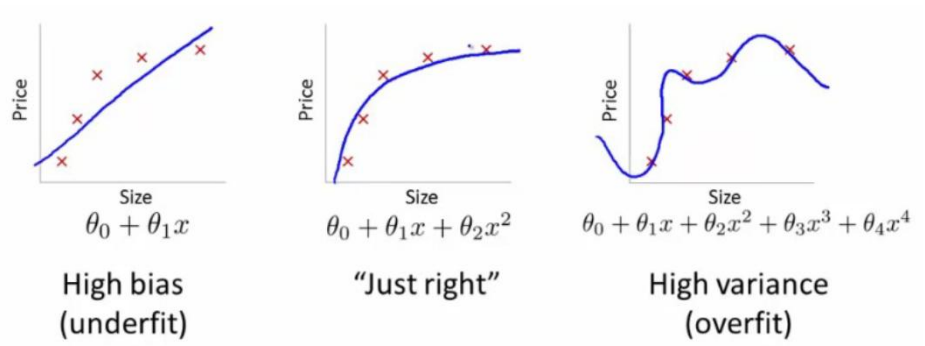
3 Fitting

As we discussed in previous sections, when using linear regression or logistic regression models, we need to choose a correct hypothesis that is able to best minimize the cost function of the model and, in particular, that is the one that best fits to our model. In this regard, in Machine Learning we talk about Fitting Problems when we have to evaluate whether the chosen hypothesis is the most suitable for describing the data and generalizing the model. In fact, our ultimate goal is not merely to replicate the trends in the training data but also to **generalize** effectively to unseen data—data that was not part of the training set (in the next chapter we will see that these unseen data are the ones forming the so-called **test set**).

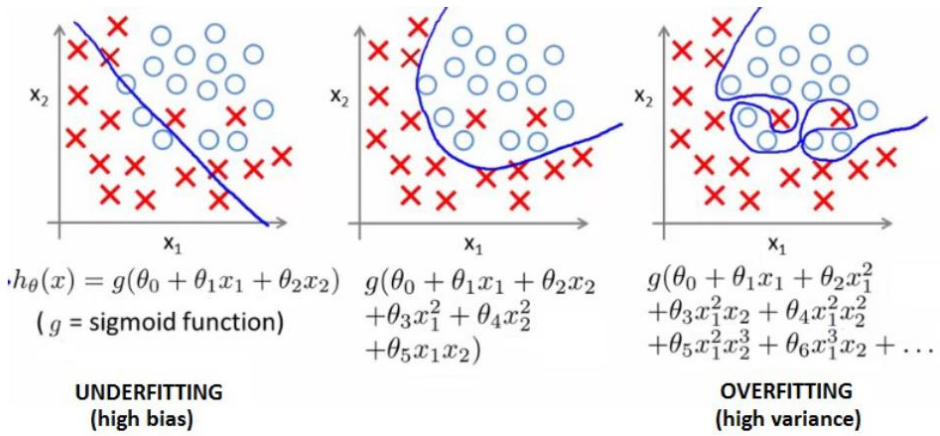
Therefore, "**Fitting**" concerns the problem of balancing the model's ability to capture patterns in the training data (by minimizing error) while also ensuring it generalizes well to new, unseen data. In this context, we can identify three main fitting scenarios:

- **Underfitting**: this is the case in which the model does not fit the training data because it is **too simple** and not able to learn. It performs badly on both training and test data.
- **Overfitting**: this is the case in which the model has been trained perfectly (too much) to predict the training examples, but as consequence it is no longer able to predict unseen examples and, therefore, to generalize. In this case the model "memorize" the training data rather than "learning" to generalize from the trend of the data itself. It performs well on training data but not on test data because it is **too complex** and not able to generalize.
- **Just Right (Good Fit)**: this is the optimal case where the model works correctly, approximating the data trend very well and moreover being **able to generalize** on the unseen examples.

Fitting - Regression



Fitting - Classification



3.1 Bias and Variance

The concept of fitting is strictly related to two other concepts known as bias and variance.

- **Bias**: is the systematic deviation of the model, i.e. it quantifies how far the model's average predictions are from the actual values:

  $$Bias\big(h(X), f(X)\big) = E[h(X)] - f(X)$$

  where:

  - $E[h(X)]$ is the **expectation (mean)** of the hypothesis $h(X)$, representing the average prediction across different training sets.
  - $f(X)$ is the true function we aim to estimate.

  In particular:

  - **Low bias** means the model's predictions are close to the true values on average, indicating that the model is capable of capturing the underlying patterns in the data.
  - **High bias** indicates that the model's predictions systematically deviate from the true values, often due to oversimplification. This results in consistent errors regardless of the training dataset, as the model fails to capture the complexity of the data.

- **Variance**: quantify how much the model's average predictions vary around their mean:

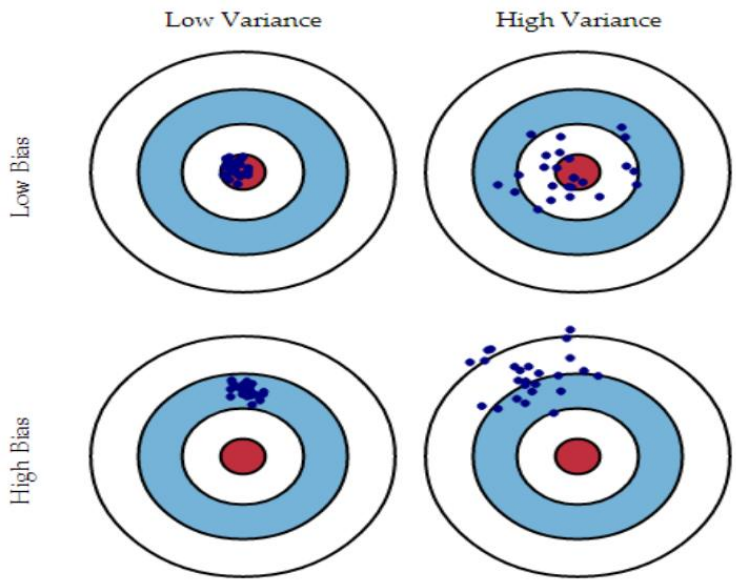
  $$Var(h(X)) = E(h(X) - E[h(X)])^2$$

  In particular:

  - **Low variance** means the predictions are consistent and tightly clustered (grouped) around their mean, indicating stability across different datasets.
  - **High variance** indicates that the predictions are widely dispersed, resulting in significant variability across different datasets. This means for instance that if we take the same model, but trained with a different dataset, it will result in significantly different values.

In the Figure below there is an illustrative example of all possible combinations of bias and variance. Note that the red region in the Figure corresponds to the actual value of the data to be estimated $(f(X))$¸ while the scattered points illustrate the model's predictions. As can be seen, there can be four scenarios:

- **Low Bias - Low Variance**: This is the ideal scenario where the model makes accurate and consistent predictions, with minimal error and little dispersion, but it's practically impossible.
- **Low Bias - High Variance**: The model's predictions on average are close to the true values (low bias), but they vary significantly (high variance), leading to dispersed predictions.
- **High Bias - Low Variance**: The predictions are consistent (low variance), but they are far from the true values, resulting in large errors (high bias).
- **High Bias - High Variance**: This is the worst-case scenario, where the model predictions have both high error (bias) and highly variability (variance), leading to poor performance.



So, at the end we can say that:

- Simple model → High bias → Underfitting
- Complex model → High variance → Overfitting

Therefore, what we want to obtain is a model of the right complexity (Just Right) that is able to generalize from the experience that it learned from the training dataset.

3.2 Bias and Variance Trade-off

When building predictive models, one of the key challenges we face is understanding and quantifying the sources of error in our predictions. The performance of a model is never perfect due to inherent limitations in the available data and the model itself. This naturally raises a fundamental question: how can we systematically analyze and minimize the sources of error to improve generalization to new data?

To address this, we first need to consider the nature of our data. In an ideal scenario, we would have access to an infinite number of samples, meaning our observed data represents the full distribution of possible values. However, in reality, we only have a finite dataset, which is a subset of the possible values a random variable can take.

Let's consider the nature of our data. In an ideal scenario, we would have access to an infinite number of samples, meaning our observed data represents the full distribution of possible values. However, in reality, we only have a finite dataset, which is a subset of the possible values a random variable can take.

To formalize this, let's assume that we have an infinite number of datasets D_i which represent all the possible combinations of the training samples that could feed our model, i.e., corresponding to all possible subsets of values assumed by the corresponding random variable.

At this point, we fix the model and we train it each time with a different dataset. From this process, we then obtain an infinite number of hypothesis functions $h^{(D_i)}(x)$.

Each training sample is of the form $< x^{(i)}, y^{(i)} >$, where $y^{(i)}$ is the sum of the true underlying function and a Gaussian error with zero mean and variance σ_e^2:

$$y^{(i)} = f(x^{(i)}) + e^{(i)}$$

Therefore, each hypothesis function $h^{(D_i)}(x)$ will be affected by an error in the predicted values w.r.t. the actual values.

We can model this error in the form of a Mean Square Error (MSE) also known as **Expected Squared Error**:

$$MSE(h^{(D_i)}(x)) = \mathbb{E}_x\left[(h^{(D_i)}(x) - y)^2\right] = \mathbb{E}_x\left[\left(h^{(D_i)}(x) - (f(x) + e)\right)^2\right]$$

This is what happens with respect to a single dataset among the i datasets defined previously. Our goal, however, is to estimate the expectation of the error with respect to all possible D_i datasets. Specifically, we want to compute the MSE of each hypothesis and then compute the overall mean. Performing this operation over all infinite datasets is not feasible, however what we can do is to compute the expected value of the mean as the expectation of the MSE over all datasets. This expectation is called **Generalization Error (GER)**:

$$GER = \mathbb{E}_D[MSE] = \mathbb{E}_D\left[\mathbb{E}_x\left[\left(h^{(D_i)}(x) - (f(x) + e)\right)^2\right]\right] =$$

$$= \mathbb{E}_x\left[\mathbb{E}_D\left[\left(h^{(D)}(x^{(i)}) - (f(x^{(i)}) + e^{(i)})\right)^2\right]\right]$$

Note that the last step can be performed because of the linearity of the expected value operator. In fact, computing the expected value with respect to D of the expected value with respect to x is equivalent to calculating the expected value with respect to x of the expected value with respect to D.

Okay, now let's focus on the term $\mathbb{E}_D\left[\left(h^{(D)}(x^{(i)}) - (f(x^{(i)}) + e^{(i)})\right)^2\right]$.

Reordering this term as:

$$\mathbb{E}_D\left[\left(\underbrace{h^{(D)}(x^{(i)}) - f(x^{(i)})}_{a} - \underbrace{e^{(i)}}_{b}\right)^2\right]$$

and using the identity $(a-b)^2 = a^2 + b^2 - 2ab$, we can rewrite it as:

$$\mathbb{E}_D\left[\left(h^{(D)}(x^{(i)}) - f(x^{(i)})\right)^2 + (e^{(i)})^2 - 2\left(h^{(D)}(x^{(i)}) - f(x^{(i)})\right)e^{(i)}\right]$$

Since expectation is linear, we can distribute it:

$$\mathbb{E}_D\left[\left(h^{(D)}(x^{(i)}) - f(x^{(i)})\right)^2\right] + \mathbb{E}_D\left[(e^{(i)})^2\right] - 2\mathbb{E}_D\left[\left(h^{(D)}(x^{(i)}) - f(x^{(i)})\right)e^{(i)}\right] =$$

$$= \mathbb{E}_D\left[\left(h^{(D)}(x^{(i)}) - f(x^{(i)})\right)^2\right] + \mathbb{E}_D\left[(e^{(i)})^2\right] - 2\mathbb{E}_D\left[\left(h^{(D)}(x^{(i)}) - f(x^{(i)})\right)\right]\mathbb{E}_D[e^{(i)}]$$

Now, recalling that for a Gaussian distribution $X \sim N(\mu, \sigma^2)$ its expectation is equal to the mean, i.e. $\mathbb{E}[X] = \mu$, in our case since we are considering a Gaussian error with zero-mean and variance σ_e^2, we will have $\mathbb{E}[e^{(i)}] = 0$ and since $Var(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$ we will also have $Var(e^{(i)}) = \mathbb{E}\left[(e^{(i)})^2\right] - 0 \Rightarrow \mathbb{E}\left[(e^{(i)})^2\right] = Var(e^{(i)}) = \sigma_e^2$. Therefore, we will obtain:

$$\mathbb{E}_D\left[\left(h^{(D)}(x^{(i)}) - f(x^{(i)})\right)^2\right] + \sigma_e^2$$

Now let's further try expanding the first term.

We can define a new quantity, the best estimation of $f(x^{(i)})$, by feeding each hypothesis with the same training sample, and then we compute the mean of all the values:

$$\bar{h}(x^{(i)}) = \mathbb{E}_D[h^{(D)}(x^{(i)})]$$

By adding and subtracting this last quantity from the initial term, we obtain:

$$\mathbb{E}_D\left[\left(\underbrace{h^{(D_i)}(x^{(i)}) - \bar{h}(x^{(i)})}_{a} + \underbrace{\bar{h}(x^{(i)}) - f(x^{(i)})}_{b}\right)^2\right]$$

Again, we can make use of the identity $(a+b)^2 = a^2 + b^2 + 2ab$ and obtain:

$$\mathbb{E}_D\left[\left(h^{(D_i)}(x^{(i)}) - \bar{h}(x^{(i)})\right)^2 + \left(\bar{h}(x^{(i)}) - f(x^{(i)})\right)^2 + 2\left(h^{(D_i)}(x^{(i)}) - \bar{h}(x^{(i)})\right)\left(\bar{h}(x^{(i)}) - f(x^{(i)})\right)\right]$$

and then exploit the linearity of the expected value operator:

$$\mathbb{E}_D\left[\left(h^{(D_i)}(x^{(i)}) - \bar{h}(x^{(i)})\right)^2\right] + \mathbb{E}_D\left[\left(\bar{h}(x^{(i)}) - f(x^{(i)})\right)^2\right] + 2\mathbb{E}_D\left[\left(h^{(D_i)}(x^{(i)}) - \bar{h}(x^{(i)})\right)\left(\bar{h}(x^{(i)}) - f(x^{(i)})\right)\right]$$

Observing the first two terms, it can be seen that:

- the first matches the definition of **variance**:

$$\mathbb{E}_D\left[\left(h^{(D_i)}(x^{(i)}) - \bar{h}(x^{(i)})\right)^2\right] = Var\left(h^{(D)}(x^{(i)})\right)$$

- the second matches the definition of **bias**, but squared:

$$\mathbb{E}_D\left[\left(\bar{h}(x^{(i)}) - f(x^{(i)})\right)^2\right] = \left(\bar{h}(x^{(i)}) - f(x^{(i)})\right)^2 = \left(\underbrace{\mathbb{E}_D[h^{(D)}(x^{(i)})] - f(x)}_{Bias}\right)^2$$
$$= Bias^2\left(h^{(D)}(x^{(i)}), f(x^{(i)})\right)$$

- the third term, after carrying out some algebraic steps, vanishes:

$$2\,\mathbb{E}_D\left[\left(h^{(D_i)}(x^{(i)}) - \bar{h}(x^{(i)})\right)\left(\bar{h}(x^{(i)}) - f(x^{(i)})\right)\right] =$$
$$= 2\left(\bar{h}(x^{(i)}) - f(x^{(i)})\right)E_D[h^{(D)}(x^{(i)}) - \bar{h}(x^{(i)})] =$$
$$= 2\left(\bar{h}(x^{(i)}) - f(x^{(i)})\right)(E_D[h^{(D)}(x^{(i)})] - E_D[\bar{h}(x^{(i)})]) =$$
$$= 2(\bar{h}(x^{(i)}) - f(x^{(i)}))\left(\bar{h}(x^{(i)}) - \bar{h}(x^{(i)})\right) = 0$$

Thus, at the end, we obtain:

$$\boldsymbol{MSE\left(h^{(D)}(x^{(i)})\right) = Bias^2\left(h^{(D)}(x^{(i)}), f(x^{(i)})\right) + Var\left(h^{(D)}(x^{(i)})\right) + \sigma_e^2}$$

This shows that the expected squared error is composed of three components: the **squared bias**, the **variance** and an **irreducible error** (inherent noise in the data that no model can eliminate).

Therefore, in order to reduce the Generalization Error, we should reduce the bias or the variance:

- Bias represents how close the best estimate function is to the ground truth. **High bias** for a specific model denotes a model that is too simple and cannot predict, no matter what dataset it is training on (it is underfitting).

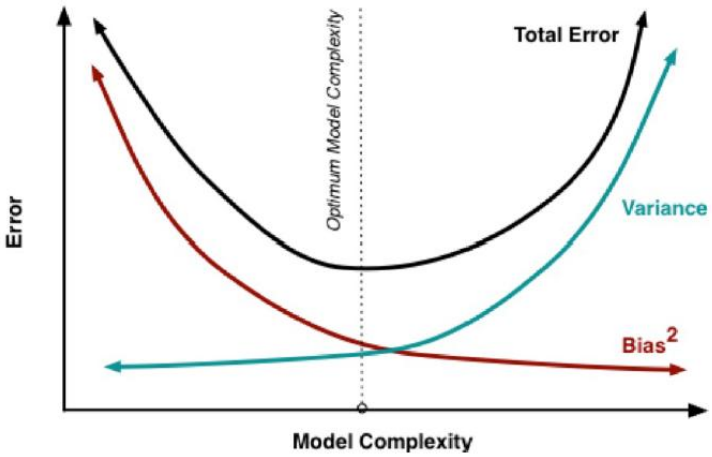
- Variance represents how much a single hypothesis can differ from the best estimate. **High variance** means that the same model, trained with different datasets, generates very different hypotheses. The hypotheses generated are very complex, are prone to overfit and are unable to generalize.

The optimum would be to mantain low bias and variance; however, given a model, the bias and the variance behave at the opposite w.r.t. the complexity of the model. Below is a descriptive graph of the relationship between bias, variance and model complexity.

As can be seen from the Figure:

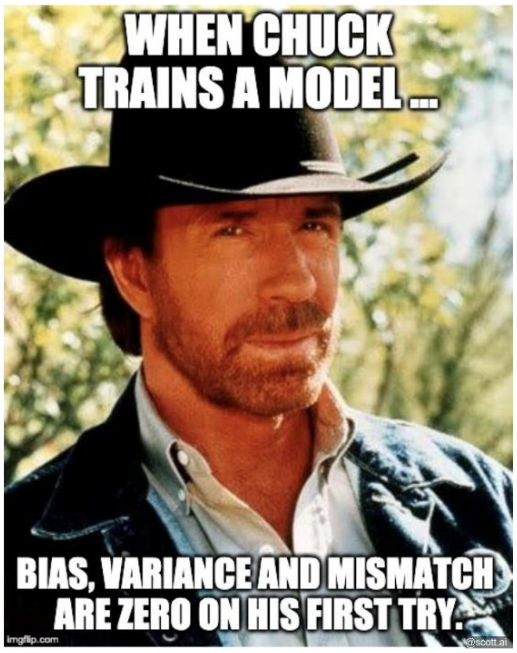
- Simple model = low variance, **high bias**
- Complex model = **high variance**, low bias

Therefore, it is necessary to make a **trade-off** in choosing to minimize bias rather than variance.

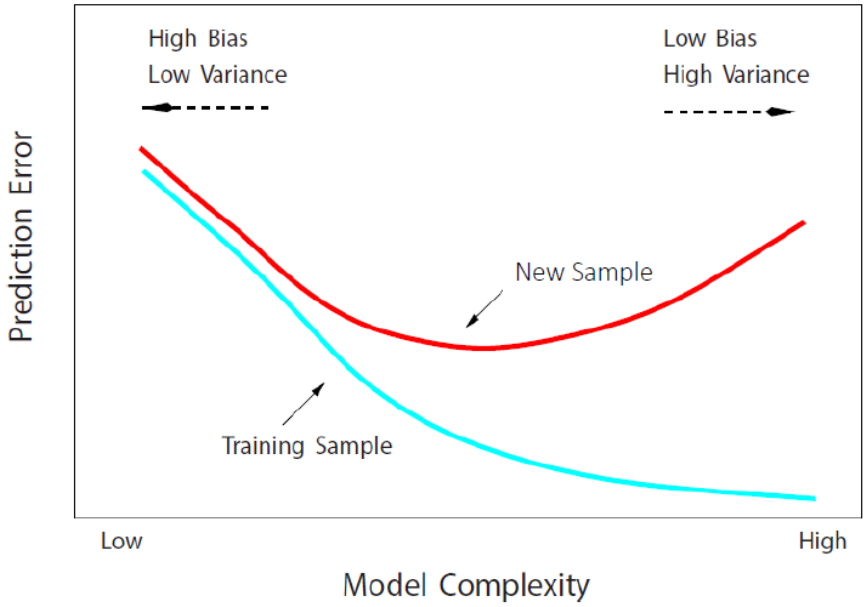


The **best trade-off** is to choose the model complexity that shows the **minimum sum between bias and variance**. At that point we will be able to state that we cannot do better in terms of error reduction and the model's generalization capacity.

Above we saw that that there is now way to reduce bias without increasing at the same time the variance (and vice versa) … however …



Below in Figure is shown the relationship between training/test set and model complexity.



As can be seen from the Figure:

- Simple model = the error on the training set and that on the test set are both very high (**High bias** – Low variance) → **Underfitting**
- Complex model = the error on the training set decreases, but the error on the test set increases because the model is not able to generalize (Low bias – **High variance**) → **Overfitting**

It follows that, if we also want to identify an "equilibrium point" in this case, this point will be determined close to the instant from which the error on the test set begins to increase.

4 Regularization

To mitigate overfitting, there are mainly two options to choose from:

1. **Reduce the number of features:** This can be done by manually selecting a subset of relevant features or using a feature selection algorithm to retain only those that contribute most to the model's accuracy.

- **Regularization:** Instead of removing features, we keep all features but reduce the magnitude/values of the parameters. This is particularly useful when there are many features, each contributing slightly to the output prediction.

4.1 Intuition Behind Regularization

To better explain how Regularization works, let's take into consideration the following practical example.

Consider a high-degree polynomial model:

$$h(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

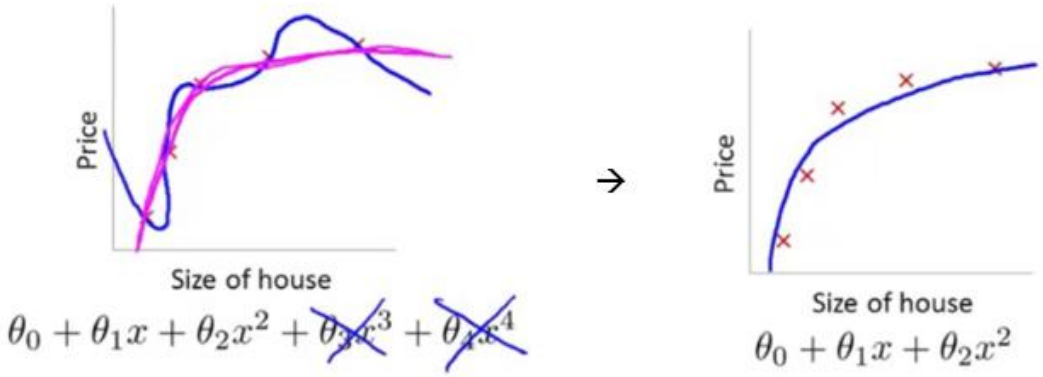
As we can notice by looking at the Figure below in this case the model in an overfitting situation. To address this, we can penalize higher-degree terms, reducing their influence.

In fact, by adding a large penalty (e.g., 1000) to the higher-degree parameters, we are effectively forcing the minimization function to make them **really small**:

$$\min_\theta J(\theta) = \min_\theta \frac{1}{2m} \sum_{i=1}^{m} \left(h\left(x^{(i)}\right) - y^{(i)}\right)^2 + 1000\,\theta_3 + 1000\,\theta_4$$

$$\Rightarrow \boldsymbol{\theta_3 \approx 0}, \quad \boldsymbol{\theta_4 \approx 0}$$

Basically, the optimization process will shrink these higher-degree parameters (θ_3, θ_4) toward zero (they assume very small values close to zero), effectively simplifying the model and smoothing the curve.



Key Idea: **smaller parameter values** result in a **simpler hypothesis**, which is less prone to overfitting.

→ **Regularization** involves adding a **penalty term** to the cost function in order to discourage the parameters from reaching large values (we want to keep them small).

4.2 Regularization in Linear Regression

Starting with the original cost function of linear regression (MSE):

$$J(\theta) = \frac{1}{2m}\left(\sum_{i=1}^{m}\left(h\left(x^{(i)}\right) - y^{(i)}\right)^2\right)$$

to regularize, we add a **penalty term**:

$$J(\theta) = \frac{1}{2m}\left(\sum_{i=1}^{m}\left(h\left(x^{(i)}\right) - y^{(i)}\right)^2\right) + \frac{1}{2m}\boldsymbol{\lambda}\|\theta\|_2^2 = \frac{1}{2m}\left(\sum_{i=1}^{m}\left(h\left(x^{(i)}\right) - y^{(i)}\right)^2\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

where here λ is a regularization hyperparameter which controls the influence of the regularization term with respect to the hypothesis: the bigger it is, the smaller the parameters will be and the more "smother" the curve will be.

Note: In the regularization term, note that j starts from 1 since θ_0 called **bias term** is excluded from regularization. This is because it is not associated with any features (there is no feature that multiplies it), so it doesn't make sense to count it.

In the formula above as regularization term an l2-norm is used, but other norms, such as the l1-norm, can also be used. We will see these two in more detail later.

Tuning λ:

The regularization hyperparameter λ controls the strength of the penalty:

- **Large λ:** If λ is too large, the model may underfit because the parameters are forced to be too small, making the model too simple.
    - **λ too high → Underfitting**
- **Small λ:** If λ is too small, the regularization effect is weak, and the model may still overfit.
    - **λ too low → Overfitting**

The goal is to choose an optimal λ that results in a balanced model that achieves a good fit.

Note: You can notice that in the formula above factor of $1/2$ is also included to simplify the derivative calculations. What is more interesting to think about is term $\frac{1}{m}$: In some cases, you will see the factor $\frac{1}{m}$ omitted and this is typical when the regularization is adopted in an absolute sum loss function such as the SSR to the rather than a scaled one such as the MSE. In fact, if your loss function is already averaged over the samples (e.g., MSE), it makes sense to scale the regularization term in the same way to ensure the regularization and loss terms are on comparable scales. Moreover, dividing by m ensures that the contribution of the regularization term doesn't grow disproportionately with the dataset size, making it consistent regardless of how many samples you have.

Regularized Gradient Descent for Linear Regression

After adding regularization, the GD update rule for linear regression becomes:

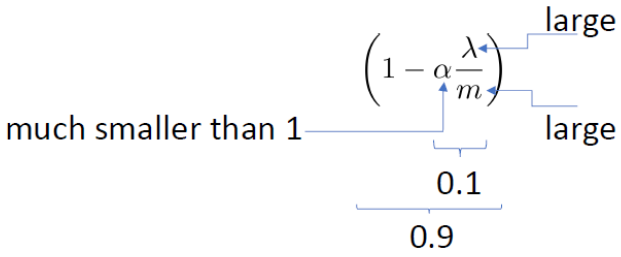
$$\begin{cases} \theta_0 = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_0^{(i)} \\ \theta_j = \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)} + \frac{\lambda}{m} \theta_j\right], \qquad j = \mathbf{1}, 2, \dots, n \end{cases}$$

where here the bias term θ_0 is updated separately because it is not regularized.

The update rule can also be written in a more compact form as:

$$\theta_j = \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)} \quad j = 1,2, \dots, n$$

It is worth noting that the second part is same as before applying the regularization, while the first part introduces a shrinkage factor:



This factor is less than 1, meaning that it gradually reduces the magnitude of θ_j on each iteration, effectively regularizing the model by discouraging overly large parameter values.

4.3 Regularization in Logistic Regression

The same concept applies to logistic regression. After adding the regularization term, the GD update rules for logistic regression becomes:

$$\begin{cases} \theta_0 = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_0^{(i)} \\ \theta_j = \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)} + \frac{\lambda}{m} \theta_j\right], \qquad j = \mathbf{1}, 2, \dots, n \end{cases}$$

Remember that we have shown (in the previous chapter) that the derivative of the cost function in logistic regression is the same as that for linear regression, so the regularized update rules are identical. Thus, the single expression for both is:

$$\boldsymbol{\theta_j = \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)} \quad j = 1,2, \dots, n}$$

4.4 Regularization with L2-norm and L1-norm

The $\boldsymbol{l^2}$ **norm** (also known as ***weight decay***) penalizes the sum of the squares of the model's parameter values:

$$l^2 = \|\theta\|_2^2 = \sum_{j=1}^{n} \theta_j^2$$

This regularization smooths the model by penalizing large parameters encouraging their values to decay **toward zero** (but not exactly zero) unless supported by the data.

Another commonly used penalty term is the $\boldsymbol{l^1}$ **norm** which penalizes the sum of the absolute values of the model's parameters:

$$l^1 = \|\theta\|_1 = \sum_{j=1}^{n} |\theta_j|$$

This regularization encourages some parameter values to become **exactly zero**.

This characteristic makes l^1 effectively acting as an **implicit feature selection method**, as it can automatically exclude irrelevant features from the model.

While l^2 regularization smooths the model by preventing large parameters and keeping all features in the model (therefore it never completely eliminates features), l^1 regularization tends to result in a sparse model, where irrelevant features are driven to zero. This makes l^1 regularization especially valuable in high-dimensional data sets with thousands of features, as it implicitly selects the most important features for the model.

Geometric Interpretation

We said that l^2 leads some parameters values to decay **toward zero**, while l^1 to exactly zero, but we didn't say why this happens. To answer it let's try to give a geometric interpretation.

Suppose we want to minimize the loss function (or cost function) while ensuring that the parameters satisfy either an l^1 or l^2 constraint:

$$Norm\ l^2: \quad \min \sum_{i=1}^{n} (\dots)^2 \quad s.t\ \ \|\theta\|_2^2 < t$$

$$Norm\ l^1: \quad \min \sum_{i=1}^{n} (\dots) \quad s.t\ \ \|\theta\|_1 < t$$

Let's consider a simplified case by limiting the analysis to two parameters:

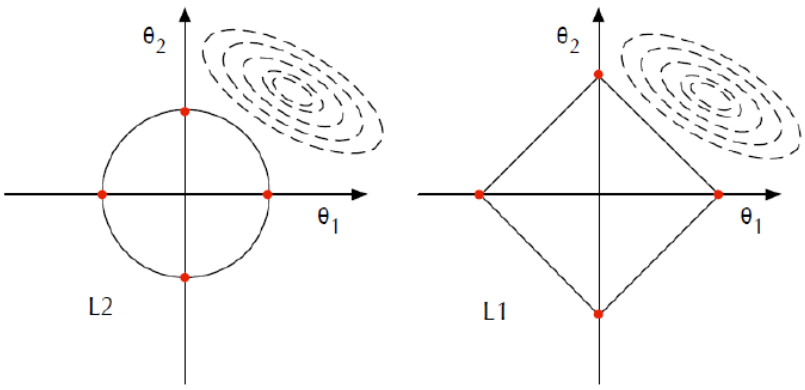
$$Norm\ l^2: \quad \min \sum_{i=1}^{n} (\dots)^2 \quad s.t\ \ \theta_1^2 + \theta_2^2 < t$$

$$Norm\ l^1: \quad \min \sum_{i=1}^{n} (\dots) \quad s.t\ \ |\theta_1| + |\theta_2| < t$$

Then:

- $\sum(\dots)^2$, which is our loss, is a quadratic function (a parabola) and looks like an elliptical shape in 2D space.

- $\theta_1^2 + \theta_2^2 < t$, which is our l2 constraint, forms a **circle** in 2D.
- $|\theta_1| + |\theta_2| < t$, which is our l1 constraint, forms **rhombus** in 2D.

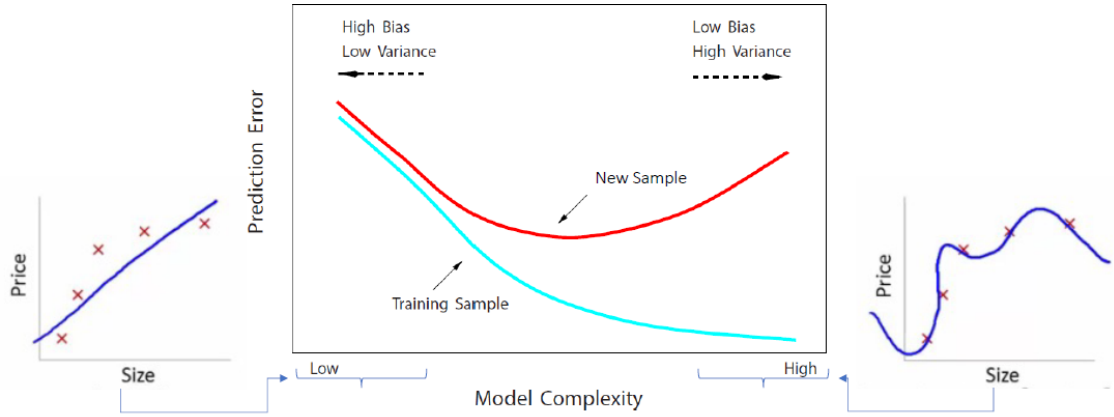


These shapes define the boundary of the region where the regularization allows the parameters to be. So, during the minimization we aim to minimize that parabolic curve while staying within the boundary defined by the regularization constraint (either l^2 or l^1).

- With the **circular constraint** from l^2, the minimum point on the loss surface will generally touch the boundary at some point on the circle, but not at the axes. This means that the parameters θ_1 and θ_2 are **reduced**, but they **rarely reach exactly zero** because the circle's smooth boundary distributes the reduction evenly across both parameters. As a result, l2 regularization shrinks parameter values but typically does **not eliminate** any of them, meaning all features are retained albeit with smaller parameters values.
- On the other hand, with the **rhombus constraint** from l^1, the corners of the rhombus are sharp and located on the axes (i.e., at points where either $\theta_1 = 0$ and $\theta_2 = 0$). When minimizing the loss, the algorithm is more likely to **hit one of these corners** due to the shape of the rhombus. When this happens, one of the parameters (either θ_1 or θ_2) becomes **exactly zero**. This behavior is the key reason why l^1 regularization leads to an effective **feature selection.** In higher dimensions (with more parameters), this process results in many parameters being set to zero, leaving only the most important features with non-zero coefficients.

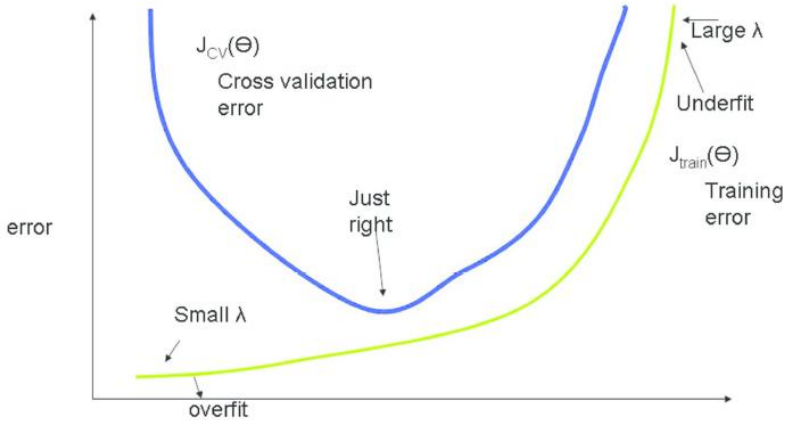
4.5 Regularization VS Bias/Variance

As we know, by decreasing the complexity of the model too much, the error on the training set and that on the test will both increase (High bias – Low variance), producing an Underfitting phenomenon. Conversely, by increasing that complexity excessively, the error on the training set will be reduced, but the error on the test set will increase since the model will not be able to generalize (Low bias – High variance), producing an Overfitting phenomenon.



Addressing the same considerations with respect to the λ regularization hyperparameter, we can therefore say that, by using too high λ values, we tend towards Underfitting, vice versa, by using too low values we remain (or arrive) in an Overfitting situation.

- **Large λ → Underfitting**
- **Small λ → Overfitting**

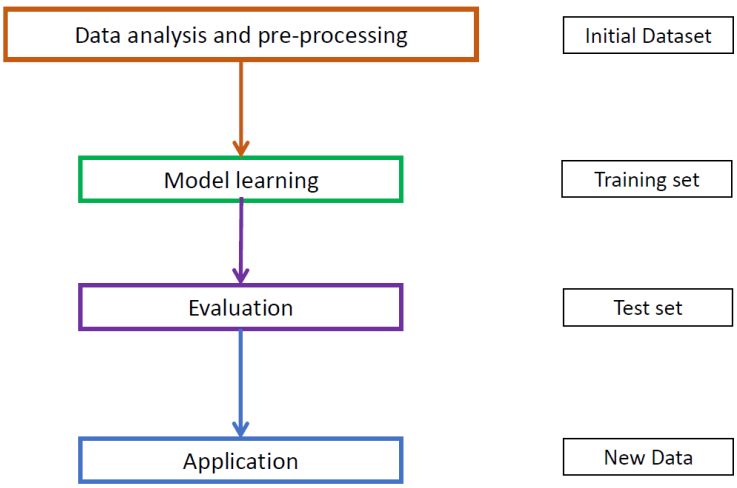


5 How to Build an ML System

The learning process grounds on three parts: **representation** + **optimization** + **evaluation**

1) **Representation**: identify the space of the learner's hypothesis (or in other words choose the model) and decide which features to use to represent the data.
2) **Optimization**: choose the method to train the model (e.g. GD, greedy search).
3) **Evaluation**: choose an evaluation function (score/objective function) to distinguish a good from a bad learner.

Goal: generalize well from seen examples (training set) on unseen examples.



On the left is shown the life cycle of an ML system, at least as a first draft. It consists of four parts, starting with a preliminary ***data analysis and pre-processing*** phase in which, starting from "real" data, cleaned data is obtained from which a training set is composed. Using the latter it is therefore possible to train one or more models (**model learning**) and then, after having identified the best of all in the **evaluation** phase, use the latter in specific **applications** starting from new unseen data.

5.1 Data Analysis and Pre-Processing

Real world data are “dirty”. Very often they turn out to be:

- **Incomplete**: the values of some features are missing, or even interesting features are completely missing.
- **Inaccurate**: data contains incorrect values resulting deriving from inaccurate or partial (biased) observations.

This leads to the famous golden rule: **“GIGO” → Garbage In Garbage Out**. Basically, if we put unusable, useless, or corrupted input we will always get a bad output as a result, no matter how good the model is.

An accurate data analysis and cleaning is therefore necessary, that’s why pre-processing is a very important phase. This phase generally takes a long time and consists of the following steps:

- Cleaning of data: outliers removing, noise removing, duplicates removing
- Transforming data:
  - Discretize
  - Aggregate
  - Normalization and re-scaling
- Creating new features

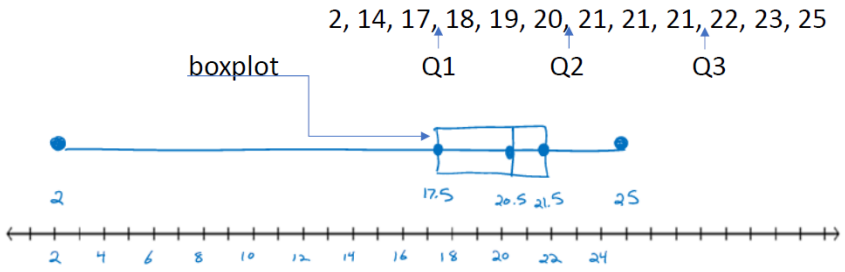
Removing Outliers

Outliers can be identified using various methods, such as quartiles, deciles, or percentiles.

Fractiles	Summary	Symbols
Quartiles	Divide a data set into four equal parts.	Q1 , Q 2, Q3
Deciles	Divide a data set into ten equal parts.	D1 , D2 , D3 ...D9
Percentiles	Divide a data set into one hundred equal parts.	P1, P2, P3 ... P99

Specifically, **quartiles** divide a dataset (sorted in ascending order) into four equal parts:

- Q_2 **(Second Quartile)** is equal to the **median** of the dataset.
- Q_1 **(First Quartile)** is the median of the values that are below Q_2.
- Q_3 **(Third Quartile)** is the median of the values that are above Q_2.



$$Q_2 = \frac{20+21}{2} = 20.5$$
$$Q_1 = \frac{17+18}{2} = 17.5$$
$$Q_3 = \frac{21+22}{2} = 21.5$$

Using these quartiles, we can compute the **Interquartile Range (IQR)**, defined as:

$$\boldsymbol{IQR = Q_3 - Q_1}$$

The IQR measures the spread of the middle 50% of the data. In this case $IQR = 21.5 - 17.5 = 4$.

The IQR can be used to detect outliers in a dataset based on the following rule:

> "If a data value is less than $\boldsymbol{Q_1 - 1.5 * IQR}$ or greater than $\boldsymbol{Q_3 + 1.5 * IQR}$, it is considered an **outlier**".

In this case, the lower bound is $17.5 - 1.5 * 4 = 11.5$ which shows that the value "2" < 11.5 is an outlier. While since the upper bound is $21.5 + 1.5 * 4 = 27.5$ the value "25" is not an outlier.

Note: Quartiles are commonly used to create a **boxplot**, a graphical representation of the data. In a boxplot, the "box" represents the interquartile range (IQR), with the median (Q2) shown inside the box. Outliers, identified using the method described, are typically plotted as individual points outside the whiskers of the box.

Feature Selection

Sometimes there are many features, some of which may be redundant or not very important; for this reason, it is important to make a selection of the features most relevant for the learning task. This operation can be carried out via:

- **Domain expertise**: Relying on experts who have knowledge of the specific field to identify relevant features.
- **Filters**: These techniques assess the importance of each feature by calculating metrics such as Information Gain, Entropy, or Mutual Information to rank features according to their usefulness in distinguishing between classes.
- **Wrappers**: An iterative method that evaluates various subsets of features to find the best combination for the model.
- **Dimensionality reduction**: Methods like **PCA** or **SVD** reduce the number of features by transforming the data into a lower-dimensional space while preserving as much information as possible.

5.2 Hypothesis Evaluation

Typically, in order to perform the model learning task, the starting dataset obtained following the pre-processing phase is then splitted (this split is usually operated in a random way) in two portions:

- **Training set**: Dataset used to train the model.
- **Test set**: Dataset composed of unseen examples, used to evaluate the real performance of the model.

After splitting the data, the model is trained on the **training set**, where it learns the parameters θ by minimizing the training error $J(\theta)$. Once the model is trained, we calculate the **test set error** to assess the model's generalization ability.

The **test set error** is crucial because it represents the only indicator of the real performance of the model on data it has never seen before. In fact, without it we would keep training the model to achieve a very low training error without taking into consideration that it is wrong since we reach it just because we are aligning more and more with the data leading to overfitting. Training errors don't reflect the model's ability to generalize to new data. As overfitting occurs, the model performs well on the training data but poorly on unseen examples, so the actual generalization error is likely higher than the training error. For this reason, it is necessary to evaluate the hypothesis on a portion of data never used before (the test set).

For the two linear and logistic regression cases respectively, the test set error will be equal to:

- Test set error for linear regression:

$$J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

- Test set error for logistic regression:

$$J_{test}(\theta) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \left(y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right)$$

Said that, imagine now that you want to compare several hypotheses characterized by polynomials of different degrees:

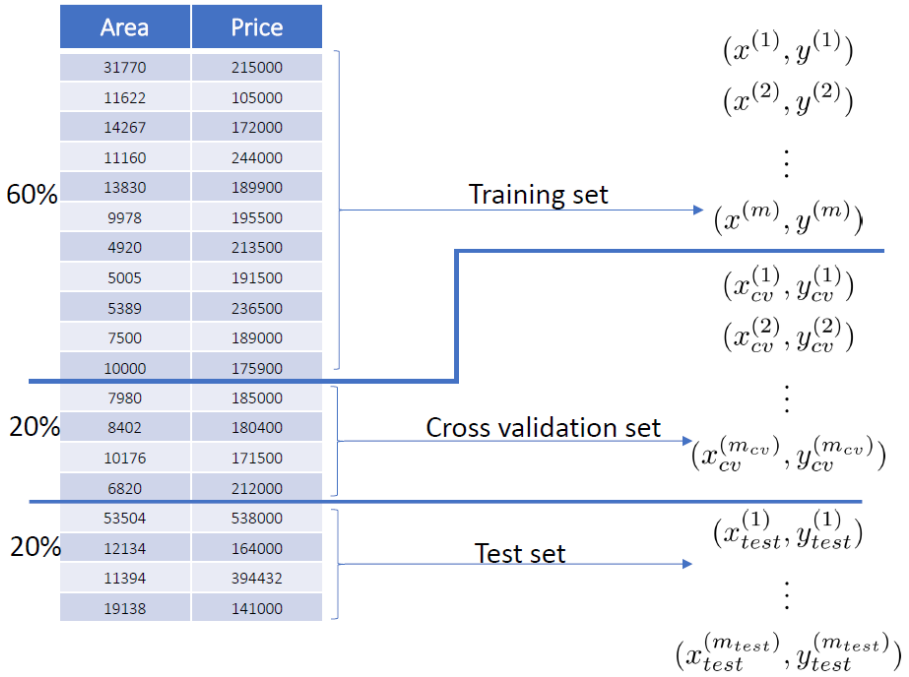
$$\begin{array}{llll} d1 & 1. & h_\theta(x) = \theta_0 + \theta_1 x & \rightarrow \theta^{(1)} \rightarrow J_{test}(\theta^{(1)}) \\ d2 & 2. & h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 & \rightarrow \theta^{(2)} \rightarrow J_{test}(\theta^{(2)}) \\ d3 & 3. & h_\theta(x) = \theta_0 + \ldots + \theta_3 x^3 & \rightarrow \theta^{(3)} \rightarrow J_{test}(\theta^{(3)}) \\ & \vdots & \vdots & \\ d10 & 10. & h_\theta(x) = \theta_0 + \ldots + \theta_{10} x^{10} & \rightarrow \theta^{(10)} \rightarrow J_{test}(\theta^{(10)}) \end{array}$$

Imagine that the polynomial of degree 5 shows the lowest test cost function $J_{test}\left(\theta^{(5)}\right)$. Is it possible to say that this model is the one that generalizes best?

Actually **no**, since the hypothesis we have chosen is the one that guarantees the best performance on that specific test set and, for this reason, is likely to be an optimistic estimate of the generalization error. Therefore, we need an additional "test set" → the convention is to use a part of the training set as **validation set**.

Given a dataset, we therefore divide it into three sets:

- **Training set**: used to train models.
- **Validation set**: used as a "test set" for each model among those present, so as to be able to compare errors and choose the best model. It is also useful for selecting features and setting hyper-parameters.
- **Test set**: used to evaluate the best model previously chosen.



In summary:

- **Can we choose the model that gives the best performance on the Training Set? NO!**
  - We risk overfitting: the model will not be able to generalize.
  - The error on the training data is an underestimation of the generalization error.
- **Can we choose the model that gives the best performance on the Validation Set? YES!**
  - We must train the model on the training set and then validate it on the validation set.
  - In this way we evaluate the generalization error on unseen examples.

Usually, the validation set consists of 1/3 of the training set (hold-out validation), or you can use k-folds cross-validation.

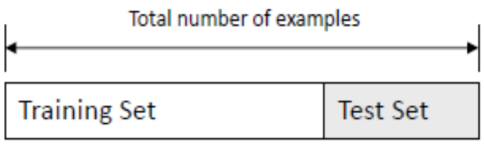
The validation set therefore allows us to choose the best model among all the candidates. Once the latter (the best model) has been identified, to finally evaluate it we need a Test Set: data not in Training and Validation sets.

As mentioned, evaluating the model on a test set is useful to see if the model really generalize well enough.

!: It is not correct to use the test set for choosing a model for an algorithm. You must use the test set only when the model is chosen and you want to evaluate it.

Hold-out

The **Hold-out** method is commonly used to split a dataset into **training** and **test sets**. Typically, the data is divided so that 80% is used for training and 20% for testing.



A key drawback of this simple Holdout method is that the specific way the data is split can influence model evaluation. For example, the samples in the training and test sets may not be representative of the overall dataset, and class imbalances can occur between the two sets. To address this, **Stratified Sampling**, also known as

Stratification, can be used. Stratified sampling ensures that the distribution of classes in the training and test sets reflects their proportions in the entire dataset. For example, if a dataset contains 70% samples from Class A and 30% from Class B, stratification will maintain this ratio in both the training and test sets. Therefore, it makes the split by preserving the percentage of samples for each class which ensures that class distributions are proportionally represented in both the training and test sets.

Using just a single training/test partition as done in Hold-out can introduce another issue: **the performance of the model can become overly dependent on the specific partition used, leading to high variance in evaluation metrics and a lack of robustness in assessing the model's true generalization capability.** To mitigate this issue, an additional split is often made within the training dataset, creating a **validation set**.

As we already anticipated, a way to do this is to perform an additional split within the training set to create a **validation set**, with a typical split of 70% for training and 30% for validation. This is also known as **Hold-out Validation**.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model. In fact:

- a larger validation set provides a more accurate estimate of the model's performance (i.e., lower variance)
- on the other hand, … a larger training set is necessary for a more effective learning process.

Additionally, the results can be sensitive to the specific random split of the data. A single (train-validation) split may not capture how robust the model is to variations in the data (i.e. how sensitive the model's accuracy is to a particular training sample). To address these issues, **Cross Validation (CV)** techniques are often used.

K-Fold Cross Validation

K-fold Cross Validation is a CV technique where the training set is split into k **equally sized subsets** called folds. Of the k folds, **only one** is kept as the validation set and all the others are used together as the training set (**«leave-one-out»** method). The cross-validation process is repeated $\boldsymbol{k}$ **times**, with each of the k folds used exactly once as a validation set. The k results can then be averaged to produce a single estimation.

Case i	Train on					Test on	Error
Case1		F2	F3	F4	F5	F1	1.5
Case2	F1		F3	F4	F5	F2	0.5
Case3	F1	F2		F4	F5	F3	0.3
Case4	F1	F2	F3		F5	F4	0.9
Case5	F1	F2	F3	F4		F5	1.1

$$Error = \frac{1}{k}\sum_{i=1}^{k} Error_i$$

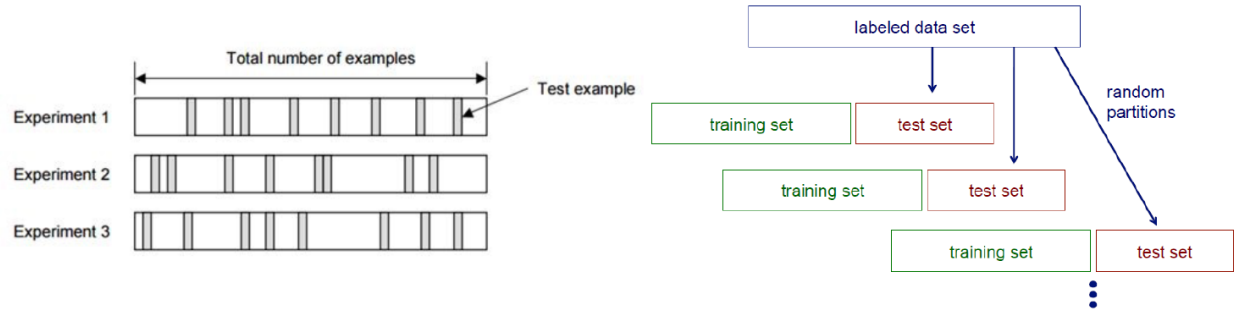
This way, the advantage you get is that all observations are used for both training and validating, and each observation is used for validation exactly once. **10-folds cross validation** is usually used.

Note: To ensure that in the k fold used the class distributions are balanced, it is a good idea to apply stratification also to K-Fold Cross Validation.

Random Subsampling

As for the "sensitivity to the particular split" problem mentioned above with the Holdout method you can address it by adopting random subsampling. In **Random Subsampling**, the dataset is repeatedly split into random training and validation sets. Specifically:

- **K splits** of the dataset are made, with each split **randomly** selecting a fixed number of examples **without replacement**.
- For each split, the classifier is retrained from scratch using the training examples, and the error is estimated on the validation samples.
- At the end we average the result of these K evaluations to get the final estimation.



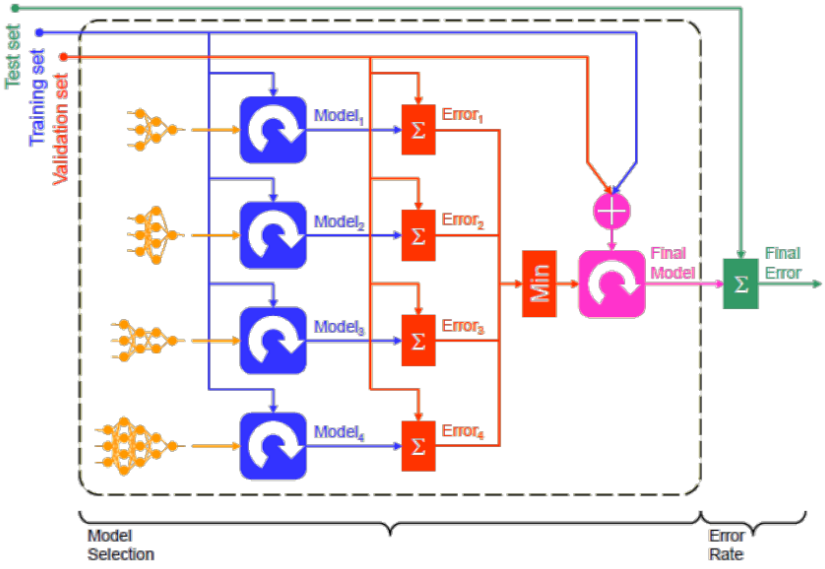
By repeating this process, random subsampling provides a more robust estimate of model performance, as it reduces the variance caused by any particular data partition.

! ***K-fold cross validation vs random subsampling***: In **K-fold cross-validation**, each data point is guaranteed to be in the validation set exactly once, ensuring complete coverage. In **random subsampling**, we basically perform K experiments (K hold-outs) where splits are done randomly each time, so some data points may appear in the validation set more than once, while others may never appear.

Evaluation Overview

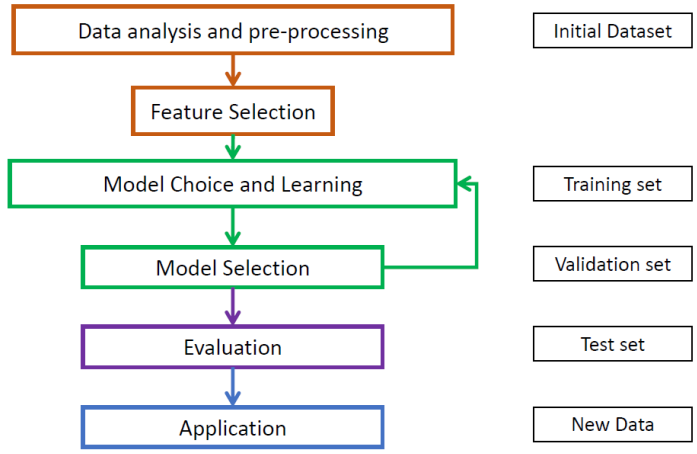
To evaluate and select the best model, we first split the dataset into a training set and test set, typically with an **80-20 split**. This can be done with standard **Holdout** or with its **stratified** version.

The training set is further split into **training** and **validation** sets. This can be done using **K-fold Cross-Validation** (or its stratified version), **Random Sampling** or a simpler **Hold-out Validation**, depending on the complexity of the task. The key idea is to use the training set for learning the model and the validation set to estimate the



model's performance. We then compare models based on their validation errors and select the one with the **lowest error**.

Once the best model is selected, we use the **test set** to compute the final test error and evaluate how well the model performs on unseen data.



For completeness, the life cycle of an ML system is shown on the left, updated following the clarification of the previous concepts. It was enriched by further steps such as the feature selection in the pre-processing phase and the model selection subsequently implemented through the use of the Validation set.

6 Diagnostic and debugging

Suppose after following all the steps we've discussed, the model still performs poorly. How should we proceed to resolve this issue? Should we gather more training data, reduce the feature set, add new features (perhaps polynomial ones), adjust the regularization parameter (Lambda), or try something else? The key to making the right choice lies in **diagnostics**.

Diagnostic is a test that can be performed to gain insight into what is/isn't working with a learning algorithm and obtain indications on how best to improve its performance.

Diagnostics can take time to implement, but doing so can be a very good use of your time.

A good starting point is to determine whether the model's poor performance is due to **bias** or **variance**:

$$Bias\ (underfit):\quad J_{train}(\theta)\ will\ be\ high\ \ \&\ J_{cv}(\theta) \approx J_{train}(\theta)$$

$$Varaince\ (overfit):\quad J_{train}(\theta)\ will\ be\ low\ \ \&\ J_{cv}(\theta) \gg J_{train}(\theta)$$

The same investigation could be conducted with respect to the Lambda regularization parameter, based on the concepts already known. We can therefore say that, by using too high Lambda values, we tend towards Underfitting, vice versa, by using too low values we remain (or arrive) in an Overfitting situation.

To better understand which factor is affecting model performance, we can utilize the **learning curves**.

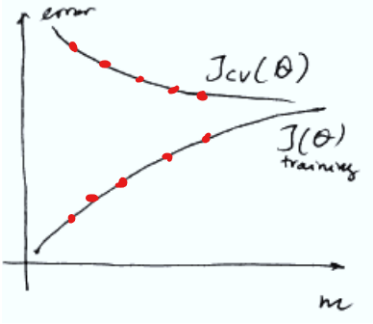
6.1 Learning Curves

Learning curves is a good technique

- To sanity-check a model
- To improve performance

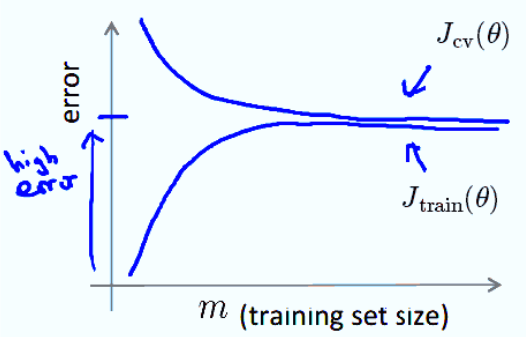
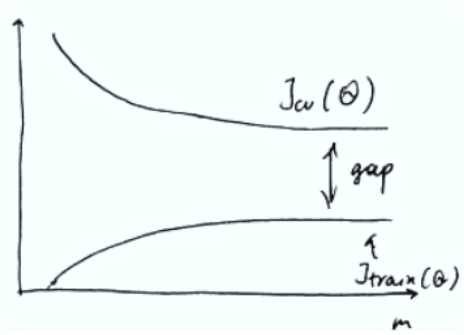
A learning curve is a plot where we have two functions of m (m is the set size):

- training set error $J_{train}(\theta)$
- the cross-validation error $J_{cv}(\theta)$



The idea behind the construction of learning curves is to artificially reduce the dimensions of the training set and then progressively increase them ($m = 1$, then $m = 2$ and so on). For each m, the training and validation errors are calculated and plotted. The result of this process will be the plot of the learning curves. By analyzing these learning curves it will be possible to notice how, as the size of the training set increases, the training and cross-validation errors evolve.

We can end up in two interesting scenarios that we can diagnose:

Scenario	
Underfitting (High Bias)	**Overfitting (High Variance)**
	
Characteristics	
Both $J_{train}(\theta)$ and $J_{cv}(\theta)$ remain high and close to each other. This indicates that the model is too simple to capture the underlying patterns in the data.	$J_{train}(\theta)$ is low, but $J_{cv}(\theta)$ is much higher, creating a large gap between the two. This suggests that the model fits the training data well but fails to generalize.
Solutions	
Decrease λ Increase complexity of the model Increase the set of features	Increase λ Decrease complexity of the model Decrease the set of features Increase the amount of training examples

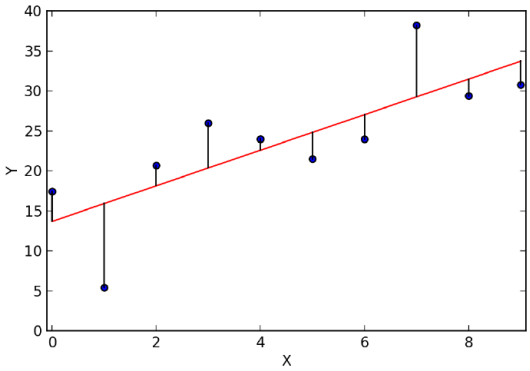
By analyzing learning curves, we can make informed decisions on how to adjust our model to achieve better generalization and improve overall performance.

→ **Goal**: Of course, the ultimate goal is for both $J_{train}(\theta)$ and $J_{cv}(\theta)$ to be low. Once this is achieved, the model should be tested on an unseen test set to evaluate its final performance.

6.2 Evaluation Metrics

In order to evaluate the performance of an ML system, different metrics can be used depending on the specific task. Specifically:

- **Evaluation metrics for regression**: For regression we make use of the **residuals**, which represent the difference between predicted values and actual values.



From these, we can adopt the following regression metrics:

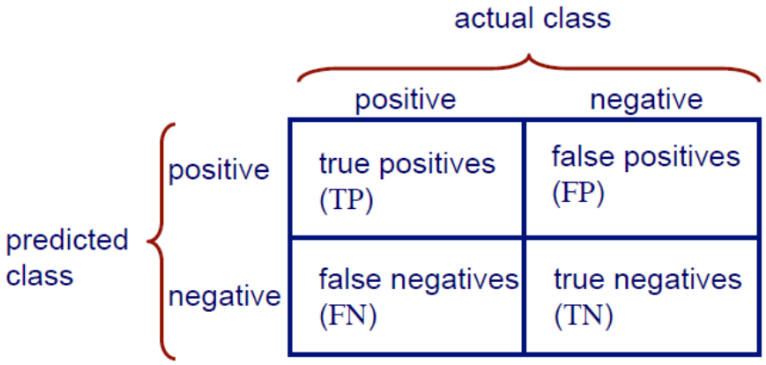
$$Mean\ Absolute\ Error\ (MAE) \qquad MAE = \frac{\sum_{i=1}^{m}|y_i^* - y_i|}{m}$$

$$Mean\ Squared\ Error\ (MSE) \qquad MSE = \frac{\sum_{i=1}^{m}(y_i^* - y_i)^2}{m}$$

$$Root\ Mean\ Squared\ Error\ (RMSE) \qquad RMSE = \sqrt{\frac{\sum_{i=1}^{m}(y_i^* - y_i)^2}{m}}$$

MSE and RMSE penalize large forecast errors compared to MAE. However, MSE is a differentiable function that simplifies the execution of mathematical operations compared to a non-differentiable function such as MAE, which is why it is often preferred to the latter.

- **Evaluation metrics for classification**: For classification we make use of a **confusion matrix**, a table which records the counts of correctly and incorrectly classified examples , dividing them into the following classes:
  - **TP (True Positive)**: Number of positive examples correctly predicted as positives
  - **FP (False Positive)**: Number of negative examples incorrectly predicted as positives
  - **FN (False Negative)**: Number of positive examples incorrectly predicted as negatives
  - **TN (True Negative)**: Number of negative examples correctly predicted as negatives



From this, several classification metrics are derived:

- **Accuracy**: ratio between "correctly classified predictions" on all predictions done. Indicates how accurate the prediction was.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision**: ratio between the "positive examples that were correctly classified as positive" on the "number of times positive values were predicted". Indicates the degree of accuracy in predicting positive examples.

$$Precision = \frac{TP}{TP + FP}$$

- **Recall** (also known as **Sensitivity** or **True Positive Rate**): ratio between the "positive examples that were correctly classified as positive" on "all truly positive samples". It indicates "how good you are" at predicting positive examples.

$$Recall = TPR = \frac{TP}{TP + FN}$$

- **Specificity** (also known as **True Negative Rate**): ratio between the "negative examples that were correctly classified as negative" on "all truly negative samples". Indicates the degree of accuracy in predicting negative examples.

$$Specificity = \frac{TN}{TN + FP}$$

- **Error rate**: The proportion of incorrect predictions (false positives + false negatives) out of all predictions. It's the complement of the accuracy.

$$Error\ rate = 1 - accuracy = \frac{FP + FN}{TP + TN + FP + FN}$$

- **F-measure** (also known as **F1-Score**): A harmonic mean of precision and recall. It provides a single measure that balances both concerns. Generally, it is adopted when you are interested in having both high precision and recall values at the same time.

$$F_{measure} = \frac{2 * precision * recall}{precision + recall}$$

- **False Positive Rate**: ratio between the "false positive predictions" on "all truly negative samples". It's the complement of specificity.

$$FPR = 1 - specificity = \frac{FP}{FP + TN}$$

Overview

Accuracy	(TP+TN)/(TP+TN+FP+FN)
Precision	TP/(TP+FP)
Recall or Sensitivity or True Positive Rate	TP/(TP+FN)
Specificity or True Negative Rate	TN/(FP+TN)
Error rate = 1- accuracy	(FP+FN)/(TP+TN+FP+FN)
F-measure	2*precision*recall/(precision+recall)
False Positive Rate=1-specificity	FP/(TN+FP)

The metrics inside the Table above are just the most important ones but there are plenty of them which are useful, depending on what we are focused on.

6.3 ROC & AUC

A **Receiver Operating Characteristic curve**, or **ROC curve**, is a graphical plot that illustrates the performance of a binary classifier model varying threshold values. The ROC curve plots the True Positive Rate (TPR) on y-axis against the False Positive Rate (FPR) on x-axis at each threshold, showing how effectively a model distinguishes between positive and negative classes. While commonly used for binary classification, ROC analysis can also be extended to multi-class problems.

Each point on the ROC curve corresponds to specific TPR and FPR obtained as result of a specific classification threshold. Plotting more of them we can see how TPR and FPR shift as the threshold changes.

- The best possible prediction model would yield a point in the upper left corner or coordinate **(0,1)** of the ROC space, representing 100% sensitivity (no false negatives) and 100% specificity (no false positives). This point is known as "**perfect classification**."

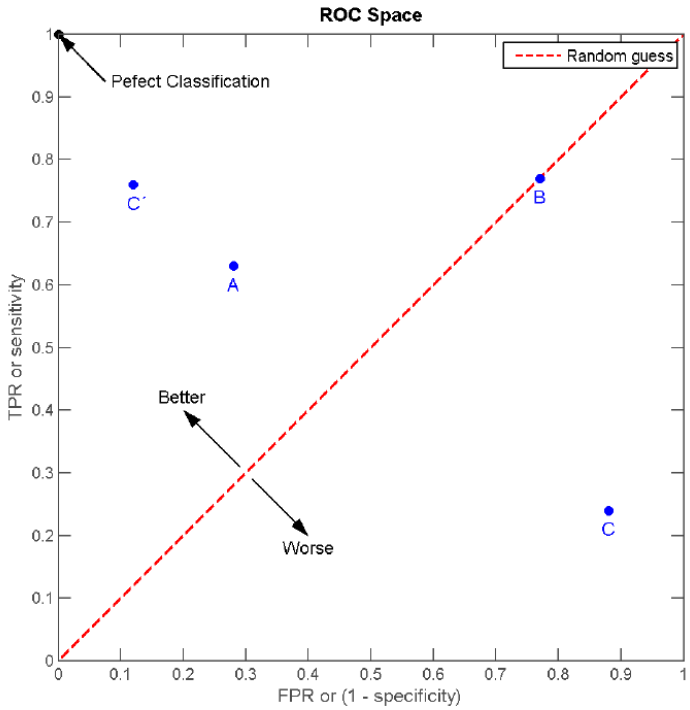
- The ROC curve also includes a **diagonal line** from the bottom left to the top right, called the "**line of no discrimination**", which represents random guessing. Points above the diagonal represent good classification results (better than random); points below the line represent bad results (worse than random).

Note: The ROC is also known as the Receiver Operating Characteristic curve, since it is a comparison between two operating characteristics (TPR and FPR) as the criterion (the threshold) changes.

To better illustrate this, consider four prediction results from 100 positive and 100 negative instances. We draw the confusion matrix and we calculate from it different metrics, in particular TPR and FPR.

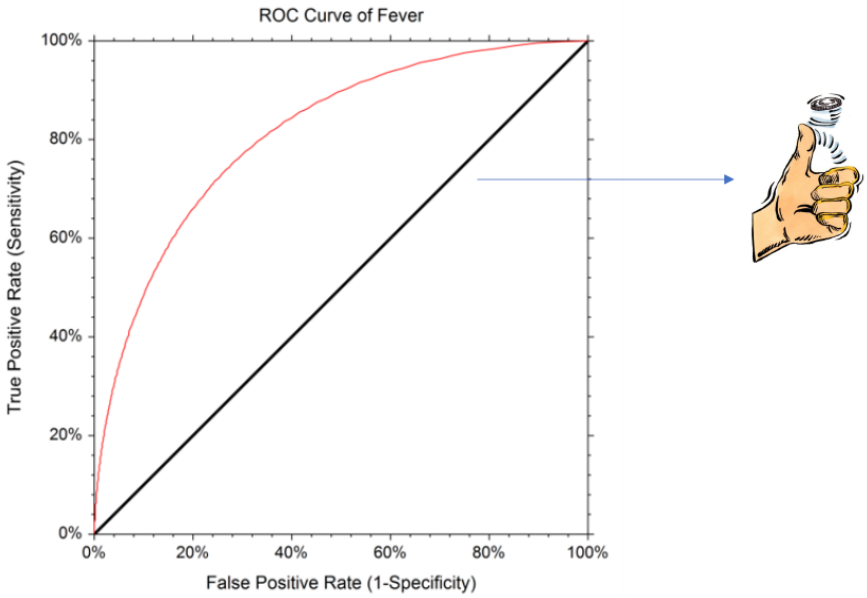
A			B			C			C′		
TP = 63	FN = 37	100	TP = 77	FN = 23	100	TP = 24	FN = 76	100	TP = 76	FN = 24	100
FP = 28	TN = 72	100	FP = 77	TN = 23	100	FP = 88	TN = 12	100	FP = 12	TN = 88	100
91	109	200	154	46	200	112	88	200	88	112	200
TPR = 0.63			TPR = 0.77			TPR = 0.24			TPR = 0.76		
FPR = 0.28			FPR = 0.77			FPR = 0.88			FPR = 0.12		
PPV = 0.69			PPV = 0.50			PPV = 0.21			PPV = 0.86		
F1 = 0.66			F1 = 0.61			F1 = 0.23			F1 = 0.81		
ACC = 0.68			ACC = 0.50			ACC = 0.18			ACC = 0.82		

Plots of the four results above in the ROC space are given in the following Figure. The result of classifier **A** clearly shows the best predictive power among **A**, **B**, and **C**. The result of **B** lies on the random guess line (the diagonal line), and it can be seen in the table that the accuracy of **B** is 50%. However, when **C** is mirrored (reversed) across the center point (0.5,0.5), the resulting classifier **C'** is even better than **A**. This mirrored classifier simply reverses the predictions of whatever classifier **C** predicts. Although the original **C** method has negative predictive power, simply reversing its decisions leads to a new predictive classifier **C'** which has positive predictive power.



The closer the results of TPR and FPR are to the upper left corner, the better it predicts, but the distance from the random guess line in either direction is the best indicator of how much predictive power a method has. If the result is below the line (i.e. the method is worse than a random guess), and in this case all of the method's predictions must be reversed in order to utilize its power, thereby moving the result above the random guess line.

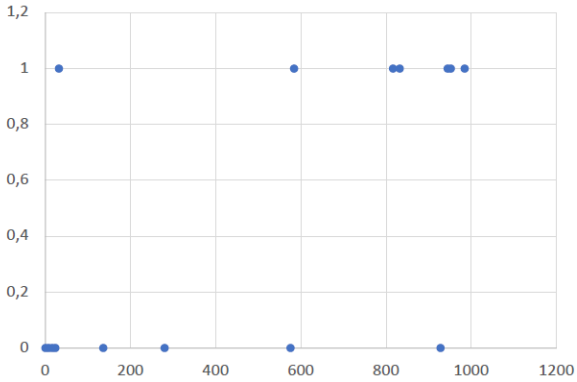
As the threshold for classification changes (from 0 to 1), each threshold produces a unique TPR and FPR, generating a series of points that form the ROC curve. By connecting these points, we create the ROC curve that shows the model performance across all thresholds.



To better see this, consider a scenario where we aim to predict if a year is “rainy” (1) or “non-rainy” (0) using forecast probabilities based on historical rainfall data from March to May in northeastern Brazil for the period 1981-1995. The data are arranged in order of decreasing probability.

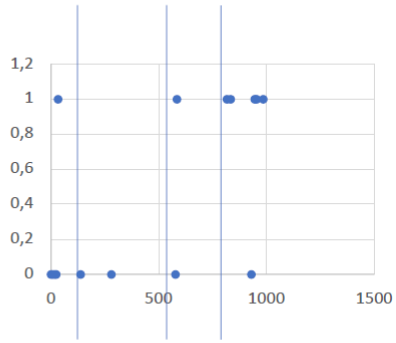
Year	Observed event (1) or non-event(0)	Forecast Probability (FP)
1994	1	0.984
1995	1	0.952
1984	1	0.944
1981	0	0.928
1985	1	0.832
1986	1	0.816
1988	1	0.584
1982	0	0.576
1991	0	0.28
1987	0	0.136
1989	1	0.032
1992	0	0.024
1990	0	0.016
1983	0	0.008
1993	0	0

Data describes March-May precipitation over North-East Brazil for 1981-1995
Arranged in decreasing probability

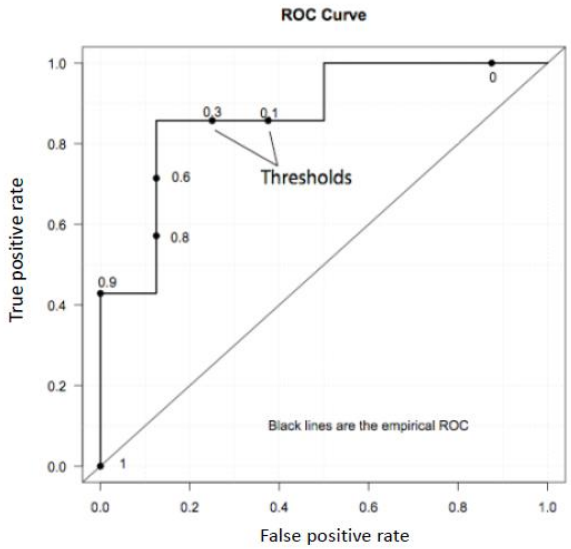


We chose three different thresholds $T = \{0.1, 0.5, 0.8\}$. If the forecast probability for a year exceeds a given threshold, we sign 1 for that threshold, 0 otherwise.

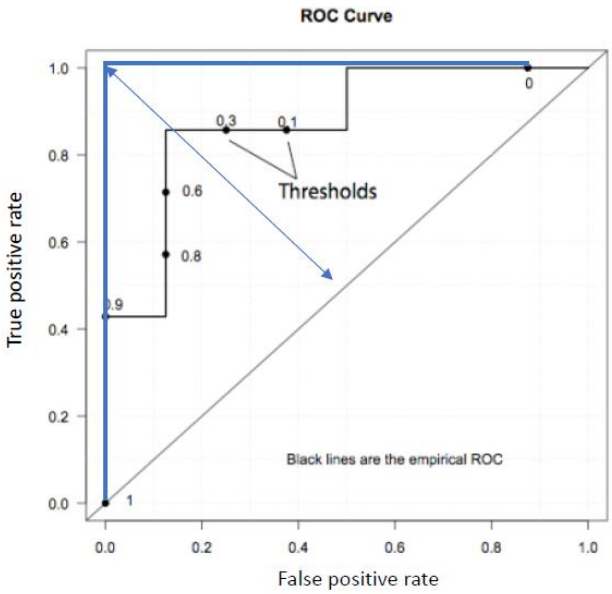
	Year	Observed event (1) or non-event(0)	Forecast Probability	T=0.1	T=0.5	T=0.8
Correct positive	1994	1	0.984	1	1	1
	1995	1	0.952	1	1	1
	1984	1	0.944	1	1	1
	1981	0	0.928	1	1	1
	1985	1	0.832	1	1	1
	1986	1	0.816	1	1	1
	1988	1	0.584	1	1	0
	1982	0	0.576	1	1	0
	1991	0	0.28	1	0	0
	1987	0	0.136	1	0	0
	1989	1	0.032	0	0	0
Correct negative	1992	0	0.024	0	0	0
	1990	0	0.016	0	0	0
	1983	0	0.008	0	0	0
	1993	0	0	0	0	0



Each threshold produces a different classifier configuration, yielding unique values of the True Positive Rate (TPR) and False Positive Rate (FPR) for each setting. These TPR and FPR pairs are then plotted as points on the ROC curve, with each point corresponding to a specific threshold. By connecting these points, we construct the ROC curve.



The best threshold is the one that positions the ROC curve nearest to (0,1), where both TPR is maximized, and FPR is minimized.



To evaluate the overall effectiveness of a classifier independently of specific threshold choices, it's useful to examine the Area Under the Curve (AUC) of the ROC curve.

AUC (Area Under the Curve) is a **single-value** metric derived from the ROC curve that summarizes the classifier's performance across all possible thresholds. AUC **ranges from 0 to 1**:

- **AUC = 1**: Represents a perfect classifier. In this case, the ROC curve would start at the origin (0,0), rise to the top-left corner (0,1) for ideal sensitivity, and extend horizontally to (1,1), covering the entire area beneath it.
- **AUC = 0.5**: Indicates performance no better than random guessing. In this case, the ROC curve would follow the diagonal line from (0,0) to (1,1).
- **AUC below 0.5**: Indicates that the classifier performs worse than random guessing.

It graphically represents the area under the ROC curve → **the larger the area under the ROC curve, the better the classifier's ability to distinguish between classes**, making AUC a **robust** measure of a model's discriminative power.

6.4 Paired t-Test

When comparing two classifiers, suppose we determine that classifier A performs "better" than classifier B based on certain evaluation metrics. However, we must consider whether this observed difference is statistically significant or simply due to random variation in our sample data. In other words, we want to know: *Are we confident that classifier A is genuinely better than B, or did we just happen to choose a sample of data where A performed better by chance?* A widely used statistical method to answer this question is the **paired sample t-test**.

The **paired sample *t*-test** is a statistical test used to compare two models on the same set of data, analyzing whether there is a significant difference in their performance. To do that it defines two competing hypotheses:

- **Null Hypothesis**: the performance differences between the two models are due to chance; so, under the null hypothesis, all observable differences are explained by random variations. → This is equivalent to saying that the two models have the same performance.

- **Alternative hypothesis**: the performance differences between the two models are genuine and not due to random chance. → This is equivalent to saying that one of the two models truly performs better than the other.

Let's see in detail how it works.

For instance, let's consider that the pairs of observations are accuracy values computed by the two binary classifiers a and b varying the threshold

$$\vec{y}^a = \{y_1^a, y_2^a, \dots, y_m^a\}$$

$$\vec{y}^b = \{y_1^b, y_2^b, \dots, y_m^b\}$$

Formally, we define the performance **difference vector** as :

$$\vec{\delta} = \{y_1^a - y_1^b, y_2^a - y_2^b, \dots, y_m^a - y_m^b\}$$

where each element, δ_i, is the difference in performance metric between the two models for a particular instance.

To determine the statistical significance of the observed difference, we perform the so-called **Hypothesis Test**. It is a test which:

- uses paired t-test to determine the probability p that the mean difference ($\bar{\delta}$) supports the null hypothesis. To do this:
    1. calculate the sample **mean**:

$$\bar{\delta} = \frac{1}{m}\sum_{i=1}^{m} \delta_i$$

    2. calculate the **t-statistic**:

$$t = \frac{\bar{\delta}}{\sqrt{\frac{1}{m(m-1)}\sum_{i=1}^{m}(\delta_i - \bar{\delta})^2}}$$

    Note: if you want to remember it:

$$t = \frac{\bar{\delta}}{\sqrt{\frac{Var(\delta)}{(m-1)}}}$$

    3. Determine the corresponding ***p–value***, by looking up t in a table of values for the **t-Student's distribution** with $m-1$ degrees of freedom. This p-value (also known as ***probability mass value***) indicates the probability that we would observe a difference as extreme as $\bar{\delta}$, assuming the null hypothesis is true.

- If p is sufficiently small ($\boldsymbol{p < 0.05}$ usually) then we reject the **null hypothesis**, concluding that the observed differences are statistically significant.

There are three variations of the paired t-test depending on the type of performance question being asked:

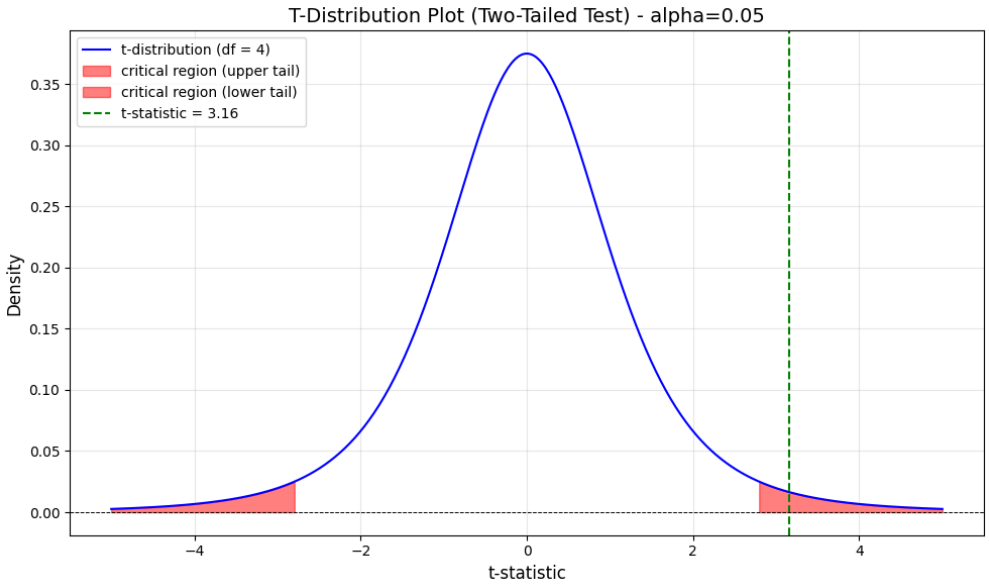
1. **Two-Tailed Test:** $\boldsymbol{p = 2 \cdot Pr(T > |t|)}$ → It answer to the question: "Is the accuracy of the two models different?". This test doesn't specify the direction (it doesn't tell us which of the two models is better).
2. **Upper-Tailed Test:** $\boldsymbol{p = Pr(T > t)}$ → It answer to the question "Is the first model better than the second one?"
3. **Lower-Tailed Test:** $\boldsymbol{p = Pr(T < t)}$ → It answer to the question "Is the second model better than the first one?"

For convenience, if you're unsure of directionality, you can first perform a **two-tailed test** in order to check if there is any difference in performance. Then if this fails to reject the null hypothesis (e.g. $p < 0.05$), then you can run one of the one-tailed test to ensure which of the two models is better than the other one.

The Figure below illustrates the **probability density function (pdf) of the t-Student distribution** (y-axis) **of the null hypothesis** (i.e. assuming the null hypothesis is true) against various **t-statistic** values (x-axis) for a Two-Tailed Test.

In this context, the **critical t-statistic** values mark the boundaries beyond which we would reject the null hypothesis, and these boundaries are determined by the chosen significance level (α) and the degrees of freedom in the data ($m - 1$). For a two-tailed test, the significance level is split equally between the two tails of the distribution, with each tail covering an area of $\alpha/2$ and the shaded red regions in the Figure represent **these critical regions**, one in each tail. For instance, if we choose $\alpha = 0.05$ the distribution has two critical regions of 0.025 each ($2 * 0.025 = 0.05$).

If the **p-value** is less than the significance level ($p < 0.05$) then the calculated t-statistic fell into one of the critical regions, meaning that we reject the null hypothesis in favor of the alternative → differences between the two models are genuine and not due to random chance.



6.5 Coefficient of Determination - R^2

The **coefficient of determination** (denoted as $\boldsymbol{R^2}$) is a measure of how well a regression model can **explain** the **variance** in the dependent variable (target) based on the independent variables (features). It is the proportion of total deviation explained by the regression model.

To compute R^2, we first need to define:

1. **Total Deviation**:

$$Det(T) = \sum_{i=1}^{m} \left(y^{(i)} - \bar{y}\right)^2$$

   This is the **total variance in the observed data**, so it captures the overall variability in the data. It is computed as the sum of squared distances between each observed value $y^{(i)}$ (ground truth) and the mean of observed values $\bar{y}$.

2. **Regression Deviation**:

$$Det(R) = \sum_{i=1}^{m} \left(y^{(i)*} - \bar{y}\right)^2$$

   This represents the **variance explained by the model**. It is computed as the sum of squared distances between each predicted value $y^{(i)*}$ and the mean of observed values $\bar{y}$.

3. **Residual Deviation**:

$$Det(E) = \sum_{i=1}^{m} \left(y^{(i)} - y^{(i)*}\right)^2$$

   This represents the **unexplained variance**. It is computed as the sum of the squared errors ($\left(y^{(i)} - y^{(i)*}\right)^2 = \left(e^{(i)}\right)^2$).

The formula for R^2 is:

$$\boldsymbol{R^2 = \frac{Det(R)}{Det(T)} = 1 - \frac{Det(E)}{Det(T)}}$$

which indicates the proportion of total variance explained by the model, which is also equal to one minus the proportion of total variance left unexplained by the model.

R^2 values range from 0 to 1:

- $R^2 = 1$: Perfect model fit, with all observed outcomes explained by the model.
- $R^2 = 0$: The model does not explain any of the variance in the observed data, performing no better than using the mean of observed values as a predictor.

→ The closer the value of R^2 to 1 the better the data fits the model.

So, depending on this value we can observe how well observed outcomes are replicated by the model, based on the proportion of total variation of outcomes explained by the model.

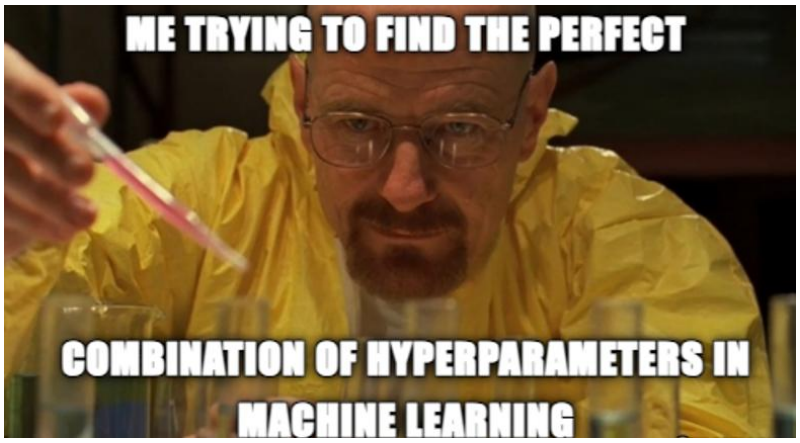
6.6 Hyperparameter Tuning

When building machine learning models, we often focus on optimizing the model's parameters (also called weights), which are learned directly from the data during training. However, models also have **hyperparameters**,

which as we already mentioned are parameters that are not learned from the data, but they are settled manually (fixed).

For instance, when working with a logistic regression with regularization you need to choose the type of regularization to apply, i.e. L1 or L2 and the value for the regularization parameter λ (e.g., 0.01, 0.05, etc.). On top of that, there are other hyperparameters to consider: for instance, the learning rate α, which controls the size of the steps taken in each iteration of the GD optimization process, or the tolerance for the stopping criteria, if you decide to use it. How do you find the right combination of these hyperparameters that will produce the best results for your model?

One might think a simple approach would be to try out a few different values for each parameter, keeping others fixed. For example, you could fix $\lambda = 1$, $\alpha = 0.01$, and try a few different tolerance values. Then, you could try $\lambda = 0.1$, $\alpha = 0.05$, and so on, manually experimenting with different settings.



However, this process is inefficient and, most importantly, it doesn't guarantee that you'll find the best combination of hyperparameters. This is where hyperparameter tuning comes in.

Hyperparameter tuning is the process of systematically searching for the best combination of hyperparameters that leads to best model performance.

One widely used method for hyperparameter tuning is **Grid Search**. This technique involves defining a grid (or range) of hyperparameters and exhaustively searching through all possible combinations within that grid. For instance, suppose we are tuning two hyperparameters: the regularization strength λ and the learning rate α. We might decide that λ can take values from the set $\{0.1,1,10\}$ and α can take values from $\{0.01, 0.1, 1\}$. Following the grid search method, we would train and evaluate the model using each of the 9 possible combinations of these hyperparameters (i.e., $\lambda = 0.1$ and $\alpha = 0.01$, $\lambda = 0.1$ and $\alpha = 0.1$, and so on). This ensures that every possible combination is explored. The combination that results in the best model performance, based on some evaluation metric (e.g. accuracy), is chosen.

Grid Search is generally performed in combination with **k-fold cross-validation**. This ensures that the evaluation of each hyperparameter combination is robust, ensuring that the model isn't overfitting to one particular train-test split.

However, Grid Search has a key limitation: it's very computationally expensive when the number of hyperparameters increases. For example, imagine tuning a model with 4 hyperparameters, each with 6 possible values. This results in $6 * 6 * 6 * 6 = 6^4 = 1296$ possible combinations to evaluate, which could be prohibitively slow. Moreover, k-fold cross-validation significantly increases this computational burden. In fact, you should repeat this for each of the 5 folds, this means you will need to evaluate $6^4 * 5 = 6480$ different model fits. As the number of hyperparameters or the size of the grid grows, this process can become extremely time-consuming.

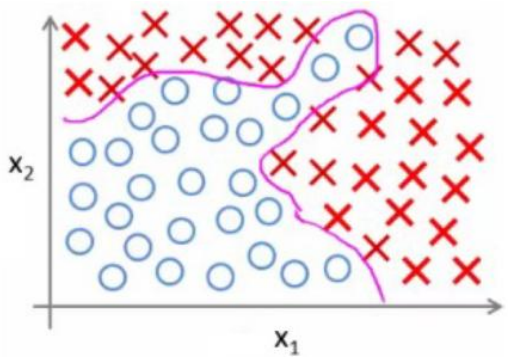
Although Grid Search is widely used, there are other approaches that can be more efficient in some cases. One such method is **Random Search**, which randomly samples combinations of hyperparameters rather than exhaustively searching through every possible combination. This can be more efficient in practice, as it may find good solutions faster than grid search, however it may miss the optimal combination.

A more advanced method is **Bayesian Optimization** which uses a probabilistic model to predict which hyperparameters are likely to result in better performance. It then focuses on exploring those promising regions of the hyperparameter space that are likely to yield better performance. This approach is particularly useful for high-dimensional or continuous hyperparameter spaces.

7 Neural Networks

7.1 Non-Linear Hypothesis

When dealing with complex datasets containing numerous features, performing linear or logistic regression becomes increasingly unwieldy. For example, consider a supervised learning classification problem. If we attempt to use logistic regression, we might try to incorporate non-linear polynomial features, such as quadratic or cubic terms, to capture more complexity in the data. Suppose we define a hypothesis function using the sigmoid function with various polynomial terms like:



$$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^3 x_2 + \theta_6 x_1 x_2^2 + \theta_7 x_1 x_2^3 + \cdots)$$

While this approach can work well when we have only a couple of features, such as x_1 and x_2, it quickly becomes impractical for higher numbers of features. That's an important problem since most cool machine learning problems have way more features.

For instance, let's come back to the housing example, so suppose we have 100 features ($x_1 = size$, $x_2 = \#bedrooms$, $x_3 = \#floors$, $x_4 = age$, …, x_{100}) and we want to predict whether a house will sell within the next 6 months. Say we want to include as non-linear features all quadratic (second order) terms, so we will have

$$x_1^2, x_2^2, x_3^2, \dots,$$

$$x_1 x_2, x_1 x_3, x_1 x_4, \dots., x_1 x_{100}$$

just including them our feature count would balloon around 5,000.

Note: The exact way to calculate how many features for all polynomial terms is the combination function with repetition: $\frac{(n+r-1)!}{r!(n-1)!}$. For example, for our 100 features by considering the quadratic terms we end-up with $\frac{(100+2-1)!}{2!(100-1)!} = 5050.$

The number of features considering all quadratic terms would grow at $O(n^2)$. Moreover, if we include cubic terms, we will have

$$x_1^3, x_2^3, x_3^3, \dots,$$

$$x_1^2, x_2^2, x_3^2, \dots,$$

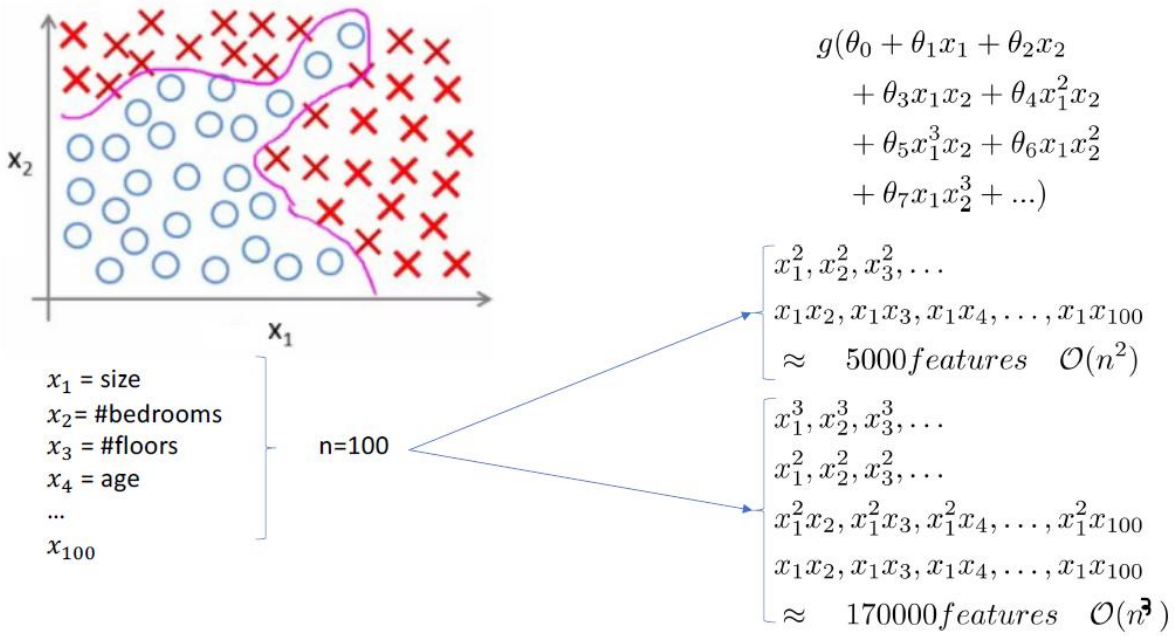
$$x_1^2 x_2, x_1^2 x_3, x_1^2 x_4, \dots., x_1^2 x_{100},$$

$$x_1 x_2, x_1 x_3, x_1 x_4, \dots., x_1 x_{100}$$

and the feature count would grow to around 170,000. In this case, there are even more features that would grow asymptotically at $O(n^3)$.

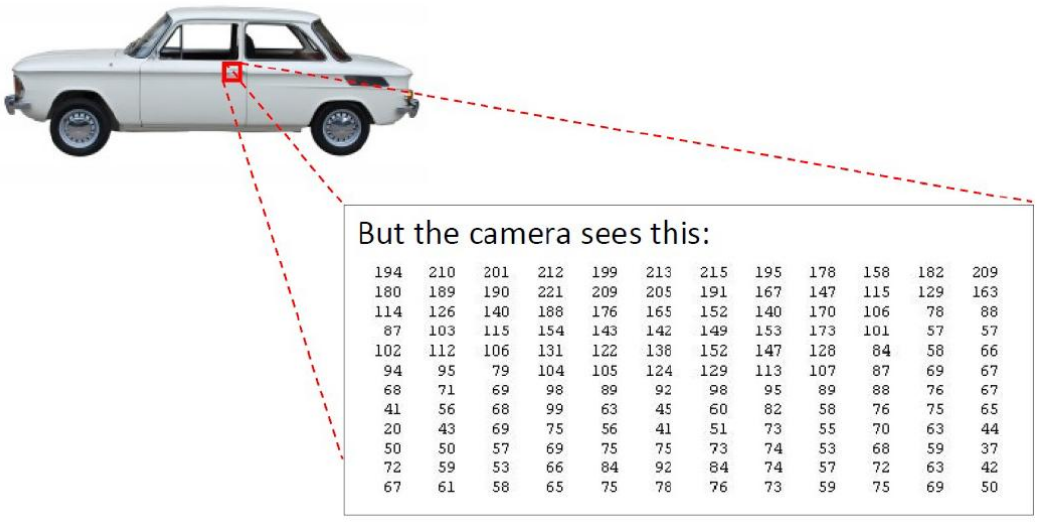
This quadratic or cubic feature growth leads to an explosion in feature count, making computation and memory requirements extremely high. As a result, while adding polynomial terms allows for modeling more complex relationships, it quickly becomes impractical for large-scale problems.

Someone could think, as a way around, to include only a subset of features. However, if you don't have enough features, often a model won't let you fit a complex dataset.



In fields like computer vision, where feature counts can be extremely high, this problem becomes even more pronounced. Imagine you want to train a machine learning classifier to identify whether an image contains a car.

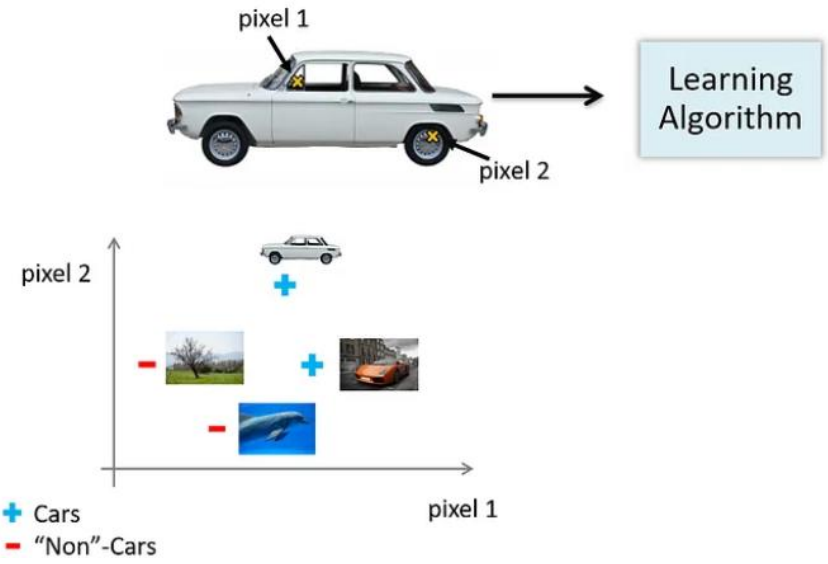
Some folks might wonder why this is even a challenge since when we look at a picture, it's pretty clear what we're looking at. For humans, recognizing a car in a picture is straightforward. However, for a computer, an image is a grid of pixel intensity values (as shown in the Figure below), where each pixel represents brightness or color intensity rather than an object or pattern. For instance, a door handle might be represented by a specific set of intensity values within this pixel grid. Thus, the computer vision problem is to analyze these pixel values and identify that they correspond to a car or part of it.



To train a machine learning model to recognize cars, we first create a labeled dataset with images of cars and non-cars. Then we feed these data into the learning algorithm to create a classifier. After that, we test it out by showing it a new image and asking if it can identify if it's a car or not. Hopefully, it'll be able to recognize that it's a car.

One approach to do this is to select specific pixel locations within each image as features and then plot them. For example, if we choose two pixel locations (say, pixel 1 and pixel 2) and analyze their intensity values across different images, we might observe patterns: cars and non-cars may appear in distinct regions when plotted based

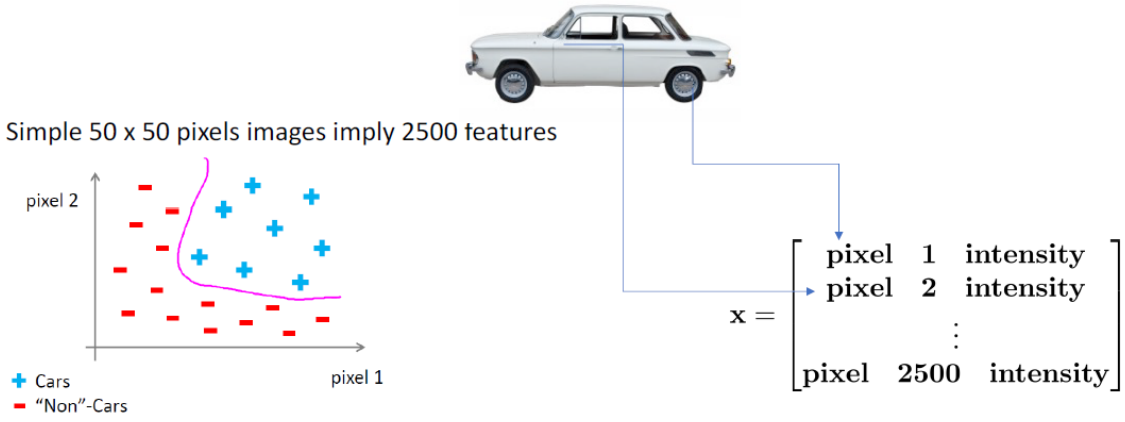
on these pixel values. If we plot several examples with cars marked as "+" and non-cars as "-", we may notice clusters of "+" and "-" symbols separating naturally in this 2D space.



However, as more images are analyzed, we find that a simple linear boundary might not separate the car and non-car examples adequately. Instead, a **non-linear hypothesis** is necessary to accurately distinguish between the two classes, as cars and non-cars occupy complex regions within the feature space.

What is the dimension of the feature space? Suppose we use just small 50×50 images, in grayscale this will lead to 2500 pixels, so $n = 2500$. For an RGB image, this increases to $n = 7500$ (2500 pixels $*$ 3 color channels).

If we try to capture a nonlinear hypothesis by including all quadratic features, we would end up with around six million features—a size far too large to be practical. This makes traditional methods, like logistic regression, ineffective for learning complex nonlinear patterns when n is large.

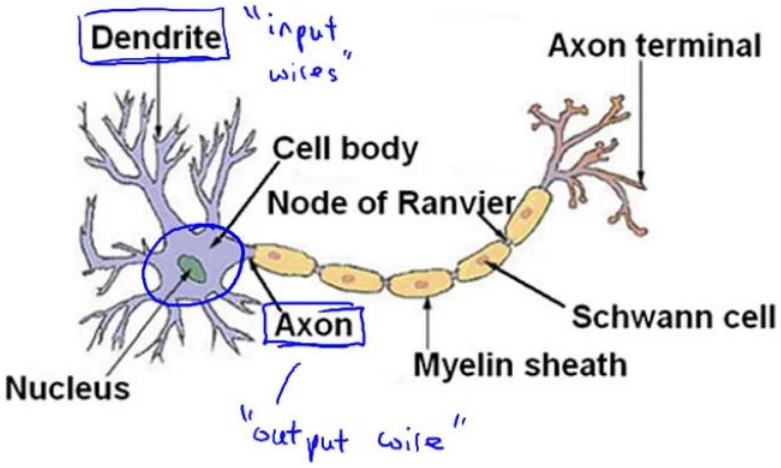


This is where **neural networks** shine. They are far more efficient at learning complex nonlinear hypotheses, even with high-dimensional feature spaces.

Neural networks, originated as algorithms designed to replicate the workings of the human brain, became popular in the 1980s and 1990s, but for various reasons their popularity diminished in the late 90s. In recent years, however, neural networks have experienced a major resurgence. One key reason is that neural networks are computationally demanding, and only with the advancement of computing power has it become feasible to run large-scale networks effectively. This, along with other technical innovations, has made neural networks the go-to approach in many state-of-the-art applications.

7.2 Inspiration from Biological Neuron

The architecture of neural networks is inspired by the structure of **neurons** in the brain, which are specialized cells that handle complex tasks. Each neuron in our brain has a **cell body** and multiple input connections, called **dendrites**, that receive signals from other neurons. These signals are then processed within the neuron, and any output is transmitted through a single output connection, the **axon**, which can send signals to other neurons.



In the brain, neurons communicate through tiny electrical impulses known as "**spikes**". When one neuron decides to send a message, it transmits a small electrical pulse through its axon to connect with the dendrite of another neuron. This next neuron then performs its own processing based on the inputs it receives, potentially sending a new signal to additional neurons.

This intricate process of message-passing underpins all thought, perception, and physical movement. For example, when you move a muscle, a neuron sends electrical pulses to contract it. Similarly, sensory organs like the eyes send signals to the brain through these neural pathways.

This communication process allows the brain to perform a wide variety of tasks, from vision to movement, with a unified underlying mechanism.

The "One Learning Algorithm" Hypothesis

It seems like if you want to mimic the brain you have to write lots of different pieces of software to mimic all of these different fascinating things that the brain can do. However, there is a fascinating hypothesis called the **"one learning algorithm" hypothesis** that suggests that, instead of relying on many separate processes (like a thousand different programs), the brain may function using a single, flexible learning algorithm capable of handling any type of input, regardless of its origin.

Evidence supporting this theory comes from a series of neuro-rewiring experiments. For example, let's consider the *auditory cortex*. Normally, when you hear sounds, your ears send signals to this part of the brain, enabling you to understand words. However, neuroscientists have performed experiments where they re-route input from an animal's optic nerve (normally responsible for vision) to the auditory cortex. Amazingly, this part of the brain adapts to "see." These animals can distinguish visual patterns and make decisions based on what they "see" through this rewired auditory cortex.

Another fascinating example involves the *somatosensory cortex*, responsible for processing touch. In similar rewiring experiments, when visual inputs are directed to this region, it can adapt to process sight, just as it would

process touch. This flexible ability to process different types of sensory information with the same brain tissue, regardless of its original function, is central to what scientists call “neuro-rewiring.”

These findings support the idea that if any part of the brain can potentially process any type of sensory input, perhaps the brain doesn’t need a multitude of specialized algorithms. Instead, it might rely on a single, adaptable learning algorithm that learns how to interpret whatever data it receives. The implications of this are profound: rather than needing unique programs or algorithms for every different cognitive or sensory function, it might be possible to approximate the brain’s general learning algorithm and implement it in AI. If we could replicate this “one learning algorithm” on a computer, we could take a significant step forward in AI development, potentially enabling machines to learn as flexibly and broadly as the human brain.

7.3 Neuron Model: Logistic Unit

In an artificial neural network, we’re going to use a very simple model of what a neuron does. We’re going to model a **neuron** (also known as **unit**) as just a **logistic unit**:

- This unit receives input features through "input wires" (analogous to dendrites in biological neurons).
- The logistic unit itself performs a computation (that can be compared to the function of a neuron’s cell body).
- then transmits its result through "output wires" (similar to an axon).

The computation within this logistic unit is just like our previous logistic regression hypothesis calculation:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

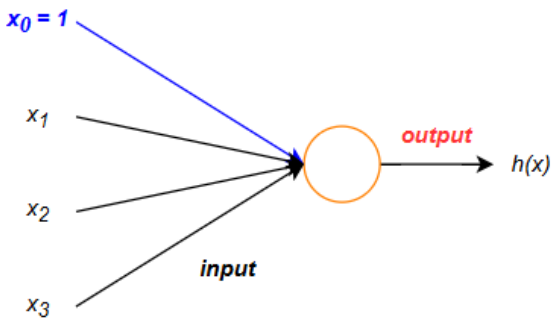
So basically, what we do is perform the weighted sum $\theta^T x$ and then apply a sigmoid (logistic) **activation** function to it $g(\theta^T x)$.

Recall that sigmoid function is defined as:

$$g(z) = \frac{1}{1 + e^{-z}}$$

The term "**activation** function" in neural networks refers to this **non-linear transformation** that occurs at each neuron.

Visually, a simplistic representation looks like:



Notes: In this model our x_0 input node is sometimes called the "**bias input**”, and it is always equal to 1. The parameter associated to it (θ_0) is also called the “**bias parameter**”. All the other parameters of θ (except the bias parameter θ_0) are instead called "**weights**" in neural networks (it will be clearer later why we want to underline this differentiation).

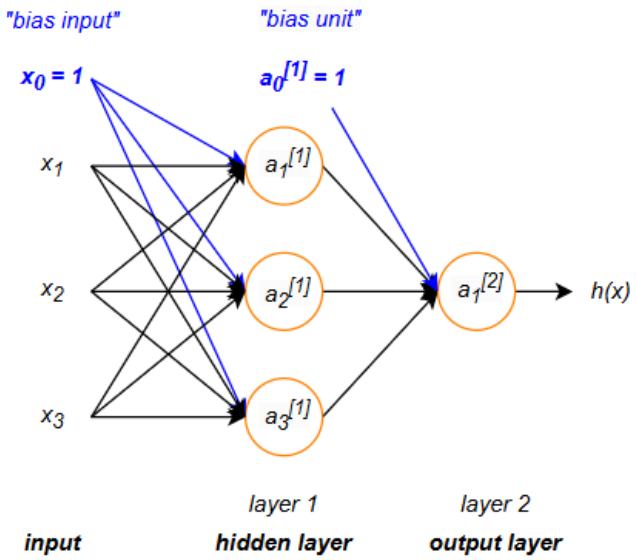
So, the tiny diagram above shows just one neuron. A neural network is essentially a collection of these neurons working together, with each neuron in one layer connected to neurons in the next layer.

More specifically, we can distinguish two types of layers in a neural network:

- **Hidden Layer**: Any intermediate layer that processes inputs. These layers are also called "hidden layers" because their values are not directly visible in the data: In supervised learning, we only observe the input and output layers, therefore any internal node (not in x or y) is considered hidden.
- **Output Layer**: The final layer that produces the network's output, either a prediction or classification, as the result of the hypothesis function. The final layer is also called the output layer because that layer has the neuron that outputs the final value computed by a hypothesis.

! **Note**: The **Input Layer doesn't exist,** and the input features are not neurons (they don't compute any calculus).

For example, consider a simple network with a single hidden layer with three neurons and an output layer with one neuron. Three input features are passed in input to the network, so they flow first trough the hidden layer and then to the output layer which at the end computes the final value of the hypothesis. You can notice that each neuron receive all the input values from the previous layer and a neural network that does so is called **fully connected**.



While this example focuses on a network with just a single hidden layer, we will later explore neural networks with multiple hidden layers.

Before going ahead in the calculus, let's introduce some notation. We denote by:

- x_j = the j-th input feature
  - For example, x_1 is the first feature of the input x
- $z_j^{[l]}$ = the weighted sum computed in a given unit j in layer l (sometimes also referred to as pre-activation)
  - For example, $z_1^{(2)}$ is the weighted sum computed by the first unit in the second layer
- $a_j^{[l]}$ = the "activation" of unit j in layer l

- For example, $a_1^{(2)}$ is the activation of the first unit in the second layer
- By activation, we mean the value which is computed and **outputted** by that node after applying the activation function

- $\Theta^{[l]}$ = the **matrix of weights** controlling function mapping from layer $l-1$ to layer l. It is constructed as the concatenation of all the weights vectors $\theta_j^{(l)}$ in the following way:

$$\Theta^{[l]} = \begin{bmatrix} - & \left(\theta_1^{[l]}\right)^T & - \\ - & \left(\theta_2^{[l]}\right)^T & - \\ & \vdots & \\ - & \left(\theta_d^{[l]}\right)^T & - \end{bmatrix}$$

  where d is the number of units in the layer l.

  Note: In some texts $\Theta^{[l]}$ is also indicated as $W^{[l]}$.

- $b^{[l]}$ = the **bias vector**, i.e the vector containing all the bias parameters for layer l:

$$b^{[l]} = \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_d^{[l]} \end{bmatrix}$$

Note: Here we separated the weights $\Theta^{[l]}$ and the biases $b^{[l]}$ into distinct components for the following reasons:

- It allows us to operate directly on vectors for more efficient computations.
- Regularization is applied only to the weights $\Theta^{[l]}$, not to the biases $b^{[l]}$. By separating them, regularization can be applied directly to the weight matrix without needing to iterate through the parameter matrix and exclude bias terms manually.

Following this notation instead of referring to $\theta_{10}^{[l]}$ we denote it as $b_1^{[l]}$. Similarly, $\theta_{20}^{[l]}$ becomes $b_2^{[l]}$, and so on.

Moreover, using this notation, θ corresponds to θ^T. Therefore, the computation:

$$h_\theta(x) = g(\theta^T x)$$

(where in this case x includes the bias input $x_0 = 1$) becomes:

$$h_\theta(x) = g(\Theta x + b)$$

(where in this case x **NOT** includes the bias input $x_0 = 1$)

Said that let's come back to calculus and see how they are performed for the network we saw above.

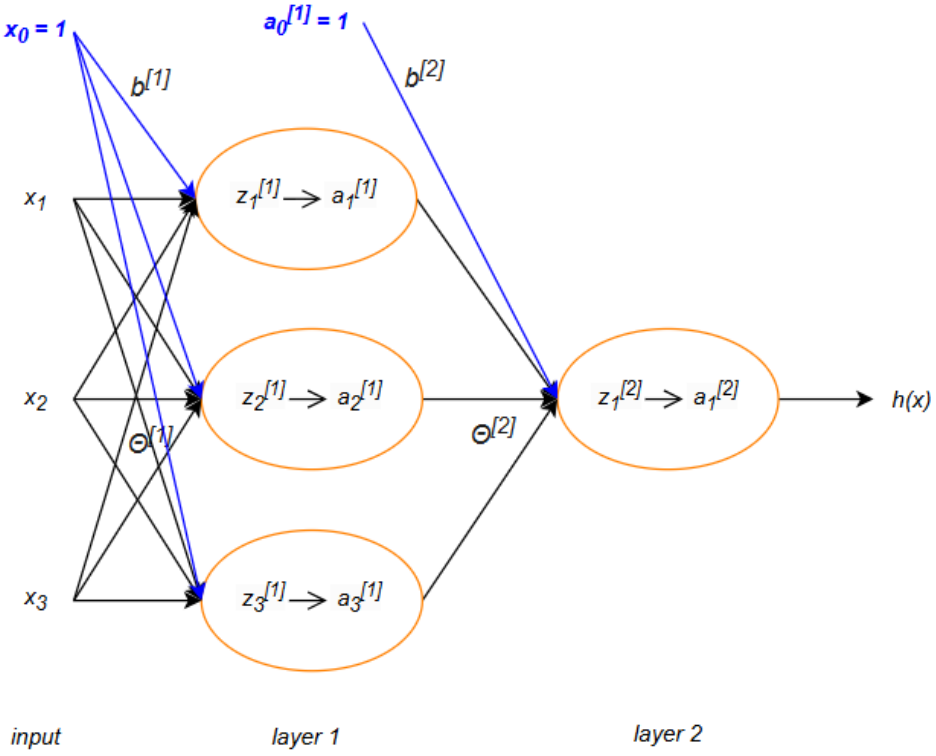
In this case we will have:

$$\Theta^{[1]} = \begin{bmatrix} \theta_{11}^{[1]} & \theta_{12}^{[1]} & \theta_{13}^{[1]} \\ \theta_{21}^{[1]} & \theta_{22}^{[1]} & \theta_{23}^{[1]} \\ \theta_{31}^{[1]} & \theta_{32}^{[1]} & \theta_{33}^{[1]} \end{bmatrix}$$ mapping from input features to layer 1

$$b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix}$$

$\theta^{[2]} = \begin{bmatrix} \theta_{11}^{[2]} & \theta_{12}^{[2]} & \theta_{13}^{[2]} \end{bmatrix}$ mapping from layer 1 to layer 2

$$b^{[2]} = \begin{bmatrix} b_1^{[2]} \end{bmatrix}$$



The values for each "activation" node in layer 1 are obtained as follows:

$$a_1^{[1]} = g\left(z_1^{[1]}\right) = g\left(b_1^{[1]}x_0 + \theta_{11}^{[1]}x_1 + \theta_{12}^{[1]}x_2 + \theta_{13}^{[1]}x_3\right)$$

$$a_2^{[1]} = g\left(z_2^{[1]}\right) = g\left(b_2^{[1]}x_0 + \theta_{21}^{[1]}x_1 + \theta_{22}^{[1]}x_2 + \theta_{23}^{[1]}x_3\right)$$

$$a_3^{[1]} = g\left(z_3^{[1]}\right) = g\left(b_3^{[1]}x_0 + \theta_{31}^{[1]}x_1 + \theta_{32}^{[1]}x_2 + \theta_{33}^{[1]}x_3\right)$$

and since our hypothesis is situated at the layer 2, it is equivalent to computing the activation $a_1^{(2)}$:

$$h_\theta(x) = a_1^{[2]} = g\left(z_1^{[2]}\right) = g\left(b_1^{[2]}a_0^{[1]} + \theta_{11}^{[2]}a_1^{[1]} + \theta_{12}^{[2]}a_2^{[1]} + \theta_{13}^{[2]}a_3^{[1]}\right)$$

Note: Following this notation we don't need to add $x_0 = 1$ or $a_0^{[l]} = 1$ for each layer.

The dimensions of each weight matrix depend on the number of units in adjacent layers and is determined as follows:

➔ **If network has d units in layer $l-1$ that we denote with d_{l-1} and has s units in layer l that we denote with s_l, then $\theta^{[l]}$ will be of dimension $s_l \times d_{l-1}$, while $b^{[l]}$ will be of dimension $s_l \times 1$.**

Examples:

- If we have 3 input features and layer 1 has 3 units, then $d_0 = 3$ and $s_1 = 3$ ➔ $\theta^{[1]}$ will be a 3x3 matrix and $b^{[1]}$ will be a 3x1 column vector.
- If layer 1 has 3 units and layer 2 has 1 unit, then $d_1 = 3$ and $s_2 = 1$ ➔ $\theta^{[2]}$ will be a 1x3 vector and $b^{[2]}$ will be a 1x1 column vector.

7.4 Forward Propagation: Vectorized Implementation

The most intuitive way to implement the calculations we've discussed is through the use of **for loops**. However, when dealing with high-dimensional inputs and hidden layers, this approach is computationally inefficient and can significantly slow down the execution of the code.

To address this challenge, we use **vectorization**, which leverages matrix algebra and optimized numerical linear algebra libraries to achieve faster computations. Vectorization eliminates explicit loops by operating on entire vectors or matrices, enabling the neural network computations to run efficiently.

Let's now examine this process in a vectorized form:

For layer 1 we have:

$$\underbrace{\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ \vdots \\ z_d^{[1]} \end{bmatrix}}_{z^{[1]} \in \mathbb{R}^{d \times 1}} = \underbrace{\begin{bmatrix} \theta_{11}^{[1]} & \dots & \theta_{1n}^{[1]} \\ \theta_{21}^{[1]} & \dots & \theta_{2n}^{[1]} \\ & \vdots & \\ \theta_{d1}^{[1]} & \dots & \theta_{dn}^{[1]} \end{bmatrix}}_{\theta^{[1]} \in \mathbb{R}^{d \times n}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}}_{x \in \mathbb{R}^{n \times 1}} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_d^{[1]} \end{bmatrix}}_{b^{[1]} \in \mathbb{R}^{d \times 1}}$$

$$\underbrace{\begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ \vdots \\ a_d^{[1]} \end{bmatrix}}_{a^{[1]} \in \mathbb{R}^{d \times 1}} = g\left(\underbrace{\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ \vdots \\ z_d^{[1]} \end{bmatrix}}_{z^{[1]} \in \mathbb{R}^{d \times 1}} \right)$$

For layer 2 we have:

$$\underbrace{\begin{bmatrix} z_1^{[2]} \\ z_2^{[2]} \\ \vdots \\ z_s^{[2]} \end{bmatrix}}_{z^{[2]} \in \mathbb{R}^{s \times 1}} = \underbrace{\begin{bmatrix} \theta_{11}^{[2]} & \dots & \theta_{1n}^{[2]} \\ \theta_{21}^{[2]} & \dots & \theta_{2n}^{[2]} \\ & \vdots & \\ \theta_{s1}^{[2]} & \dots & \theta_{sd}^{[2]} \end{bmatrix}}_{\theta^{[2]} \in \mathbb{R}^{s \times d}} \underbrace{\begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ \vdots \\ a_d^{[1]} \end{bmatrix}}_{a^{[1]} \in \mathbb{R}^{d \times 1}} + \underbrace{\begin{bmatrix} b_1^{[2]} \\ b_2^{[2]} \\ \vdots \\ b_s^{[2]} \end{bmatrix}}_{b^{[2]} \in \mathbb{R}^{s \times 1}}$$

$$\underbrace{\begin{bmatrix} a_1^{[2]} \\ a_2^{[2]} \\ \vdots \\ a_s^{[2]} \end{bmatrix}}_{a^{[2]} \in \mathbb{R}^{s \times 1}} = g\left(\underbrace{\begin{bmatrix} z_1^{[2]} \\ z_2^{[2]} \\ \vdots \\ z_s^{[2]} \end{bmatrix}}_{z^{[2]} \in \mathbb{R}^{s \times 1}} \right)$$

This process repeats for all additional layers.

Note that:

- The activation function g is applied **element-wise** to the vector $z^{[l]}$, resulting in a vector $a^{[l]}$ of the same dimensions of $z^{[l]}$.
- For simplicity, we can define the input x as the "activation of the 0-th layer": $\boldsymbol{x} = \boldsymbol{a}^{[0]}$.

Thus, said that, the forward propagation equations for the previous example can be written compactly as:

$$z^{[1]} = \Theta^{[1]} a^{[0]} + b^{[1]}$$

$$a^{[1]} = g\left(z^{[1]}\right)$$

$$z^{[2]} = \Theta^{[2]} a^{[1]} + b^{[2]}$$

$$h_\Theta(x) = a^{[2]} = g\left(z^{[2]}\right)$$

Generalized Process

This approach generalizes to any layer l in the network:

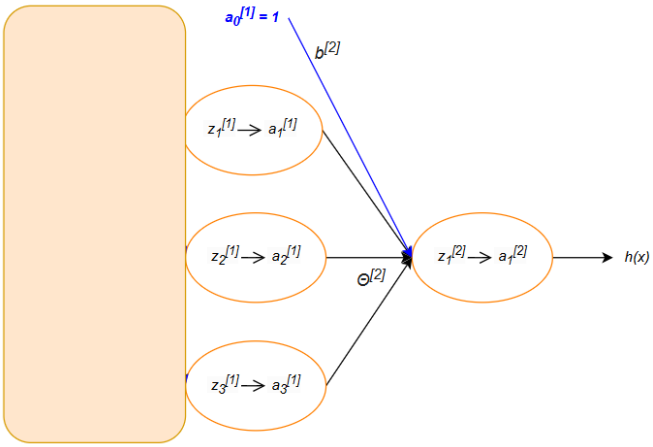
1. Compute $z^{[l]}$ by using the parameters of layer l and the activations from the previous layer $l-1$:

$$z^{[l]} = \Theta^{[l]} a^{[l-1]} + b^{[l]}$$

2. Calculate activations for the layer as: $a^{[l]} = g\left(z^{[l]}\right)$

This entire process, known as **forward propagation**, allows activations to flow from the input through hidden layers to the output layer, resulting in the final hypothesis calculation (we start with the activations of the input layer and propagate them forward through each hidden layer sequentially to compute the final output). In particular, what we saw is the **vectorized** implementation of this procedure, which is the one actually adopted in real scenarios since vector operations enable faster computation.

This view of forward propagation also gives us insight into why they are powerful tools for learning complex, nonlinear relationships. In fact, the real strength of neural networks lies in their ability to learn increasingly abstract and intricate features across multiple layers → In a network with multiple layers, each layer builds upon the representations learned by the prior layer, ultimately enabling the network to model highly nonlinear functions.



The neural network essentially learns its own features, rather than being limited to feeding raw input features like directly into the last part of the network, that is basically a **logistic regression** (see Figure on the right).

This structure allows the network to discover intricate and complex features through the parameters $\Theta^{[1]}$, $\Theta^{[2]}$ and so on. As a result, it can produce a more accurate hypothesis than if it were restricted to using only the raw features or predefined polynomial terms.

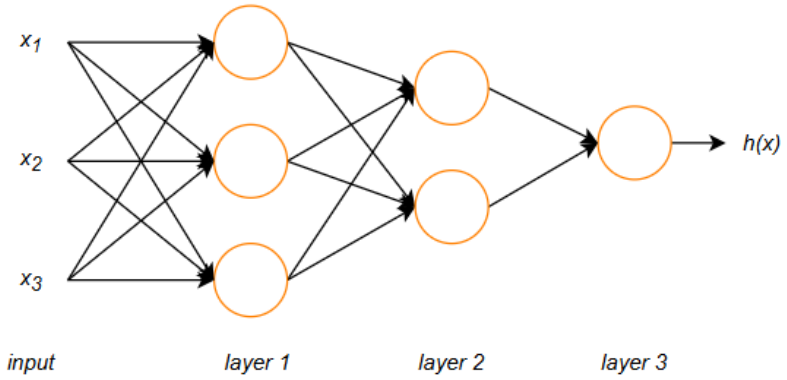
Neural network gives us the **flexibility** to learn whatever features it wants to feed into the final logistic regression calculation:

➔ So, if we compare this to previous logistic regression, you would have to calculate your own exciting features to define the best way to classify or describe something. Here, we're letting the hidden layers do that, so we feed the hidden layers our input values, and let them learn whatever gives the best final result to feed into the final output layer. We will add additional insight about this concept at the end of this chapter.

Beyond the structure we've examined, other network architectures are possible. We can adjust the **topology** of a neural network by:

- Adding more layers (depth),
- Varying the number of units in each layer (width).

For instance, a network might have three hidden units in layer 1 and two hidden units in layer 2, resulting in a very interesting nonlinear hypothesis in the output layer.



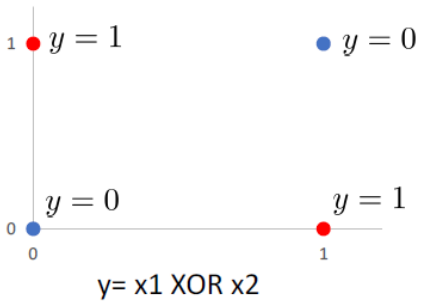
7.5 Non-Linear Classification Example: XOR

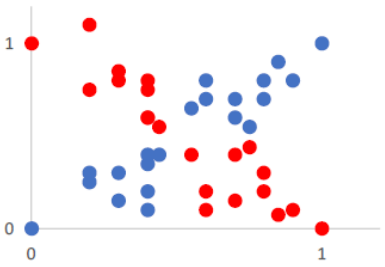
Consider a classification problem with two binary input features, x_1 and x_2 (so they can assume 0 or 1). For this scenario, imagine positive examples (labeled as 1) grouped in specific areas, like the upper right and lower left, and negative examples (labeled as 0) fill the remaining spaces (as shown in Figure below). Here, our task is to learn a nonlinear decision boundary that separates positive from negative examples.

Standard linear classifiers are not able to solve this problem due to non-linearity.

By the way we can notice that this setup follows an XOR pattern, where the target label, y, is 1 only when exactly one of x_1 or x_2 is 1, otherwise it's 0. Idea: → fit a neural network to this XOR function.

Since we know that the XOR operator is nothing more than a combination of three simpler operators AND, OR and NOT, to easily solve it we can break it in these different components. Each of them can be modeled as a "neural network" composed of a single neuron by appropriately weighting the two inputs (plus bias). By combining these individual neurons, we can at the end construct a neural network capable of performing the XOR function.





AND function

To compute $x_1\ AND\ x_2$, the target y should be 1 only if both x_1 and x_2 are 1. We can create a neural network with a single unit to approximate this behavior. We assign -30 for the bias parameter, and $+20$ for both weights associated to x_1 and x_2. Mathematically, this forms the hypothesis:

$$h(x) = g(-30 + 20x_1 + 20x_2)$$

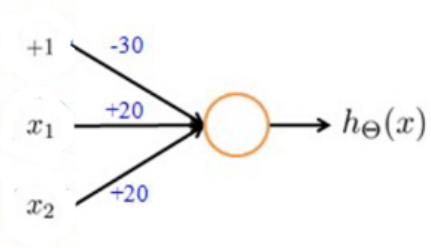
where g is the sigmoid activation function.

Let's see how this network behaves for all possible combinations of x_1 and x_2:

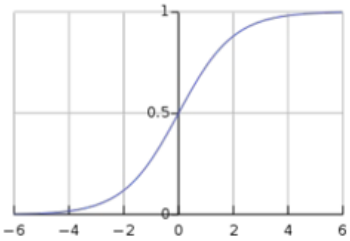
- If $x_1 = 0$ and $x_2 = 0, h(x) = g(-30)$, which is very close to 0.
- If $x_1 = 0$ and $x_2 = 1, h(x) = g(-10)$, also close to 0.
- If $x_1 = 1$ and $x_2 = 0, h(x) = g(-10)$, still close to 0.
- If $x_1 = 1$ and $x_2 = 1, h(x) = g(10)$, which is close to 1.

This output matches the AND function, where 1 appears only when both inputs are 1. So, basically like that we approximated an **AND** with a neural network.

$x_1, x_2 \in \{0, 1\}$
$y = x_1 \text{AND} x_2$



$h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$



x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(+10) \approx 1$

OR function

To compute $x_1\ OR\ x_2$, we can do the same by modifying the network's parameters. We assign -10 for the bias parameter, and $+20$ for both weights associated to x_1 and x_2. Mathematically, this forms the hypothesis:

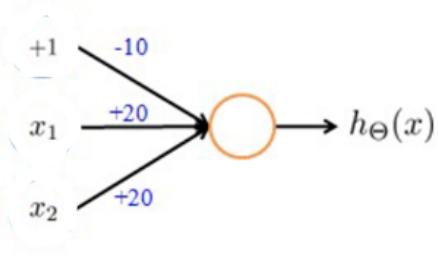
$$h(x) = g(-10 + 20x_1 + 20x_2)$$

This setup will result in:

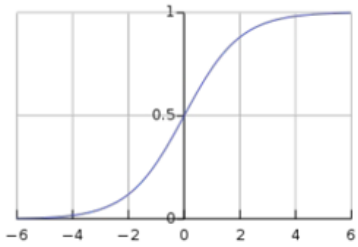
- If $x_1 = 0$ and $x_2 = 0$, $h(x) = g(-10)$, which is close to 0.
- If $x_1 = 0$ and $x_2 = 1$, $h(x) = g(10)$, also close to 1.
- If $x_1 = 1$ and $x_2 = 0$, $h(x) = g(10)$, still close to 1.
- If $x_1 = 1$ and $x_2 = 1$, $h(x) = g(30)$, which is very close to 1.

This matches the OR function, where 1 is output when at least one input is 1. So, basically like that we approximate an **OR** with a neural network.

$x_1, x_2 \in \{0, 1\}$
$y = x_1 \text{OR} x_2$



$h_\Theta(x) = g(-10 + 20x_1 + 20x_2)$



x_1	x_2	$h_\Theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(+10) \approx 1$
1	0	$g(+10) \approx 1$
1	1	$g(+30) \approx 1$

NOT function

To compute the NOT of x_1, again we consider a network with a single neuron, but in this case with just one input feature x_1. Then, by assigning $+10$ for the bias parameter and -20 for the weight associated to x_1 the hypothesis becomes:

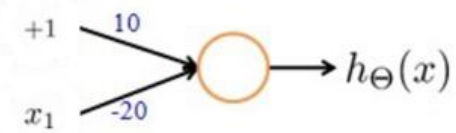
$$h(x) = g(+10 - 20x_1)$$

This setup yields:

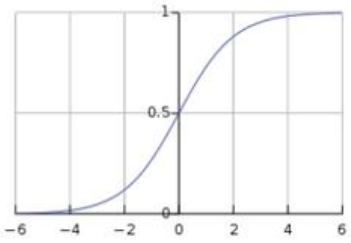
- If $x_1 = 0$, $h(x) = g(10)$, which is close to 1.
- If $x_1 = 1$, $h(x) = g(-10)$, which is close to 0.

Thus, this neural network effectively performs the **NOT** operation.

$x_1 \in \{0, 1\}$
$y = \text{NOT } x_1$



$h_\Theta(x) = g(10 - 20x_1)$



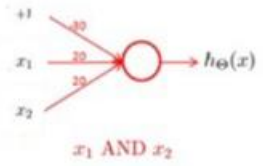
x_1	$h_\Theta(x)$
0	$g(+10) \approx 1$
1	$g(-10) \approx 0$

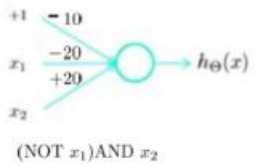
XOR function

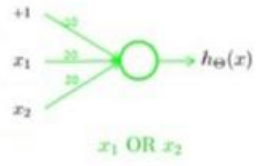
With these building blocks, we can now construct the XOR function by combining the AND, OR, and NOT networks:

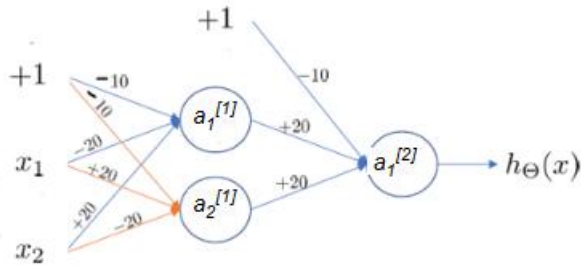
$$x_1 \; XOR \; x_2 = \left(x_1 \; AND \; (NOT \; x_2)\right) \; OR \; \left(x_2 \; AND \; (NOT \; x_1)\right)$$

$x_1 \in \{0, 1\}$
$y = x_1 \text{XOR } x_2$









x_1	x_2	$a_1^{[1]}$	$a_2^{[1]}$	$h_\Theta(x)$
0	0	0	0	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

The neural network to represent XOR can be constructed as follows:

- **Inputs** include x_1, x_2, and a bias unit (+1).
- **Hidden Layer**: Two activation units compute respectively: $a_1^{[1]}$ → "$x_2 \; AND \; (NOT \; x_1)$" and $a_2^{[1]}$ → "$x_1 \; AND \; (NOT \; x_2)$".
- **Output Layer**: A final activation unit $a_1^{[2]}$ computes the OR function to combine the results of the hidden layer.

This network outputs 1 when exactly one of x_1 or x_2 is 1, and 0 otherwise. The resulting nonlinear decision boundary effectively separates positive from negative examples in our XOR classification task.

7.6 Neural Networks for Multi-Class Classification

Multiclass classification is a task where we classify data into one of several categories. For example, in a **computer vision scenario**, instead of just identifying whether an object is a car or not, we might need to classify an image into one of four categories: **pedestrian, car, motorcycle, or truck**.

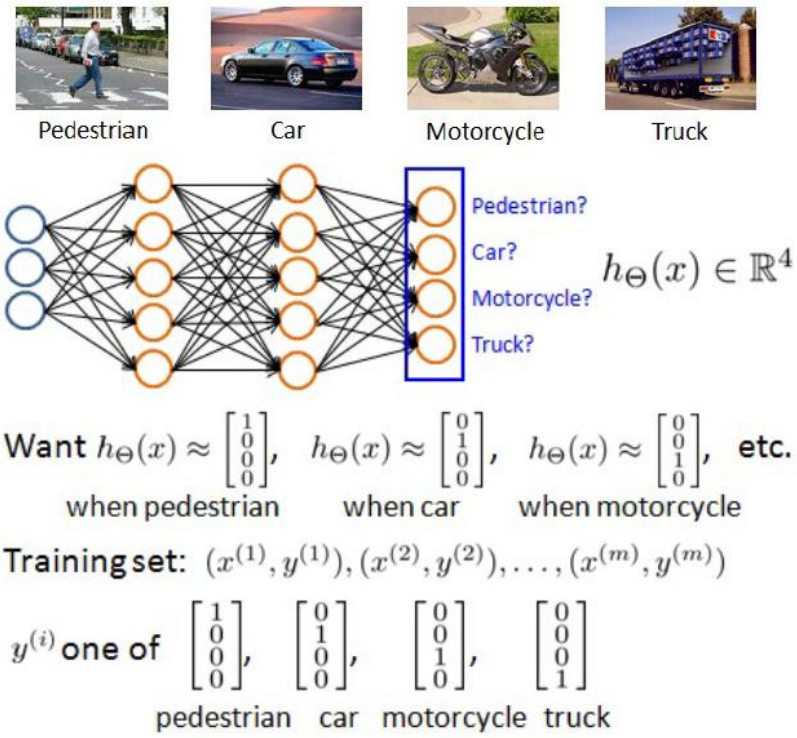
In this case, we need to modify our neural network to handle the additional categories. In particular we need to include **multiple output units—one for each class**. For four classes, we configure the neural network with four output units, each corresponding to a specific category. Like that, instead of producing a single output, the network generates a vector of four values, each representing a "yes/no decision" that the input belongs to a given category:

- If the image is of a **pedestrian**, the output should be $[1, 0, 0, 0]^T$
- For a **car**, the output should be $[0, 1, 0, 0]^T$
- For a **motorcycle**, the output should be $[0, 0, 1, 0]^T$
- And for a **truck**, the output should be $[0, 0, 0, 1]^T$

To train the neural network effectively doing this, we need to provide it with a properly formatted training set. In particular, each training example consists of a pair $(x^{(i)}, y^{(i)})$, where $x^{(i)}$ is an input image, and $y^{(i)}$ is its corresponding label vector representing the category of the image. Instead of using integer labels (e.g., 1, 2, 3, or 4), we adopt a **one-hot encoding** scheme to represent each category as a **binary vector**. In this representation:

- A pedestrian image would have a label vector of $y^{(i)} = [1, 0, 0, 0]^T$
- A car image would have a label vector of $y^{(i)} = [0, 1, 0, 0]^T$
- A motorcycle image would have a label vector of $y^{(i)} = [0, 0, 1, 0]^T$
- A truck image would have a label vector of $y^{(i)} = [0, 0, 0, 1]^T$

Goal: The neural network's task is to adjust its parameters so that, given an input image $x^{(i)}$, it can output a vector $h_\theta(x^{(i)})$ that closely matches the true label vector $y^{(i)}$.



In summary, setting up a neural network for multiclass classification involves configuring multiple output units to represent each category. By structuring the training data appropriately and using a suitable cost function (as we will see soon), the network can learn to differentiate between multiple categories.

7.7 Neural Network Cost Function

As for the logistic and regression problems, to optimize the parameters of a neural network, we minimize a cost function specific to the task at hand, whether classification or regression.

To characterize the cost function in the case of neural networks, we denote by convention:

- L = total number of layers in the network
- d_{l-1} = number of units in layer $l-1$, while s_l = number of units in layer l (not counting bias units)
- K = number of output units/classes. This of course is useful just in classification where our neural network may have multiple output nodes. In that case we denote $\big(h(x)\big)_k$ to indicate a hypothesis that results in the k^{th} output.

Cost Function for Regression

For regression tasks, we use a squared error cost function, adapted similarly to the neural network architecture. Recall the linear regression cost function was the Mean Squared Error (MSE):

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\big(h(x^{(i)}) - y^{(i)}\big)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

For a neural network performing regression, this becomes:

$$J(\theta, b) = \frac{1}{2m}\sum_{i=1}^{m}\left(h\left(x^{(i)}\right) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{v=1}^{d_{l-1}}\sum_{j=1}^{s_l}\left(\theta_{jv}^{[l]}\right)^2$$

where $h\left(x^{(i)}\right) = \theta x^{(i)} + b$

So, in this case the cost function already used by linear regression is updated by adding two summations to the regularization term which sums over all parameters in all layers of the network.

Cost Function for Binary Classification

Recall that for logistic regression in the case of binary classification, the cost function is the Binary Cross-Entropy (BCE) which is expressed as:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h\left(x^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - h\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

For a neural network performing binary-classification (one output unit), this becomes:

$$J(\theta, b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h\left(x^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - h\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{v=1}^{d_{l-1}}\sum_{j=1}^{s_l}\left(\theta_{jv}^{[l]}\right)^2$$

where $h\left(x^{(i)}\right) = g\left(\theta x^{(i)} + b\right)$

In this case, again the cost function already used by logistic regression is updated by adding only the final two summations whose purpose was just described above.

Cost Function for Multiclass Classification

For **multi-class** classification, the cost function is a generalization of the BCE function. Instead of a single output unit, we now handle K output units, corresponding to each class:

$$h(x) \in \mathbb{R}^K \quad with\ \left(h(x)\right)_k = k^{th} output$$

$$J(\theta, b) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\log\left(\left(h\left(x^{(i)}\right)\right)_k\right) + \left(1 - y_k^{(i)}\right)\log\left(1 - \left(h\left(x^{(i)}\right)\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{v=1}^{d_{l-1}}\sum_{j=1}^{s_l}\left(\theta_{jv}^{[l]}\right)^2$$

Here we have a summation over $k = 1$ to K to account for errors across all K output classes and again a regularization term summing over all parameters in all layers of the network.

The term $\left(1 - y_k^{(i)}\right)\log\left(1 - \left(h\left(x^{(i)}\right)\right)_k\right)$ is redundant and can be omitted due to the nature of one-hot encoding. We saw above that in one-hot encoding only one value equals 1 (indicating the correct class), while all other are 0. This means that for each example, the summation over K will include only the term corresponding to the correct class, while for all other classes the corresponding terms contribute nothing to the loss. Consequently, the $\left(1 - y_k^{(i)}\right)\log\left(1 - \left(h\left(x^{(i)}\right)\right)_k\right)$ term becomes unnecessary and does not affect the outcome.

With this simplification, the cost function for multi-class classification in a neural network reduces to:

$$J(\theta, b) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\log\left(\left(h\left(x^{(i)}\right)\right)_k\right) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{v=1}^{d_{l-1}}\sum_{j=1}^{s_l}\left(\theta_{jv}^{[l]}\right)^2$$

This cost function is known as **Cross-Entropy Loss**. Binary Cross-Entropy is simply a specific version of Cross - Entropy when you have only two classes.

In multi-class classification, the **softmax function** is used in the output layer to convert the raw output values (pre-activation values) also known as **logits** into a probability distribution. The softmax function for the k-th class of the i-th example is:

$$\left(h_\theta\left(x^{(i)}\right)\right)_k = s\left(z_k^{(i)}\right) = \frac{\exp\left(z_k^{(i)}\right)}{\sum_{j=1}^{K}\exp\left(z_j^{(i)}\right)}$$

where $z_k^{(i)}$ is the raw output (logit) for the k-th class for the i-th example. The denominator sums over all K classes, ensuring that the output values are normalized to form a valid probability distribution.

This ensures that the network outputs a vector of probabilities, with each element corresponding to the probability of the input belonging to a specific class. The class with the highest probability is then selected as the predicted class.

Therefore:

- **Binary Classification**: Combines **sigmoid activation** with **Binary Cross-Entropy** loss to handle the single output unit representing the probability of one class.
- **Multi-class Classification**: Combines **softmax activation** with **Cross-Entropy** loss to handle multiple output units, each representing the probability of a specific class.

7.8 Neural Network Backpropagation

After defining the cost function $J(\theta, b)$, our objective is to minimize it by finding the optimal values for all network parameters:

$$\min_{\theta,b} J(\theta, b)$$

This is true for both regression and classification tasks, and to achieve this, we can apply a minimization algorithm such as gradient descent as we always did until now. As we know, to use gradient-based optimization methods, we need the gradient of the cost function. In our case this involves compute the gradients of the weight matrix and bias vector for each layer:

$$\frac{\partial}{\partial \Theta^{[l]}} J(\theta, b)$$

$$\frac{\partial}{\partial B^{[l]}} J(\theta, b)$$

However, this calculation isn't straightforward since a neural network's parameters interact across layers, the contribution of each parameter to the overall error depends not only on its own value but also on the error passed from subsequent layers.

So, we need an efficient technique for computing the gradient of the cost function for our neural network. We will now see that this can be achieved using a local message-passing scheme in which information is sent backwards through the network and is known as **Error Backpropagation**, or sometimes simply as **Backprop**.

Preliminary: Chain Rule

Before diving into backpropagation details, let's resume an essential component that we will use which is the chain rule.

The **chain rule** provides a way to compute the derivative of a composite function by relating the derivatives of its individual components. If we have a composition of two differentiable functions, $f\big(g(x)\big)$, the chain rule states that the derivative of this composition is the derivative of the outer function f with respect to g, multiplied by the derivative of the inner function g with respect to x. In Leibniz notation, this is expressed as:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

When dealing with a composition of more than two functions, such as $h\Big(f\big(g(x)\big)\Big)$, we apply the chain rule sequentially. For example, the derivative of h with respect to x can be written as the composite of h with $f(g)$:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f}\frac{\partial f}{\partial x}$$

where:

$$\frac{\partial h}{\partial f} = \frac{\partial h}{\partial f}\frac{\partial f}{\partial g}$$

Alternatively, the derivative of h with respect to x can be also written as the composite of $h(f)$ with g:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial g}\frac{\partial g}{\partial x}$$

where:

$$\frac{\partial h}{\partial g} = \frac{\partial h}{\partial f}\frac{\partial f}{\partial g}$$

Ultimately, we can also express it directly as:

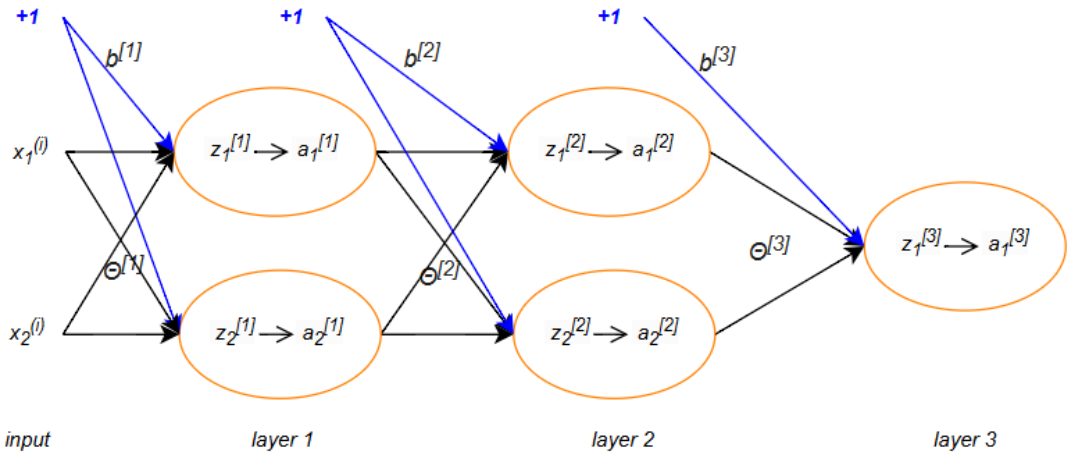
$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f}\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

The chain rule as we will see is crucial in backpropagation since it enables us to handle complex compositions by breaking them into manageable steps, allowing derivatives to be calculated layer by layer.

Backpropagation with Scalar Notation

This section explains backpropagation using scalar notation, which will help us to build an intuitive understanding of how the backpropagation process works. In this approach, we will treat all elements of the neural network as scalars, allowing us to work with straightforward, analytic notation for derivatives. However, this method is not the correct one because, as we have seen, neural network elements are vectors and matrices. I made this choice to try to be aligned as possible with the slides of the course. So, here what we will do is derive the basic formulas with scalar notation and at the end we will transition to the correct vectorial notation, as it's important to develop the right notation to work with neural networks.

Let's start our discussion. Suppose we have the following neural network:



and let's consider a simple regression case (without regularization), which makes it easier to derive the gradients for the loss function. Our regression loss in its vectorial form is expressed as:

$$J(\theta, b) = \frac{1}{2}\left(a^{[3]} - y\right)^2$$

where $a^{[3]} = h_\theta(x)$ is the output of the network and y is the true value.

Note: The term $\frac{1}{m}$ has not vanished from the MSE formula it is simply in this case equal to 1 since we are considering a single training sample ($m = 1$).

We first start by doing the **forward propagation** as we already seen previously. So given the training example (x, y) (recall that we indicate $x = a^{(0)}$) we push it through the network as:

$$z^{[1]} = \theta^{[1]} a^{[0]} + b^{[1]}$$

$$a^{[1]} = g\left(z^{[1]}\right)$$

$$z^{[2]} = \theta^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = g\left(z^{[2]}\right)$$

$$z^{[3]} = \theta^{[3]} a^{[2]} + b^{[3]}$$

$$h_\theta(x) = a^{[3]} = g\left(z^{[3]}\right)$$

Note: This process generates the network's output hypothesis, which in the case of regression is a scalar (a single real number since we consider a single activation unit at the output), but could be a vector in the case of multi-class classification.

At this point, let's figure out how to calculate the gradients $\frac{\partial}{\partial \theta^{[l]}} J(\theta, b)$ and $\frac{\partial}{\partial b^{[l]}} J(\theta, b)$. To achieve this, we use, as we anticipated above, the backpropagation algorithm, where we walk through the network in reverse order (i.e from the last layer to the input). Let's go in the detail of it.

For this first derivation of backprop let's just focus on computing the partial derivates w.r.t to the weight matrix parameters, then we will also compute the partial derivates w.r.t the bias parameters.

We start by computing the gradient of the cost function with respect to $\theta^{[3]}$ (therefore we start by computing the gradient of the cost function w.r.t to the theta matrix of layer 3). Since it is a composite function of multiple differentiable functions, we can apply the chain rule to help us in solving it (look the *forward propagation* scheme we have defined above to help you derive them):

$$\frac{\partial J}{\partial \theta^{[3]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial \theta^{[3]}}$$

and we can compute each of these partial derivatives separately:

- $\frac{\partial J}{\partial a^{[3]}} = \frac{\partial \frac{1}{2}(a^{[3]}-y)^2}{\partial a^{[3]}} = \frac{1}{2} 2 \left(a^{[3]} - y\right) = \left(a^{[3]} - y\right)$
- $\frac{\partial a^{[3]}}{\partial z^{[3]}} = \frac{\partial g(z^{[3]})}{\partial z^{[3]}} = g'\left(z^{[3]}\right)$
- $\frac{\partial z^{[3]}}{\partial \theta^{[3]}} = \frac{\partial (\theta^{[3]} a^{[2]} + b^{[3]})}{\partial \theta^{[3]}} = a^{[2]}$

Thus, the gradient with respect to $\theta^{[3]}$ is:

$$\frac{\partial J}{\partial \theta^{[3]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial \theta^{[3]}} = \left(a^{[3]} - y\right) g'\left(z^{[3]}\right) a^{[2]}$$

Let's indicate $\left(a^{[3]} - y\right) g'\left(z^{[3]}\right)$ with $\boldsymbol{\delta^{[3]}}$ (soon it will be clearer what δ will mean):

$$\delta^{[3]} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}}$$

Thus, the gradient with respect to $\theta^{[3]}$ becomes:

$$\frac{\partial J}{\partial \theta^{[3]}} = \delta^{[3]} a^{[2]}$$

Next, we calculate the gradient with respect to $\theta^{[2]}$. Applying the chain rule again:

$$\frac{\partial J}{\partial \theta^{[2]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial \theta^{[2]}}$$

We already know the terms $\frac{\partial J}{\partial a^{[3]}}$ and $\frac{\partial a^{[3]}}{\partial z^{[3]}}$, which we denoted as $\delta^{[3]}$. So, substituting we get:

$$\frac{\partial J}{\partial \theta^{[2]}} = \delta^{[3]} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial \theta^{[2]}}$$

Let's compute the other terms not already computed:

- $\frac{\partial z^{[3]}}{\partial a^{[2]}} = \frac{\partial \theta^{[3]} a^{[2]} + b^{[3]}}{\partial a^{[3]}} = \theta^{[3]}$
- $\frac{\partial a^{[2]}}{\partial z^{[2]}} = \frac{\partial g(z^{[2]})}{\partial z^{[2]}} = g'\left(z^{[2]}\right)$
- $\frac{\partial z^{[2]}}{\partial \theta^{[2]}} = \frac{\partial \theta^{[2]} a^{[1]} + b^{[2]}}{\partial \theta^{[2]}} = a^{[1]}$

Substituting these into the equation gives:

$$\frac{\partial J}{\partial \theta^{[2]}} = \delta^{[3]} \theta^{[3]} g'\left(z^{[2]}\right) a^{[1]}$$

Let's indicate $\delta^{[3]} \theta^{[3]} g'\left(z^{[2]}\right)$ with $\delta^{[2]}$:

$$\delta^{[2]} = \delta^{[3]} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} = \delta^{[3]} \theta^{[3]} g'\left(z^{[2]}\right)$$

Thus, the gradient with respect to $\theta^{[2]}$ becomes:

$$\frac{\partial J}{\partial \theta^{[2]}} = \delta^{[2]} a^{[1]}$$

Finally, we compute the gradient with respect to $\theta^{[1]}$. Again, we apply the chain rule:

$$\frac{\partial J}{\partial \theta^{[1]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial \theta^{[1]}}$$

We indicated $\frac{\partial J}{\partial a^{[3]}}, \frac{\partial a^{[3]}}{\partial z^{[3]}}$ terms with $\delta^{[3]}$, so we can substitute:

$$\frac{\partial J}{\partial \theta^{[1]}} = \delta^{[3]} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial \theta^{[1]}}$$

We indicated $\delta^{[3]} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}}$ terms with $\delta^{[2]}$, so we can substitute:

$$\frac{\partial J}{\partial \theta^{[1]}} = \delta^{[2]} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial \theta^{[1]}}$$

Let's compute the remaining terms:

- $\frac{\partial z^{[2]}}{\partial a^{[1]}} = \frac{\partial \theta^{[2]} a^{[1]} + b^{[2]}}{\partial a^{[1]}} = \theta^{[2]}$
- $\frac{\partial a^{[1]}}{\partial z^{[1]}} = \frac{\partial g(z^{[1]})}{\partial z^{[1]}} = g'(z^{[1]})$
- $\frac{\partial z^{[1]}}{\partial \theta^{[1]}} = \frac{\partial \theta^{[1]} a^{[0]} + b^{[1]}}{\partial \theta^{[1]}} = a^{[0]}$

Substituting these gives:

$$\frac{\partial J}{\partial \theta^{[1]}} = \delta^{[2]} \theta^{[2]} g'(z^{[1]}) a^{[0]}$$

Let's indicate $\delta^{[2]} \theta^{[2]} g'(z^{[1]})$ with $\delta^{[1]}$:

$$\delta^{[1]} = \delta^{[2]} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} = \delta^{[2]} \theta^{[2]} g'(z^{[1]})$$

Thus, the gradient with respect to $\theta^{[1]}$ becomes:

$$\frac{\partial J}{\partial \theta^{[1]}} = \delta^{[1]} a^{[0]}$$

So, by applying the chain rule to compute the gradients for each of the θ matrices in the network we got:

$$\frac{\partial J}{\partial \theta^{[3]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial \theta^{[3]}}$$

$$\frac{\partial J}{\partial \theta^{[2]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial \theta^{[2]}}$$

$$\frac{\partial J}{\partial \theta^{[1]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial \theta^{[1]}}$$

As we can see there were a lot of repetitive terms, but by simply using the δ terms above we end up with:

$$\frac{\partial J}{\partial \Theta^{[3]}} = \delta^{[3]} a^{[2]}$$

$$\frac{\partial J}{\partial \Theta^{[2]}} = \delta^{[2]} a^{[1]}$$

$$\frac{\partial J}{\partial \Theta^{[1]}} = \delta^{[1]} a^{[0]}$$

From this we deduce that:

- $\boldsymbol{\delta^{(l)}}$ term represent the "**error**" of neurons in layer l.
- To compute the gradient of the cost function with respect to the parameters of a given layer we need simply to perform a dot product between the error δ of that layer with the activation values from the previous layer:

$$\frac{\partial J}{\partial \Theta^{[l]}} = \delta^{[l]} a^{[l-1]} \; for \; l = 1, \dots, L$$

- The value of δ for the neurons in a particular hidden layer can be obtained starting from the error derived from the previous step (layer $l+1$):

$$\delta^{[l]} = \delta^{[l+1]} \Theta^{[l+1]} g'\left(z^{[l]}\right)$$

  E.g. the layer 2 gradient can be computed locally by using the error $\delta^{[3]}$ arrived up to it back; the layer 1 gradient can be computed by passing the error $\delta^{[2]}$ come up to it back.

Moreover, remembering the property of chain rule we have seen above that say that "*the composite of h, f, and g (in that order) is also the composite of $h(f)$ with g*", we can rewrite the followings:

$$\frac{\partial J}{\partial \Theta^{[3]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial \Theta^{[3]}}$$

$$\frac{\partial J}{\partial \Theta^{[2]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial \Theta^{[2]}}$$

$$\frac{\partial J}{\partial \Theta^{[1]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial \Theta^{[1]}}$$

also as:

$$\frac{\partial J}{\partial \Theta^{[3]}} = \frac{\partial J}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial \Theta^{[3]}} \qquad where \; \frac{\partial J}{\partial z^{[3]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} = \delta^{[3]}$$

$$\frac{\partial J}{\partial \Theta^{[2]}} = \frac{\partial J}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial \Theta^{[2]}} \qquad where \; \frac{\partial J}{\partial z^{[2]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} = \delta^{[2]}$$

$$\frac{\partial J}{\partial \Theta^{[1]}} = \frac{\partial J}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial \Theta^{[1]}} \qquad where \; \frac{\partial J}{\partial z^{[1]}} = \frac{\partial J}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} = \delta^{[1]}$$

So, it follows that:

$$\boldsymbol{\delta^{[l]} = \frac{\partial J}{\partial z^{[l]}}}$$

Gradients for Biases: The gradients for the bias vectors are derived with the same logic we adopted for the weight matrices:

$$\frac{\partial J}{\partial b^{[3]}} = \frac{\partial J}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial b^{[3]}} = \delta^{[3]} \cdot \frac{\partial\left(\theta^{[3]}a^{[2]} + b^{[3]}\right)}{\partial b^{[3]}} = \delta^{[3]} \cdot 1 \qquad \Rightarrow \frac{\partial J}{\partial b^{[3]}} = \delta^{[3]}$$

$$\frac{\partial J}{\partial b^{[2]}} = \frac{\partial J}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial b^{[2]}} = \delta^{[2]}\frac{\partial z^{[2]}}{\partial b^{[2]}} = \delta^{[2]} \cdot 1 \qquad \Rightarrow \frac{\partial J}{\partial b^{[2]}} = \delta^{[2]}$$

$$\frac{\partial J}{\partial b^{[1]}} = \frac{\partial J}{\partial z^{[1]}}\frac{\partial z^{[1]}}{\partial b^{[1]}} = \delta^{[1]}\frac{\partial z^{[1]}}{\partial b^{[1]}} = \delta^{[1]} \cdot 1 \qquad \Rightarrow \frac{\partial J}{\partial b^{[1]}} = \delta^{[1]}$$

Backpropagation with Vectorial Notation

In the previous discussion, we introduced the concept of backpropagation using scalars to build an intuitive understanding of the process and the role of the error term δ. However, in practice, backpropagation involves vectors and matrices and failing to account for this can lead to significant issues.

Just to say one, a first well-known problem is that matrix multiplication is **not commutative**, unlike scalar multiplication. This means that, for example, while $a \cdot b = b \cdot a$ holds for scalars, this does not hold in matrix multiplication, i.e. $a \cdot b \neq b \cdot a$. Thus, the order of operations is crucial. Any failure to respect this order can cause dimensional errors or incorrect results.

Additionally, when adding or multiplying matrices or vectors, their dimensions must align. For instance, in the case of adding θa with B, the dimensions must be compatible.

We saw that, during backpropagation, we apply the chain rule multiple times to compute gradients. When working with scalars, we simply take regular derivatives. However, when dealing with **vectors and matrices**, the chain rule involves multiplying matrices of derivatives, which are known as **Jacobian matrices**.

The Jacobian matrix is a generalization of derivatives for functions that map vectors to vectors. Given a vector-valued function $f: \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian matrix of the function f in x is the matrix whose elements are the first-order partial derivatives of the components of f with respect to the variables in x.

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \cdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Jacobian matrices naturally emerge when differentiating functions that operate on **vectors** or **matrices**. For instance, in the case of a layer's activation (output) $a^{[l]}$ depending on its weighted input $z^{[l]}$ through an **activation function** $g(\cdot)$, the derivative $\frac{\partial a^{[l]}}{\partial z^{[l]}}$ is a Jacobian matrix. Or moreover, derivatives like $\frac{\partial z^{[l]}}{\partial \theta^{[l]}}$ are also Jacobian matrices.

The layout of the Jacobian depends on the convention used. In **matrix calculus**, there are two commonly used **layout conventions** that refer to how derivatives are arranged:

- **Numerator layout:** The Jacobian matrix is structured such that **outputs (dependent variables) as rows** and **inputs (independent variables) as columns**, i.e.,

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \cdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- **Denominator layout** (the opposite): The Jacobian matrix is structured such that **inputs (independent variables) as rows** and **outputs (dependent variables) as columns**, i.e.,

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \cdots \\ \frac{\partial f_1}{\partial x_n} & \frac{\partial f_2}{\partial x_n} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

The choice of layout has a direct impact on how gradient formulas are derived and written. For example, when applying the **chain rule** to a composition of functions $f\left(g\left(u(x)\right)\right)$, the order of derivatives differs between the two layouts. In the **numerator layout**, the chain rule is written as:

$$\frac{\partial f(g(u(x)))}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial u}\frac{\partial u}{\partial x}$$

while in the denominator layout, the chain rule is written as:

$$\frac{\partial f(g(u(x)))}{\partial x} = \frac{\partial u}{\partial x}\frac{\partial g}{\partial u}\frac{\partial f}{\partial g}$$

This reversal of the order of Jacobians that we have in **denominator layout** often appears more intuitively aligned with the way backpropagation is implemented in many deep learning frameworks such as **PyTorch**, which is why the denominator layout is more commonly used in this context.

This is just a part of the considerations you should take when performing backprop with vector and matrices.

A full vectorized derivation requires a deep understanding of **matrix calculus**. To simplify the discussion and make it practical for backpropagation, we will focus on key derivative identities that are directly relevant to our development. These identities are **inherent to the denominator layout**.

For a more comprehensive exploration of **matrix calculus identities**, you can refer to this **link**.

To keep things straightforward and avoid unnecessary complexity, I'll present the key relations directly here. However, if you're crazy or curious enough you can explore the full derivations on your own.

If you're really strong in math to work through the full derivation, and you'd like to share your findings with others, please get in touch with me. I'd be happy to upload your detailed derivation for the benefit of the community.

Suppose to have:

$$A = g(z)$$

$$z = \theta a + b$$

where $z \in \mathbb{R}^{d\times 1}$, $\theta \in \mathbb{R}^{d\times n}$, $a \in \mathbb{R}^{n\times 1}$, $b \in \mathbb{R}^{d\times 1}$ and $g(\cdot)$ is a function applied element-wise to z. In this case, the derivative identities we are interested in are:

$$\frac{\partial A}{\partial \theta} = \frac{\partial A}{\partial z} \cdot a^T$$

$$\frac{\partial A}{\partial a} = \theta^T \frac{\partial A}{\partial z}$$

$$\frac{\partial A}{\partial b} = \frac{\partial A}{\partial z}$$

$$\frac{\partial A}{\partial z} = \frac{\partial A}{\partial g(z)} \odot g'(z) \quad \textit{where } g(x) \textit{ is applied element\_wise to } z$$

The last relation assumes $g(x)$ is an element-wise function, which is the case for common activation functions like our sigmoid $\sigma(x)$.

Note: The operator $\odot$ is the **Hadamard product** (also known as the **element-wise product**) in which each element of the first matrix is multiplied by the corresponding element in the second matrix. The Hadamard product operates on identically shaped matrices and produces a third matrix of the same dimensions. For example:

$$\begin{bmatrix}1\\2\end{bmatrix} \odot \begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}1*3\\2*4\end{bmatrix} = \begin{bmatrix}3\\8\end{bmatrix}$$

You can notice the resulting matrix has dimensionality 2×1, which is the same as the first and the second ones.

Now, let's explore how the considerations from vectorized notation apply when computing the gradients of the cost function with respect to the weights and biases of a neural network. Specifically, we will focus on the gradients for the **output layer (layer 3)** and the **immediate preceding hidden layer (layer 2)**. This process can be generalized to any hidden layer in the network.

1. Let's compute $\frac{\partial J}{\partial \theta^{[3]}}$ and $\frac{\partial J}{\partial b^{[3]}}$:

$$\frac{\partial J}{\partial \theta^{[3]}} = \frac{\partial J}{\partial z^{[3]}} \left(a^{[2]}\right)^T$$

where

$$\delta^{[3]} = \frac{\partial J}{z^{[3]}} = \frac{\partial J}{\partial a^{[3]}} \odot g'(z^{[3]}) = (\boldsymbol{a^{[3]}} - \boldsymbol{y}) \odot \boldsymbol{g'}(\boldsymbol{z^{[3]}})$$

Remember that we define $a^{[3]} = g(z^{[3]})$.

This gives us that the gradient of $\Theta^{[3]}$ is:

$$\Rightarrow \frac{\partial J}{\partial \Theta^{[3]}} = \boldsymbol{\delta^{[3]}}(\boldsymbol{a^{[2]}})^{\boldsymbol{T}}$$

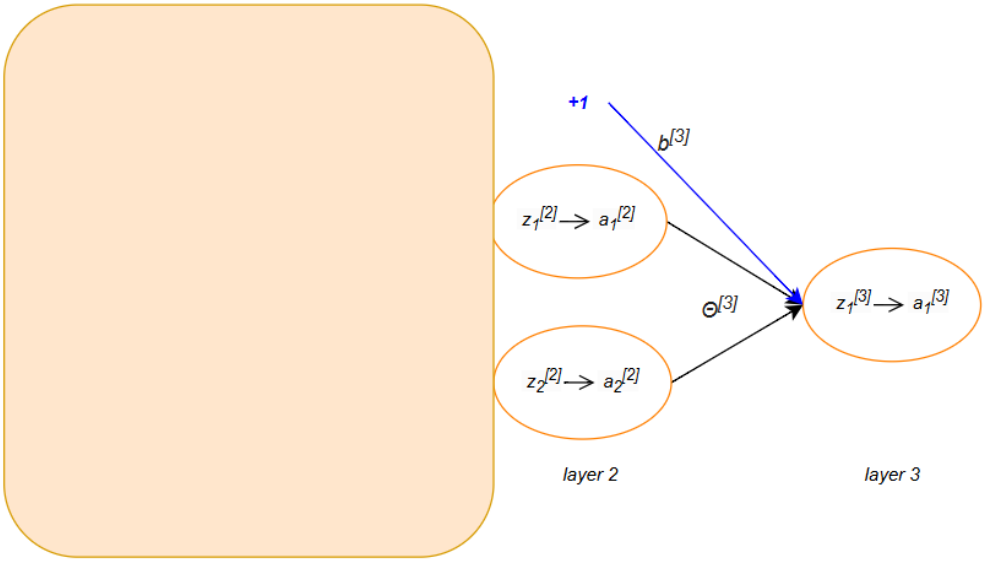
and the gradient of $b^{[3]}$ is:

$$\frac{\partial J}{\partial b^{[3]}} = \frac{\partial J}{\partial z^{[3]}} = \delta^{[3]}$$

Dimensionality Check:

- $\frac{\partial J}{\partial \Theta^{[3]}}$ must have the same dimensions as $\Theta^{[3]}$:
  - The dimensions of $(a^{[3]} - y)$ are 1×1.
  - The dimensions of $g'(z^{[3]})$ are 1×1, so $\delta^{[3]}$ is also 1×1.
  - $\Theta^{[3]}$ has dimensions 1×2.
  - Then dimensions of $a^{[2]}$ are 2×1, so $\delta^{[3]}(a^{[2]})^T$ has dimensions 1×2 which matches the dimensions of $\Theta^{[3]}$.

- $\frac{\partial J}{\partial b^{[3]}}$ must have the same dimensions as $b^{[3]}$.
  - $b^{[3]}$ has dimensions 1×1.
  - Since we saw that $\frac{\partial J}{\partial b^{[3]}} = \delta^{[3]}$ and we know that $\delta^{[3]}$ has dimensions 1×1, the dimensions match.



2. Now let's compute $\frac{\partial J}{\partial \Theta^{[2]}}$:

$$\frac{\partial J}{\partial \Theta^{[2]}} = \frac{\partial J}{\partial z^{[2]}} (a^{[1]})^T$$

Here:

$$\frac{\partial J}{z^{[2]}} = \frac{\partial J}{\partial a^{[2]}} \odot g'(z^{[2]})$$

and

$$\frac{\partial J}{\partial a^{[2]}} = (\theta^{[3]})^T \frac{\partial J}{\partial z^{[3]}} = (\theta^{[3]})^T \delta^{[3]}$$

So:

$$\delta^{[2]} = \frac{\partial J}{\partial z^{[2]}} = \left((\boldsymbol{\theta}^{[3]})^T \boldsymbol{\delta}^{[3]}\right) \odot \boldsymbol{g}'(\boldsymbol{z}^{[2]})$$

This gives us that the gradient of $\theta^{[2]}$ is:

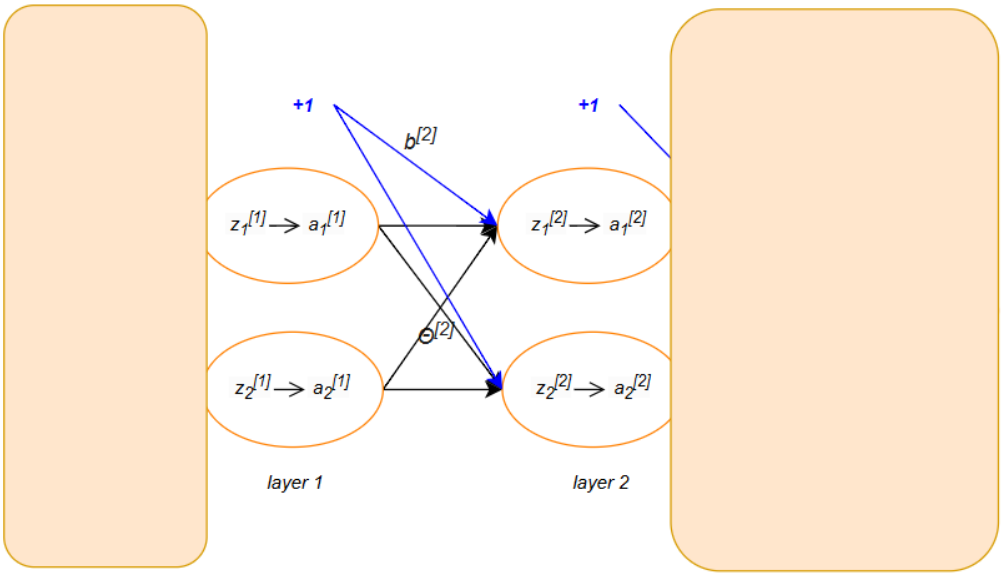
$$\Rightarrow \frac{\partial J}{\partial \theta^{[2]}} = \delta^{[2]}(a^{[1]})^T$$

and the gradient of $b^{[2]}$ is:

$$\frac{\partial J}{\partial b^{[2]}} = \frac{\partial J}{\partial z^{[2]}} = \delta^{[2]}$$

Dimensionality Check:

- $\frac{\partial J}{\partial \theta^{[2]}}$ must have the same dimensions as $\theta^{[2]}$:
  - We saw above that $\theta^{[3]}$ has dimensions 1×2 and $\delta^{[3]}$ has dimensions 1×1, so $(\theta^{[3]})^T \delta^{[3]}$ is 2×1.
  - $g'(z^{[2]})$ has dimensions 2×1, so $\delta^{[2]}$ is also 2×1.
  - $\theta^{[2]}$ has dimensions 2×2.
  - $a^{[1]}$ has dimensions 2×1, so $\delta^{[2]}(a^{[1]})^T$ is 2×2, which is the same as $\theta^{[2]}$.
- $\frac{\partial J}{\partial b^{[2]}}$ must have the same dimensions as $b^{[2]}$.
  - $b^{[2]}$ has dimensions 2×1.
  - Since we saw that $\frac{\partial J}{\partial b^{[2]}} = \delta^{[2]}$ and we know that $\delta^{[2]}$ has dimensions 2×1, the dimensions match.



Why Vectorized Notation Matters: You can notice that without considering vectorized notation, important mistakes can occur. For example, computing the gradient of $\Theta^{[3]}$ in scalar notation, we would have:

$$\frac{\partial J}{\partial \Theta^{[3]}} = (a^{[3]} - y) g'(z^{[3]})\, a^{(2)}$$

while in vectorial notation:

$$\frac{\partial J}{\partial \Theta^{[3]}} = [(a^{[3]} - y) \odot g'(z^{[3]})](a^{[2]})^T$$

If you think that is not an important enough problem since you could think that is just a matter of using the correct operators, then consider the computation for $\Theta^{[2]}$:

$$\frac{\partial J}{\partial \Theta^{[2]}} = \delta^{[3]} \Theta^{[3]} g'(z^{[2]}) a^{[1]}$$

that in in vectorial notation is:

$$\frac{\partial J}{\partial \Theta^{[2]}} = \left[\left((\Theta^{[3]})^T \delta^{[3]}\right) \odot g'(z^{[2]})\right](a^{[1]})^T$$

The order for scalar notation was wrong and without remapping it in vectorial notation we would not even notice.

This order is critical. If not correctly implemented, such mistakes could lead to errors in a practical implementation, particularly when dealing with matrix operations and ensuring correct dimensions.

Summary of Gradient Computations

In summary, the gradient computations follow these general rules:

$$\boldsymbol{\frac{\partial J}{\partial \Theta^{[l]}} = \delta^{[l]} (a^{[l-1]})^T \; for \; l = 1, \dots, L}$$

$$\boldsymbol{\frac{\partial J}{\partial b^{[l]}} = \delta^{[l]} \; for \; l = 1, \dots, L}$$

where we can compute the error δ as follows:

- For the **output layer** $\boldsymbol{L}$, the error $\delta^{[L]}$ is given by:

  $$\boldsymbol{\delta^{[L]} = \frac{\partial J}{\partial a^{[L]}} \odot g'(z^{[L]})}$$

  For regression problems we generally do not apply a final activation function (e.g., when predicting continuous values), so the error for the last layer simplifies to:

  $$\boldsymbol{\delta^{[L]} = (a^{[L]} - y)}$$

  In this case, the "error" at the output layer is simply the difference between the predicted value $a^{[L]}$ and the true value y.

  Note: However, if you want you can include a final activation function, such as sigmoid, to constrain the output (e.g., between 0 and 1) and in that case the error term will include the derivative of the activation function:

  $$\delta^{[L]} = (a^{[L]} - y) \odot g'(z^{[L]})$$

- For any **hidden layer** the error contribution δ is given by:

  $$\boldsymbol{\delta^{[l]} = \left((\Theta^{[l+1]})^T \delta^{[l+1]}\right) \odot g'(z^{[l]})}$$

The $\boldsymbol{\delta}^{[l]}$ is calculated by multiplying the transpose of theta matrix of layer $l+1$ with the delta of layer $l+1$. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by $z(l)$.

The **g-prime** derivative term depends on the type of activation function considered.

For example, if the activation function is sigmoid:

$$g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$$

it can be demonstrated that its derivative is:

$$\boldsymbol{g'(z) = \sigma(z) \odot (1 - \sigma(z))}$$

Note: Recall that we saw the derivative of the sigmoid in the Logistic Regression section, here we simply report its formulation in vectorial notation.

For our specific case, where we use the sigmoid function as activation, and we defined $a^{[l]} = g(z^{[l]}) = \sigma(z^{[l]})$ we end up with:

$$\boldsymbol{g'(z^{[l]}) = a^{[l]} \odot (1 - a^{[l]})}$$

Backpropagation Algorithm Derivation

Putting it all together, to compute all the gradients during backpropagation, we follow these steps:

1. **Forward Pass**:
   - First, perform a forward pass through the network to compute all pre-activations $z^{[l]}$ and activations $a^{[l]}$ for each layer $l = 1, \dots, L$
2. **Error Calculation for the Output Layer**:
   - Using the target output y, compute the error for the output layer L. In the case of a regression task, this error is simply the difference between the predicted output and the actual value:

$$\boldsymbol{\delta^{[L]} = a^{[L]} - y}$$

3. **Backpropagate the Error**:
   - To compute the errors for the hidden layers, backpropagate the error from the output layer to the earlier layers in order to compute the errors for the hidden layers $\delta^{[L-1]}, \delta^{[L-2]}, \dots, \delta^{[1]}$. Basically, for each hidden layer $l = L-1, \dots, 1$, the error $\delta^{[l]}$ is computed as:

$$\boldsymbol{\delta^{[l]} = \left(\left(\theta^{[l+1]}\right)^T \delta^{[l+1]}\right) \odot \left(a^{[l]} \odot \left(1 - a^{[l]}\right)\right)}$$

4. **Gradient Calculation**:
   - Once the errors are computed for all layers, we calculate the gradient of the cost function with respect to the parameters $\theta^{[l]}$ and $b^{[l]}$ for each layer l as follows:

$$\boldsymbol{\frac{\partial J}{\partial \theta^{[l]}} = \delta^{[l]}\left(a^{[l-1]}\right)^T + \lambda\theta^{[l]}}$$

$$\boldsymbol{\frac{\partial J}{\partial b^{[l]}} = \delta^{[l]}}$$

Note that here we added a **regularization** term to each weight matrix, but not for bias vectors.

This step ends our backpropagation algorithm giving us all the gradients $\frac{\partial}{\partial \Theta^{[l]}} J(\Theta, b)$ and $\frac{\partial}{\partial b^{[l]}} J(\Theta, b)$ for each layer.

After that we can use these gradients that we have found with backprop to update the parameters of each layer using **gradient descent**:

$$\Theta^{[l]} := \Theta^{[l]} - \alpha \frac{\partial J(\Theta, b)}{\partial \Theta^{[l]}}$$

$$b^{[l]} := b^{[l]} - \alpha \frac{\partial J(\Theta, b)}{\partial b^{[l]}}$$

In the explanation above, we have considered the case where we performed an update considering a single training example. Basically, we were performing **Stochastic Gradient Descent (SGD)**.

If we want to consider a dataset with more training samples and consider all of them to make an update, i.e. we want to perform **Batch Gradient Descent (BGD)** then we need to slightly modify it. We can follow two main approaches.

First method: Using Accumulators

One approach involves maintaining two "accumulators", $\Delta_\theta^{[l]}$ and $\Delta_b^{[l]}$, for each layer. These accumulators aggregate the gradients for the layer parameters across all training examples. The procedure is as follows:

1. We set $\Delta_\theta^{[l]} = 0$ and $\Delta_b^{[l]} = 0$ for all layers (**initialization**).
2. Then, for each training sample, iterating on all the layers of our network we update the accumulators as:

$$\Delta_\theta^{[l]} = \Delta_\theta^{[l]} + \delta^{[l]} \left(a^{[l-1]}\right)^T$$

$$\Delta_b^{[l]} = \Delta_b^{[l]} + \delta^{[l]}$$

3. After processing all m training examples, we compute the average gradients of each layer (same logic as BGD):

$$\frac{\partial J}{\partial \Theta^{[l]}} = \frac{1}{m} \left(\Delta_\theta^{[l]} + \lambda \Theta^{[l]}\right)$$

$$\frac{\partial J}{\partial b^{[l]}} = \frac{\Delta_b^{[l]}}{m}$$

Backprop Pseudo-algorithm (for BGD)

Training set $\left\{\left(x^{(1)}, y^{(1)}\right), \ldots, x^{(m)}, y^{(m)}\right\}$

Set $\Delta_\theta^{[l]} = 0$ (for all l)

Set $\Delta_b^{[l]} = 0$ (for all l)

For $i = 1$ to m

Set $a^{[0]} = x^{(i)}$

Perform forward propagation to compute $z^{[l]}$ and $a^{[l]}$ for $l = 1, \ldots, L$

Using $y^{(i)}$, compute $\delta^{[L]} = a^{[L]} - y^{(i)}$ (this in case of regression)

Compute $\delta^{[L-1]}, \delta^{[L-2]}, \ldots, \delta^{[1]}$

$\Delta_\theta^{[l]} = \Delta_\theta^{[l]} + \delta^{[l]}\left(a^{[l-1]}\right)^T$

$\Delta_b^{[l]} = \Delta_b^{[l]} + \delta^{[l]}$

At the end:

$$\frac{\partial J}{\partial \Theta^{[l]}} = \frac{1}{m}\left(\Delta_\theta^{[l]} + \lambda\Theta^{[l]}\right)$$

$$\frac{\partial J}{\partial b^{[l]}} = \frac{\Delta_b^{[l]}}{m}$$

Once we have the values of all gradients, we can update the parameters of each layer using as:

$$\Theta^{[l]} := \Theta^{[l]} - \alpha\frac{\partial J}{\partial \Theta^{[l]}}$$

$$b^{[l]} := b^{[l]} - \alpha\frac{\partial J}{\partial b^{[l]}}$$

Second Method: Fully Vectorized Implementation

An alternative, more efficient approach avoids explicit for loops (as we did above) by leveraging **vectorization**. This approach uses matrix operations to perform the forward and backward passes for all training examples simultaneously.

Suppose you have a training set with three examples $x^{(1)}, x^{(2)}, x^{(3)}$. The first-layer activations for each example are as follows:

$$z^{[1](1)} = \Theta^{[1]}x^{(1)} + b^{[1]}$$

$$z^{[1](2)} = \Theta^{[1]}x^{(2)} + b^{[1]}$$

$$z^{[1](3)} = \Theta^{[1]}x^{(3)} + b^{[1]}$$

Note the difference between square brackets $[\cdot]$, which refer to the layer number, and parenthesis $(\cdot)$, which refer to the training example number. Following the previous approach, we saw we implemented this using a for loop. However, as we will now see we can vectorize these operations. First, define:

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} \in \mathbb{R}^{n\times 3}$$

where in this case $m = 3$.

Note that we are stacking training examples in columns and not rows. We can then combine this into a single unified formulation:

$$Z^{[1]} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = \Theta^{[1]}X + b^{[1]}$$

You may notice that we are attempting to add $b^{[1]} \in \mathbb{R}^{d\times 1}$ to $\Theta^{[1]}X \in \mathbb{R}^{d\times 3}$ (where we recall d is the number of units in the layer) . Strictly following the rules of linear algebra, this is not allowed. In practice, however, this addition is performed using ***broadcasting***. We create an intermediate $\tilde{b}^{[1]} \in \mathbb{R}^{d\times 3}$:

$$\tilde{b}^{[1]} = \begin{bmatrix} | & | & | \\ b^{[1]} & b^{[1]} & b^{[1]} \\ | & | & | \end{bmatrix} \in \mathbb{R}^{n\times 3}$$

This allows us to perform the computation $Z^{[1]} = \Theta^{[1]}X + \tilde{b}^{[1]}$. In NumPy broadcasting is done implicitly.

Using this approach, we can apply similar vectorized operations for $A^{[l]}$, and thus, we can perform backpropagation directly without relying on explicit loops or accumulators. Specifically, we can compute the gradients for each layer efficiently using matrix operations as:

$$\frac{\partial J}{\partial \Theta^{[l]}} = \frac{1}{m}\left(\delta^{[l]}\left(A^{[l-1]}\right)^T + \lambda\Theta^{[l]}\right)$$

$$\frac{\partial J}{\partial b^{[l]}} = \frac{\delta^{[l]}}{m}$$

and once we have the values of all gradients, we can update the weight matrix and bias vector of each layer as follows:

$$\Theta^{[l]} \coloneqq \Theta^{[l]} - \alpha\frac{\partial J}{\partial \Theta^{[l]}}$$

$$b^{[l]} \coloneqq b^{[l]} - \alpha\frac{\partial J}{\partial b^{[l]}}$$

Note: In the above derivation, we assumed that the input matrix X has dimensions $\boldsymbol{n \times m}$, where n is the number of features and m is the number of training examples. However, in many real-world datasets, X is structured as $\boldsymbol{m \times n}$, with training instances as rows and features as columns. To align with the formulation here, you would need to transpose X (i.e., use $\boldsymbol{X^T}$) so that it matches the $n \times m$ shape required for these calculations. Similarly, y (which is $m \times 1$) should also be transposed, resulting in $\boldsymbol{y^T}$ with shape $1 \times m$ (output labels as a single row, one per example).

! By following this last note and using the second approach illustrated above you should now be aligned with the implementation from scratch shown in the code exercise.

+ Backpropagation for Classification

In the case of **classification**, the key difference lies in the **derivative with respect to the last layer**, as the cost function used for classification is different from the one used for regression.

While the error formulations for the hidden layers $\delta^{[l]}$ remain the same as in the regression case, we will focus on the computation of the error for the last layer $\delta^{[L]}$ (the output layer) which is the only one that changes.

BProp for Binary Classification (BCE)

The cost function for binary classification (using sigmoid activation) is the **BCE**:

$$J = -\left[y \log a^{[3]} + (1-y) \log\left(1 - a^{[3]}\right)\right]$$

Here we need to compute $\frac{\partial J}{\partial z^{[3]}}$ where we know that:

$$\delta^{[3]} = \frac{\partial J}{\partial z^{[3]}} = \frac{\partial J}{\partial a^{[3]}} \odot g'\left(z^{[3]}\right)$$

So, all we need to do is to compute $\frac{\partial J}{\partial a^{[3]}}$:

$$\frac{\partial J}{\partial a^{[3]}} = \frac{\partial\left\{-\left[y \log a^{[3]} + (1-y) \log\left(1 - a^{[3]}\right)\right]\right\}}{a^{[3]}} =$$

$$= -\left[\frac{1}{a^{[3]}} y + (-1) \frac{1}{1 - a^{[3]}} (1-y)\right] =$$

$$= -\left[\frac{y}{a^{[3]}} - \frac{(1-y)}{1 - a^{[3]}}\right]$$

Thus, the error term for the output layer becomes:

$$\boldsymbol{\delta^{[L]} = \frac{\partial J}{\partial z^{[L]}} = -\left[\frac{y}{a^{[L]}} - \frac{(1-y)}{1 - a^{[L]}}\right] \odot g'\left(z^{[L]}\right)}$$

Bprop for Multiclass Classification (CE)

Recall that in multi-class classification in the last layer we don't apply a sigmoid, but instead we apply a softmax. Therefore, in this case the output layer is given by $a^{[3]} = softmax\left(z^{[3]}\right)$.

For brevity let's indicate $a^{[3]}$ as s representing the predicted probabilities for each class (output probabilities), produced by the softmax function. The softmax function in our case will be:

$$s = softmax\left(z^{[3]}\right) = \frac{e^{z^{[3]}}}{\sum_{j=1}^{K} e^{z_j^{[3]}}}$$

For an individual class k, the corresponding predicted probability s_k is given by:

$$s_k = \frac{e^{z_k^{[3]}}}{\sum_{j=1}^{K} e^{z_j^{[3]}}}$$

Derivative of Softmax: The derivative of s_k with respect to z_j is given by:

$$\frac{\partial s_k}{\partial z_j} = \begin{cases} s_k(1 - s_k), & if\ j = k \\ -s_k s_j, & if\ j \neq k \end{cases}$$

which is typically expressed more compactly as:

$$\frac{\partial s_k}{\partial z_j} = s_k\left(\delta_{kj} - s_j\right)$$

where δ_{kj} is the Kronecker delta.

The cost function for multiclass classification using the softmax activation is the **CE** loss:

$$J = -\sum_{k=1}^{K} y_k \log(s_k)$$

where recall that y_k is the true label for the k-th class (using one-hot encoding).

All we need to do is to compute $\delta^{[3]} = \frac{\partial J}{\partial z^{[3]}}$.

We begin by computing $\frac{\partial J}{\partial z_j^{[3]}}$:

$$\frac{\partial J}{\partial z_j^{[3]}} = -\sum_{k=1}^{K} y_k \frac{\partial \log(s_k)}{z_j^{[3]}}$$

Recalling the derivative of the logarithm, $\frac{\partial}{\partial x} log(x) = \frac{1}{x}$, we rewrite this as:

$$-\sum_{k=1}^{K} y_k \frac{\partial \log(s_k)}{z_j^{[3]}} = -\sum_{k=1}^{K} y_k \frac{1}{s_k} \frac{\partial s_k}{z_j^{[3]}}$$

Next, we substitute the derivative of the softmax function:

$$-\sum_{k=1}^{K} y_k \frac{1}{s_k} \frac{\partial s_k}{z_j^{[3]}} = -\sum_{k=1}^{K} y_k \frac{1}{s_k} s_k(\delta_{kj} - s_k) = -\sum_{k=1}^{K} y_k (\delta_{kj} - s_k) =$$

Breaking this into two terms:

$$= -\sum_{k=1}^{K} y_k \delta_{kj} + \sum_{k=1}^{K} y_k s_k$$

Since $\delta_{kj} = 1$ only when $j = k$ and 0 everywhere else, the first term simplifies to $-y_k$:

$$= -y_k + \sum_{k=1}^{K} y_k s_k$$

Finally, using the fact that $\sum_{k=1}^{K} y_k = 1$ as y_k is a one-hot encoded vector we get:

$$\frac{\partial J}{\partial z_j^{[3]}} = -y_k + s_k = s_k - y_k$$

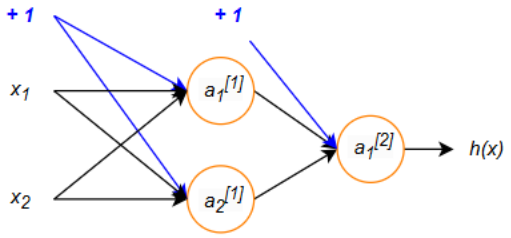
In vectorized form, this can be written as:

$$\boldsymbol{\delta^{[L]} = \frac{\partial J}{\partial z^{[L]}} = s - y}$$

7.9 Parameters Initialization

After describing all the steps that need to be addressed to compute the gradient to then minimize the cost function, however, the question that arises is: how to choose the initialization values of the parameters?

- **Zero initialization**: Initializing all parameters to zero may not be a very good idea since, after each update, they will always look the same due to the "symmetric" nature of the backpropagation algorithm.
  As an example, consider the neural network shown in the Figure. If all parameters are initialized to zero, then during forward propagation, we will have $a_1^{(1)} = a_2^{(1)}$ for the second layer's activations. This symmetry continues into backpropagation, where the gradients for the weights will be identical: $\frac{\partial J(\theta)}{\partial \theta_{11}^{[1]}} = \frac{\partial J(\theta)}{\partial \theta_{12}^{[1]}}$. As a result, when we apply gradient descent, the updated weights will remain identical, leading to $\theta_{11}^{[1]} = \theta_{12}^{[1]}$. Thus, this symmetry prevents the network from learning distinct features, which is why zero initialization is ineffective in training neural networks.



- **Random initialization**: To break this symmetry, **Random initialization** is used. Instead of setting weights to zero, they are initialized to small random values $\theta_{ij}^{[l]} \in [-\epsilon, \epsilon]$, typically drawn from a standard normal or uniform distribution. Or even better, we can make use of **Xavier or He initializations**, but we will see it more in detail in the deep learning course.

7.10 Activation Functions

At the beginning of this chapter, we highlighted that neural networks are far more efficient at learning complex, non-linear hypotheses. But how is this achieved? The key lies in the use of **non-linear activation functions**. These functions play a key role in introducing **non-linearity** into the model. Without them, a neural network—regardless of how many layers it has—would reduce to a linear model!

Moreover, what is truly remarkable is how neural networks learn these complex non-linear relationships. In polynomial regression, for instance, we explicitly define the type of non-linearity with polynomial terms such as quadratic or cubic terms of the initial features. In contrast, neural networks can implicitly learn more intricate non-linear relationships **implictly**. Let's understand how this happens.

At each layer, the network performs two basic steps: it computes a weighted sum of the inputs and then applies a non-linear activation function to the result. Although activation functions like the sigmoid may seem simple on their own, their true power emerges when they are applied across multiple layers sequentially. Each layer refines and builds upon the transformed outputs of the previous one, enabling the network to construct progressively sophisticated representations of the input data.

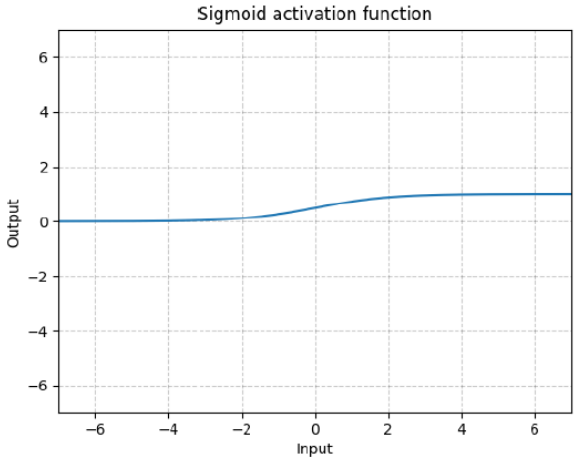
This **cascading process** of weighted sums and non-linear activations is what allows neural networks to model highly intricate patterns. It's this process that makes neural networks so powerful (I hope you understood this concept since is one of the most important aspects to understand) and this becomes even more apparent in **deep neural networks**, neural networks with many layers which amplify the complexity of these learned relationships allowing for example the network to learn features such as the shapes of objects in images.

Moreover, I want to highlight to you again that activation functions must be non-linear. For example, a function like the **identity function**, which is linear, does not introduce non-linearity making it unsuitable to be applied in a neural network (however identify function could be useful in other cases, so don't put it in the trash bin!).

Each layer in a neural network can theoretically use a different activation function. Different activation functions exhibit different characteristics which effects become more evident when we consider deep neural networks. Some of the most widely used activation functions include:

- **Sigmoid**: The **Sigmoid** function, as we already know, is an activation function defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



which maps input values to an output range between 0 and 1, making it particularly useful when the output needs to represent a probability or a binary decision.

However, Sigmoid has a critical drawback that affects its suitability for deep neural networks, it suffers of the so called “vanishing gradient” problem. Let’s understand what it is.

We saw that the derivative of Sigmoid is:

$$\sigma'(x) = \sigma(x) \odot \big(1 - \sigma(x)\big)$$

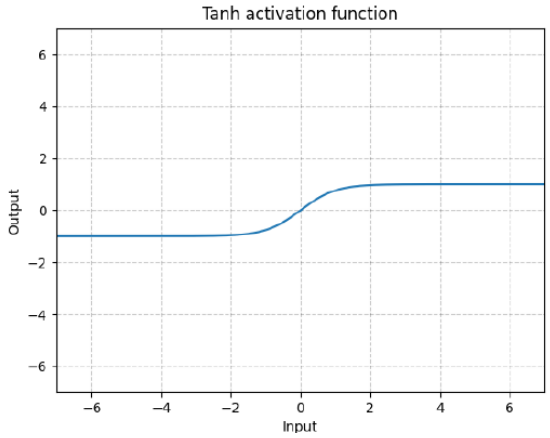
This derivative becomes very small when input values are far from zero, impacting the error term δ during backpropagation, since it is part of it ($\delta^{[l]} = \left(\left(\Theta^{[l+1]}\right)^T \delta^{[l+1]}\right) \odot \left(\boldsymbol{\sigma'(x^{[l]})}\right)$). Specifically:

  - For large positive inputs, $\sigma(x) \approx 1$, we will have that the term inside the derivate $\big(1 - \sigma(x)\big) \approx 0$, so $\sigma'(x) \approx 0$ and as consequence $\delta \approx 0$.
  - For large negative inputs, $\sigma(x) \approx 0$, we will have that the term inside the derivate $\sigma(x) \approx 0$, again resulting in $\sigma'(x) \approx 0$ and as consequence $\delta \approx 0$.

So, since $\sigma(x)$ produces values close to 0 or 1 for most input values (given its exponential S-shaped curve), its derivate $\sigma'(x)$ becomes almost always very small—often close to zero. . As a result, the gradients quickly diminish as they propagate backward through the multiple layers of the network, continuing to diminish and resulting in even smaller updates for earlier layers. This phenomenon, known as the **vanishing gradient problem**, can essentially "shut down" learning in deep networks, as earlier layers receive minimal updates. This cumulative effect renders deep neural networks with Sigmoid activation inefficient or even impractical, as they struggle to learn effectively across multiple layers.

- **Tanh**: Tanh (Hyperbolic Tangent) is an activation function defined as:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

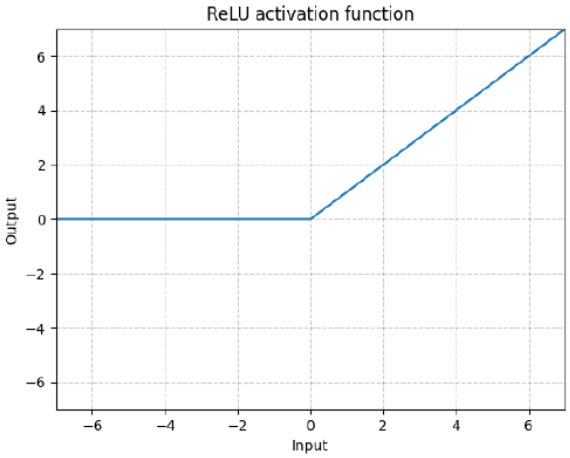


This function is similar to the Sigmoid function, but it is **centered around zero** and produces outputs in the **range of** $[-1, 1]$ rather than $[0, 1]$. As a result, its output has a **zero mean**, which helps to balance weight updates during training. In fact, convergence is usually faster if the mean of each input variable is close to zero.

However, like Sigmoid, Tanh also suffers from the vanishing gradient problem, making it less suitable for deep neural networks.

- **ReLU**: ReLU (Rectified Linear Unit) is an activation function defined as:

$$ReLU(x) = max\,(0, x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$



ReLU outputs the input directly if it's positive, and zero if the input is negative. This leads to a derivative:

$$ReLU'(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Since for $x \geq 0$, $ReLU'(x) = 1$, the gradient for positive values is consistently 1, allowing gradients to propagate as they are without shrinking and therefore avoiding the issue of small gradients (*vanishing gradient*).

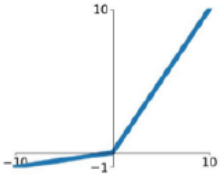
Although the ReLU has a non-zero gradient for positive input values, this is not the case for negative inputs, which can mean that some hidden units receive no 'error signal' during training. This leads to the famous **"dying ReLU" problem**, where neurons with negative inputs always output zero and may stop learning because their gradients are zero. **ReLU variants** were introduced to mitigate this issue by allowing small negative gradients.

ReLU has another important drawback that it is **non-differentiable at $x = 0$**. At this point, the function abruptly shifts from outputting 0 for negative inputs to increasing linearly for positive inputs, resulting in an undefined slope. In other words, there is no unique derivative at $x = 0$ because the left- and right-hand limits do not match. However, this is generally not a major concern in practice: the exact value $x = 0$ is an isolated point, and neural networks rarely operate precisely at zero. Additionally, gradient descent methods manage this discontinuity by using **subgradient** approximations, which allow effective training despite the non-differentiability at that point.

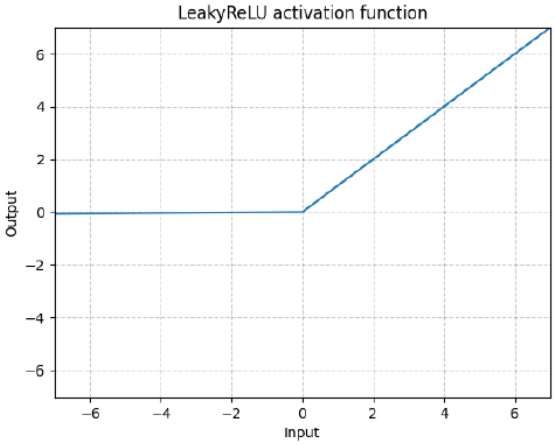
- **LeakyReLU**: LeakyReLU is a variant of ReLU defined as:

$$LeakyReLU(x) = max\,(\alpha x, x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$$

where α is a constant value (typically small, e.g., $\alpha = 0.01$) that controls the slope of the linear segment of the function for negative input values ($0 < \alpha < 1$).



The slope is very small →



Using a non-zero value for negative values leads to a derivate:

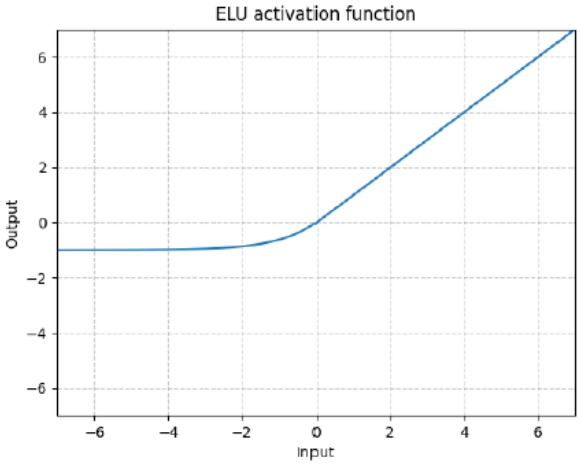
$$\text{LeakyReLU}'(x) = \begin{cases} 1, & x \geq 0 \\ \alpha, & x < 0 \end{cases}$$

which prevents the **dying ReLU problem** by ensuring negative inputs contribute to a small gradient, allowing the neuron to continue learning even with negative inputs.

- **ELU**: ELU (Exponential Linear Unit) is an activation function defined as:

$$ELU(x) = \begin{cases} x & , \quad x \geq 0 \\ \alpha\,(e^x - 1), & x < 0 \end{cases}$$

where α is a positive parameter that adjusts the slope of the function in the negative region.



Compared with Leaky ReLU, ELU has an exponential negative slope which leads to a slight curvature for negative values instead of being flat, which help to bring the mean activation closer to zero and speed up learning.

- **Maxout**: Maxout is one of the most flexible and powerful activation functions. A Maxout unit computes the maximum of "*n* linear functions", defined as:

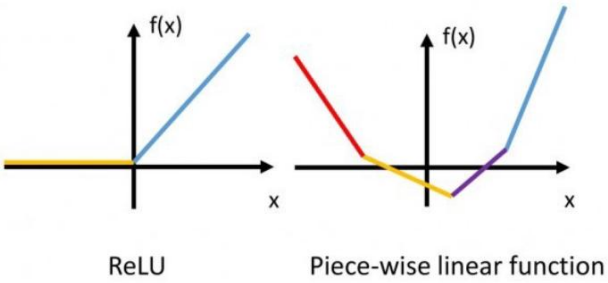
$$\max\,(w_1 x + b_1, \ldots, w_n x + b_n)$$

The number of linear functions, or "pieces", in this setup is defined by $\boldsymbol{n}$, a hyperparameter that is chosen beforehand.

Maxout can be seen as a generalization of ReLU because it can represent ReLU as a special case. In fact, if we define two linear functions for Maxout, with parameters $w_1 = 1$, $b_1 = 0$, $w_2 = 0$, $b_2 = 0$, Maxout simplifies to the ReLU function:

$$\max(1 \cdot x + 0, 0 \cdot x + 0) = \max(x, 0)$$

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

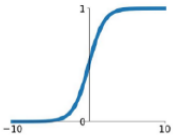


A Maxout unit can be interpreted as making a piece-wise linear approximation to an arbitrary **convex function** that can model more complex patterns than ReLU or ELU. This flexibility allows Maxout to approximate any arbitrary convex function, making it highly effective for feature learning.

  - **Advantages**: Maxout can capture a broader range of patterns, improving network efficiency and feature extraction.
  - **Drawbacks**: Maxout is computationally expensive because it requires evaluating multiple linear functions per unit, increasing resource and time requirements.

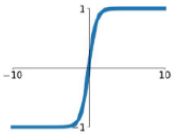
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



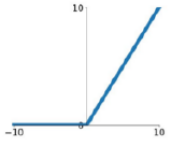
tanh

$$\tanh(x)$$



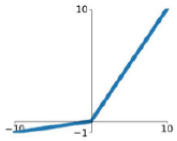
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

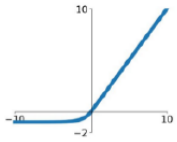


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Note: Softmax is also an activation function, but it is generally applied only at the output layer.

Historically, the **Sigmoid** function was widely used at the beginning but has mostly been replaced by ReLU due to better performance with deeper networks. **ReLU** has become the standard activation function for deep neural networks, but **ELU** and **Leaky ReLU** are often preferred for improved gradient flow. Maxout is not generally used for very deep neural networks due to its evaluating complexity (as enounced above).

8 Support Vector Machines

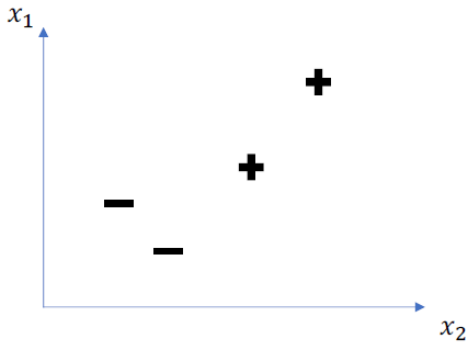
Support vector machines (**SVMs**) are among the best "off-the-shelf" supervised learning algorithms. SVMs are basically **binary classifiers** that take a completely different approach to solving statistical problems (in specific classification): unlike logistic regression or neural networks, SVMs focus on maximizing the separation, or margin, between two classes of data points. This novel strategy enables SVMs to excel in distinguishing between classes with a high degree of accuracy.

To tell the SVM story, we'll need to first talk about margins and the idea of separating data with a large "gap." Next, we will proceed by considering the concept of an optimal separating hyperplane, which consists of a simple type of linear classifier known as a Large Margin Classifier. We will then make a digression on Lagrange duality, which simplifies the optimization problem central to SVMs. At the end we will introduce kernel-based SVMs to handle non-linear decision boundaries.

8.1 SVM Intuition

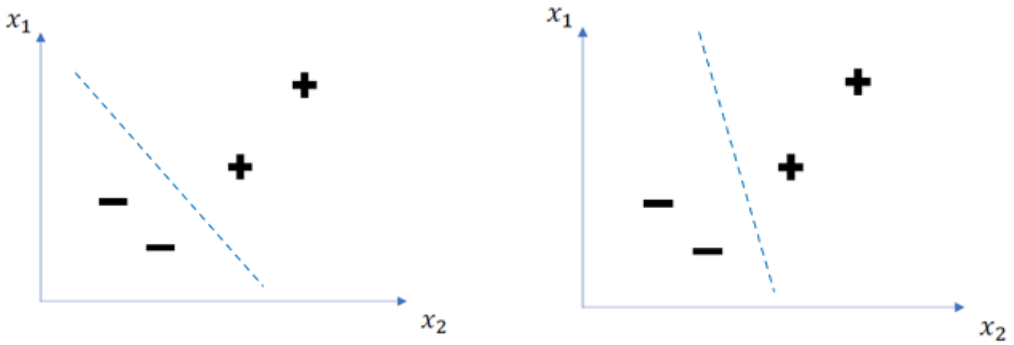
Consider a dataset in a two-dimensional space defined by two features x_1 and x_2. Suppose the data is categorized into two classes: positive (+) and negative (-). Our objective is to identify a hyperplane that that is able to cleanly separate the samples belonging to each class. A hyperplane in this context is a simple line defined by the equation:

$$\omega^T \cdot x + b$$



Observations:

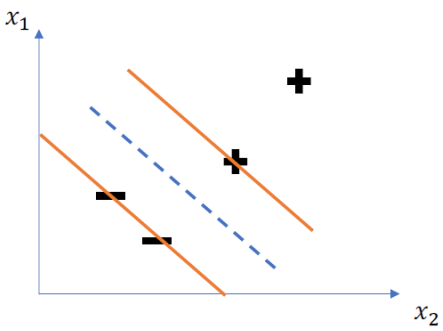
1. We can see that the training points $x^{(i)}$ do not all have the same distance from the hyperplane. For points far away from the separating hyperplane we can be sure of our particular class label. Conversely for points close to the separating hyperplane we have less confidence in our label class.
2. Theoretically there are infinite determinable hyperplanes that can separate two linearly separable classes. So, among these hyperplanes which one do we choose?



The idea behind the SVM algorithm is to determine the **best hyperplane** (the optimal one), in such a way as to be able to draw a kind of "street" around it that is as wide as possible.

➔ So let us assume that "**optimal**" means that we could draw this "street" around the hyperplane as wide as possible.

To this end, we want to identify the points that are able to determine the edges of this "street"; these points are called **Support Vectors**.



Note: The **support vectors** are parallel & equidistant from the hyperplane.

Once these points have been determined, we can consider as optimal, that hyperplane located within this "street", such that its amplitude is maximized.

The distance between the hyperplane and each of the straight lines identifying the support vectors is called the **Margin**. Thus, our goal is to maximize this distance.

➔ The goal of the algorithm is to determine only the hyperplane that guarantees the **maximum width**, so the highest **Margin**.

Note: Since the objective is to identify a hyperplane that completely separates data points belonging to different classes, ensuring a clear demarcation with the utmost margin width possible this approach is also known as **Hard Margin SVM**.

8.2 Maximum Margin

To derive the **objective function** for a SVM, we start by assuming that the dataset is ***linearly separable***.

The **decision rule** for classifying points is based on which side of the hyperplane they lie:

$$if\ \omega^T \cdot x^{(i)} + b \geq 0 \rightarrow y^{(i)} = +1$$

$$if\ \omega^T \cdot x^{(i)} + b < 0 \rightarrow y^{(i)} = -1$$

This rule is derived from the fact that, starting from the equation of the hyperplane $\omega \cdot x + b = 0$ we will obtain all the points on the hyperplane. Consequently, by placing it greater than zero, the equation would describe all points belonging to the right side of the hyperplane (+), conversely, by placing it less than zero it would describe all points belonging to the left side of the hyperplane (-).

Then we aim to maximize the **margin** by setting a minimum distance between the hyperplane and the support vectors; respectively this distance is +1 for class "+" ($y^{(i)} = +1$) and −1 for class "−" ($y^{(i)} = -1$).

Given that, the support vectors are determined by the following equations:

$$\omega^T \cdot x_+ + b = 1 \qquad (for\ y^{(i)} = +1)$$

$$\omega^T \cdot x_- + b = -1 \qquad (for\ y^{(i)} = -1)$$

Since no points can exist within the margin, we can deduce that:

- for all values with label $y^{(i)} = +1, \quad \omega^T \cdot x_+ + b \geq 1$
- for all values with label $y^{(i)} = -1, \quad \omega^T \cdot x_- + b \leq -1$

We can rewrite these constraints in a unified form.

If we consider the first formula, we realize that multiplying by $y^{(i)} = +1$ is like multiplying by +1 (which does not change the inequality):

$$y^{(i)}(\omega^T \cdot x_+ + b) \geq 1$$

But for the second formula multiplying by $y^{(i)} = -1$ is like multiplying by −1 which reverses the inequality, resulting in:

$$y_{-}(\omega^T \cdot x_{-} + b) \leq -1 \rightarrow -y^{(i)}(\omega^T \cdot x_{-} + b) \leq -1 \rightarrow y^{(i)}(\omega^T \cdot x_{-} + b) \geq 1$$

Thus, the general condition for all points i is:

$$y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) \geq 1 \quad \forall i = 1, \dots, m$$

That we can also rewrite as:

$$\boldsymbol{y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1 \geq 0 \quad \forall i = 1, \dots, m}$$

This constraint ensures that all points are classified correctly and remain outside the margin. So, this will be the constraint for our objective function.

Now, to find the **margin**, we need to express it in terms of the hyperplanes. While the equation of the hyperplane is initially unknown (we don't yet know $\vec{\omega}$ and b), we can compute the margin geometrically as the distance between the two support hyperplanes. Let's derive the relation between the margin M and the support vector hyperplanes.

Before diving into the derivation, let's revisit some foundational concepts that will help us to derive the margin relation.

Revisiting Dot Product, Vector Projection & Geometric Perspective of a Hyperplane

The **dot product** between two vectors a and b is a scalar defined as:

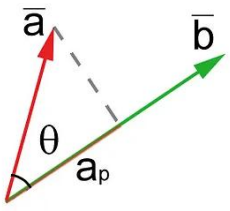
$$a \cdot b = \|a\|\|b\| \cos\theta$$

where θ is the angle between the vectors and provides a scalar measure of similarity between vectors.

From it we can easily recover that:

$$\cos\theta = \frac{a \cdot b}{\|a\|\|b\|}$$

The dot product is also closely related to **projections** of one vector onto another. Consider the projection of a onto b in the Figure below, indicated as a_p:



If a_p is the projection of the vector a onto b, then the **magnitude of a_p** is defined as:

$$\|a_p\| = \|a\| \cos\theta$$

where $\cos\theta$ is defined as:

$$\cos\theta = \frac{a \cdot b}{\|a\|\|b\|}$$

Now simply substituting $\cos\theta$ in $\|a_p\|$ and simplifying we get:

$$\|a_p\| = \|a\|\cos\theta = \|a\|\frac{a \cdot b}{\|a\|\|b\|} =$$

$$= \frac{a \cdot b}{\|b\|} = \boldsymbol{a} \cdot \frac{\boldsymbol{b}}{\|\boldsymbol{b}\|}$$

Thus, the magnitude of the projection of a onto b is equal to the dot product between a and the **unit vector** in the direction of b (i.e. $b/\|b\|$).

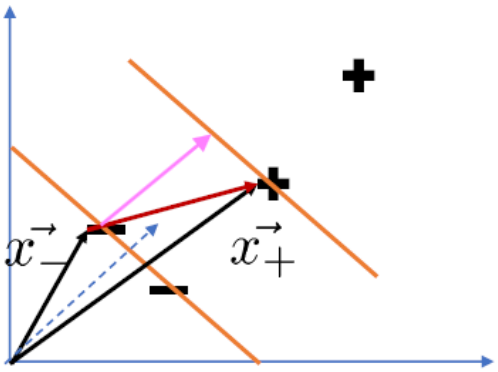
Moreover, **geometrically**, a hyperplane $\omega^T \cdot x + b = 0$ is defined by:

- its **normal vector** ω which is perpendicular to the hyperplane and gives its orientation.
- its **bias term** $\boldsymbol{b}$, determining the hyperplane's position relative to the origin.

Now we have all the fundamental concepts needed to derive the margin relation.

Deriving Margin

Now let's come back to margin formualtion. Looking at the Figure below we have that:



- The margin width M corresponds to the width of the space between the two support hyperplanes (the "street").
- The red vector represents the difference between the right and left support vectors $x_+ - x_-$.
- The purple vector represents a unit vector perpendicular to the margin.

To calculate the margin width, what we can think to do is project the difference vector, $x_+ - x_-$, onto the unit vector perpendicular to the margin. Now, since we know that the hyperplane's normal vector, ω, is already perpendicular to the margin, we can think of using it after converting it into a unit vector. Therefore, we can define the margin width M as:

$$M = (x_+ - x_-) \cdot \frac{\omega}{\|\omega\|}$$

Recalling that the two support vectors have the following equations:

$$\omega \cdot x_+ + b = 1 \qquad (for\ y^{(i)} = +1)$$

$$\omega \cdot x_- + b = -1 \qquad (for\ y^{(i)} = -1)$$

the margin width expression simplifies as follows:

$$M = (x_+ - x_-) \cdot \frac{\omega}{\|\omega\|} = \left(\left(\frac{1-b}{\omega}\right) - \left(\frac{-1-b}{\omega}\right)\right) \cdot \frac{\omega}{\|\omega\|} = \frac{2}{\omega} \cdot \frac{\omega}{\|\omega\|} =\cdot \frac{2}{\|\omega\|}$$

So, we found the relation between Margin M and the two support vectors.

Now, wanting to maximize the area of the road identified by the support vectors, the objective will be to **maximize the margin**:

$$\max(M) = \max \frac{2}{\|\omega\|}$$

At this point, as usual, it is possible to move from a maximization problem to a minimization problem. In this case, our problem undergoes the following transformation:

$$\max(M) = \max \frac{2}{\|\omega\|} = \min \frac{1}{2} \|\omega\|$$

Evaluating $\min \frac{1}{2} \|\vec{\omega}\|$ is equivalent to $\min \frac{1}{2} \|\vec{\omega}\|^2$ and the SVM prefers the second one. So, we get:

$$\max(M) = \min \frac{1}{2} \|\omega\|^2$$

This will be our new objective function.

Therefore, we have obtained that we can maximize the margin M by minimizing the quantity:

$$\boldsymbol{\max(M) = \min_{\omega,b} \frac{1}{2} \|\omega\|^2}$$

and by respecting the constraint:

$$\boldsymbol{y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1 \geq 0 \quad \forall i = 1, \dots, m}$$

This represents an optimization problem with a convex quadratic objective and only linear constraints. While such problems can be solved using quadratic programming (QP) software, we will take a different and more insightful approach. Specifically, we will explore **Lagrange duality**, which will lead us to the dual form of our optimization problem. The dual form is not just an alternative way to express the problem—it will play a key role in allowing us to use kernels to get optimal margin classifiers to work efficiently in very high dimensional spaces. The dual form will also allow us to derive an efficient algorithm for solving the above optimization problem that will typically do much better than generic QP software.

Therefore, let's temporarily put aside SVM and talk about solving constrained optimization problems.

8.3 Lagrange Duality

Method of Lagrange Multipliers

The **method of *Lagrange Multipliers*** is a mathematical strategy for solving constrained optimization problems, specifically those involving equality constraints.

Consider a problem where the objective is to minimize/maximize a function (ω), subject to equality constraints of the form $g_i(\omega) = 0, \forall i = 1, \dots, l$. E.g. in case minimization:

$$\min_{\omega} f(\omega)$$

$$s.t. \quad g_i(\omega) = 0, \quad \forall i = 1, \dots, l$$

This method transforms this constrained problem into an equivalent **unconstrained** one, enabling the use of standard techniques for finding extrema (minimum or maximum) like derivates. To do that it constructs a *Lagrangian function*, which incorporates the constraints directly into the objective function. This is reached by introducing auxiliary variables, called *Lagrange multipliers*, which act as penalty terms for violations of the constraints. The ***Lagrangian function*** is defined as:

$$L(\omega, \beta) = f(\omega) + \sum_{j=1}^{l} \beta_j \cdot g_j(\omega)$$

where β_j are the ***Lagrange multipliers*** associated with each constraint. By reformulating the problem this way, the original constraints $g_i(\omega) = 0$ are no longer treated explicitly, instead they are embedded into the Lagrangian.

At this point in order to find the maximum or minimum of a function $f(\omega)$ subject to the equality constraints $g_j(\omega) = 0$, we just need to find the **stationary** points of $L(\omega, \beta)$ considered as a function of ω and the Lagrange multipliers β_j. This means that optimal solutions are determined by setting the partial derivatives of $L(\omega, \beta)$ with respect to ω and β_j to zero:

$$\frac{\partial L(\omega, \beta)}{\partial \omega} = 0 \qquad \frac{\partial L(\omega, \beta)}{\partial \beta_j} = 0$$

By solving these equations simultaneously, we obtain the values of ω and β that correspond to the optimal solution of the constrained problem.

Note: The method of Lagrange multipliers is foundational in optimization theory and serves as the basis for more advanced frameworks. A notable generalization is provided by the KKT conditions (that we will see soon), which extends the applicability of Lagrange multipliers to problems involving inequality constraints.

Lagrange Duality & KKT Conditions

In this section, we extend our discussion to constrained optimization problems which may have **inequality constraints** as well as equality constraints. In this part we will focus on taking directly the results of the **Lagrange Duality** (I also took in consideration Andrew Ng notes to derive it), so that we will could then apply to our optimal margin classifier's optimization problem.

Consider a constrained optimization problem of the form:

$$\min_{\omega} f(\omega)$$

$$s.t. \begin{cases} h_i(\omega) \leq 0, & \forall i = 1, \dots, k \\ g_i(\omega) = 0, & \forall i = 1, \dots, l \end{cases}$$

which as we can see involves both inequality constraints $h_i(\omega) \leq 0$ and equality constraints $g_i(\omega) = 0$. Let's call this constrained problem the **primal optimization problem**.

To address such a constrained problem, we generalize the **method of Lagrange multipliers** by introducing a Lagrangian function that incorporates both types of constraints. The Lagrangian is defined as:

$$L(\omega, \alpha, \beta) = f(\omega) + \sum_{i=1}^{k} \alpha_i \cdot h_i(\omega) + \sum_{j=1}^{l} \beta_j \cdot g_j(\omega)$$

where $\alpha_i \geq 0$ and β_i are the **Lagrange multipliers** introduced to penalize respectively the inequality and the equality constraints.

Now, for a fixed ω, let's consider the quantity:

$$\mathcal{P}(\omega) = \max_{\alpha,\beta} L(\omega, \alpha, \beta)$$

The behavior of $\mathcal{P}(\omega)$ depends on whether ω satisfies the primal constraints:

- If ω violates any of the primal constraints (i.e., $h_i(\omega) > 0$ for some i or $g_i(\omega) \neq 0$ for some j), then the terms $\alpha_i \cdot h_i(\omega)$ and $\beta_j \cdot g_j(\omega)$ can grow arbitrarily large as $\alpha_i \to +\infty$ or $\beta_j \to +\infty$ respectively, resulting in $\mathcal{P}(\omega) = +\infty$.
- On the other hand, if all the primal constraints are satisfied for a particular value of ω, since all terms $\alpha_i \cdot h_i(\omega)$ will be negative or at least zero and all terms $\beta_j \cdot g_j(\omega)$ will be zero, we will have that $\mathcal{P}(\omega) = f(\omega)$.

Thus, it can be written as:

$$\mathcal{P}(\omega) = \begin{cases} f(\omega), & if\ \omega\ satisfies\ all\ primal\ constraints \\ +\infty, & otherwise \end{cases}$$

Now if we consider the minimization problem

$$\min_{\omega} \mathcal{P}(\omega) = \min_{\omega} \max_{\alpha,\beta} L(\omega, \alpha, \beta)$$

we see that it is the same problem (i.e., and has the same solutions as) our original, **primal problem**.

Therefore, minimizing $\mathcal{P}(\omega)$ over all ω is equivalent to solving the primal optimization problem and in this case $\mathcal{P}(\omega)$ represents the so called the **primal function**.

Now, let's look at a slightly different problem. For a fixed set of α and β let's consider the quantity:

$$\mathcal{D}(\alpha) = \min_{\omega} L(\omega, \alpha, \beta)$$

You can notice that while for $\mathcal{P}(\omega)$ we were optimizing (maximizing) with respect to α and β, here in $\mathcal{D}(\alpha)$ we are minimizing with respect to ω. We call $\mathcal{D}(\alpha)$ the **dual function**.

We can now pose the **dual optimization problem**:

$$\max_{\alpha,\beta} \mathcal{D}(\alpha) = \max_{\alpha,\beta} \min_{\omega} L(\omega, \alpha)$$

We can notice that this is exactly the same as our primal problem shown above, except that the order of the "max" and the "min" are now exchanged.

Note: In a dual optimization problem, the Lagrange multipliers are also known as **dual variables**.

So, we have found the forms of the primal and the dual problems, but at this point we may ask: how these problems are related?

Let's indicate with p^* the optimal value of the primal problem, i.e. $p^* = \min_{\omega} \mathcal{P}(\omega)$ and with d^* the optimal value of the dual problem, i.e. $d^* = \max_{\alpha,\beta} \mathcal{D}(\alpha)$. As consequence of **minmax inequality** (the "max min" of a function is always less than or equal to the "min max") it holds that:

$$d^* \leq p^*$$

This relationship, called **weak duality**, therefore states that the dual function represents a lower bound on the primal function. The difference $p^* - d^*$ is also known as the optimal **duality gap**.

Moreover, it can be shown that under certain conditions (outlined below) there exists a pair of primal and dual optimal solutions $(\omega^*; \alpha^*, \beta^*)$ such that the following holds:

$$d^* = p^*$$

This is known as **strong duality**, meaning that the dual and primal problems are equivalent, and in this case the optimal duality gap is zero. Unlike weak duality, which guarantees only that the dual problem provides a lower bound on the primal problem, strong duality ensures that the dual problem also achieves this bound exactly.

➔ When **strong duality** holds, solving the dual problem is equivalent to solving the primal problem. This is a great result, especially when the dual problem is easier to solve than the primal problem, since it provides a different path that we can follow to reach the optimization solution.

The conditions required for strong duality are that:

- $\boldsymbol{f(\omega)}$ **and** $\boldsymbol{h_i(\omega)}$ **are convex functions**. For example, in the case of $f(\omega)$ this means that for any two points ω_1 and ω_2 in its domain, and for any $t \in [0,1]$, the following holds:
  $$f(t\omega_1 + (1-t)\omega_2) \leq tf(\omega_1) + (1-t)f(\omega_2)$$
  Geometrically, this means that the graph of a convex function lies below or on the straight line connecting any two points on the graph. The same property holds for $h_i(\omega)$.
  Convex functions simplify the analysis and guarantee that local minima are also global minima.
- The equality constraints $\boldsymbol{g_i(\omega)}$ **are affine**. This means that each is both linear and has a constant extra intercept term, i.e. $g_j(\omega) = a_j^T \omega + b_j$. Affine functions ensure that equality constraints do not introduce non-convexities into the problem.
- The inequality constraints $\boldsymbol{h_i(\omega)}$ **are strictly feasible**, i.e. $\exists \omega \ \ s.t. \ \ h_i(\omega) < 0 \ \ \forall i$ (**Slater's condition**). This condition guarantees that the feasible region of the problem is "well-behaved" (i.e., it has a non-degenerate interior).

Moreover, to ensure the that solutions found are optimal $(\omega^*, \alpha^*, \beta^*)$ a set of necessary conditions known as Karush–Kuhn–Tucker conditions (or **KKT conditions**) **must** also be satisfied:

- **Stationary:**
  $$\frac{\partial L}{\partial \omega}(\omega^*, \alpha^*, \beta^*) = 0$$
  This ensures that ω^* is a stationary point of the Lagrangian.
- **Primal Feasibility:**
  $$h_i(x^*) \leq 0, \quad \forall i$$
  $$g_j(\omega^*) = 0, \quad \forall j$$
  This ensures that ω^* lies in the feasible region of the primal problem.

- **Dual Feasibility:**

$$\alpha_i^* \geq 0, \quad \forall i$$

This ensures that the multipliers for violating inequalities are non-negative penalty terms (since negative penalties would not make sense in this context).

- **Complementary Slackness:**

$$\alpha_i^* \cdot h_i(\omega^*) = 0, \quad \forall i$$

This condition states that if a constraint is active (i.e., $h_i(\omega^*) = 0$), then the corresponding Lagrange multiplier must be non-zero ($\alpha_i^* \neq 0$). Conversely, if a constraint is inactive ($h_i(\omega^*) \neq 0$), the corresponding multiplier must be zero ($\alpha_i^* = 0$).

Explanation:

  - A constraint is considered **active** if it is exactly at its limit, i.e., $h_i(\omega^*) = 0$. This means that the solution ω^* lies exactly on the boundary of the feasible region with respect to that constraint. In this case, complementary slackness condition allows the corresponding Lagrange multiplier α_i^* to take on a non-zero value. This reflects the fact that an active constraint directly impacts the optimization problem, and therefore its associated multiplier (representing the penalty for violating that constraint) should assume a non-zero value.
  - On the other hand, a constraint is considered **inactive** if it is strictly satisfied, i.e., $h_i(\omega^*) < 0$. This means that the solution ω^* lies inside the feasible region (not on the boundary) relative to that constraint, and it does not influence the optimal solution. In this case, complementary slackness condition mandates that the associated multiplier α_i^* must be zero. This ensures that the inactive constraint does not contribute to the Lagrangian, as it does not affect the optimal solution.

Thus, complementary slackness ensures that only the **active** constraints, those that are exactly on the boundary of feasibility ($h_i(\omega^*) = 0$), play a role in determining the optimal solution through their associated Lagrange multipliers. Constraints that do not influence the optimal solution because they are inactive ($h_i(\omega^*) < 0$) will have a corresponding multiplier of zero, ensuring they do not artificially affect the Lagrangian.

In essence, complementary slackness ensures that dual variables (Lagrange multipliers) only contribute to the optimization problem when their associated constraints are exactly on the boundary of feasibility.

The KKT conditions provide the necessary conditions that must be met for a solution to be **optimal** in a constrained optimization problem. Moreover, it can be demonstrated that for convex problems (such the one of the strong duality) the KKT conditions are **necessary and sufficient**.

! KKT conditions (just inequalities): The KKT conditions outlined above account for both equality and inequality constraints. However, the KKT framework is also valid for optimization problems that involve **only inequality constraints**. In such cases, the condition involving equality constraints $g_j(\omega^*) = 0, \quad \forall j$ does not apply and should be omitted. Of course, the remaining KKT conditions are still necessary for optimality in problems with inequality constraints only.

8.4 SVM Optimization

Let's move back to our SVM optimization problem.

Previously, we posed the following (primal) optimization problem for finding the hyperplane that maximize the margin M:

$$\min_{\omega,b} \frac{1}{2}\|\omega\|^2$$

$$s.t. \quad y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1 \geq 0 \quad \forall i = 1, \dots, m$$

We can write the constraints as:

$$h_i(\omega) = -\left(y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1\right) \leq 0$$

Which enables us to construct the Lagrangian function for our optimization problem as:

$$L(\omega, b, \alpha) = \frac{1}{2}\|\omega\|^2 - \sum_{i=1}^{m} \alpha_i \left[y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1\right]$$

Note that there're only α_i but no β_j Lagrange multipliers, since the problem has only inequality constraints.

At this point instead of solving the primal problem, we could think to solve the dual problem (that as we will seen will lead a very powerful formulation):

$$\max_{\alpha} \mathcal{D}(\alpha) = \max_{\alpha} \min_{\omega,b} L(\omega, b, \alpha)$$

However, to be sure that solving the dual problem is equivalent to solving the primal problem we need to ensure that all the conditions for strong duality ($d^* = p^*$) are satisfied. Specifically in our case we need to ensure that the objective function and $h_i(\omega)$ are convex functions and that the Slater's condition is respected:

- **Convexity of the Problem**:
  - The objective function $\frac{1}{2}\|\omega\|^2$ is convex because it is quadratic in ω and has no linear or higher-degree terms that could make it non-convex.
  - The constraints $h_i(\omega) = -\left(y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1\right)$ are affine functions, which are linear and therefore convex.

  Since both the objective function and the constraints are convex, we can say that the convexity is satisfied.

- **Slater's condition**: Slater's condition requires that there exists a strictly feasible point (i.e., a point that strictly satisfies the inequality constraints).

  For the SVM problem, we have the inequality constraints:

  $$-\left(y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1\right) \leq 0 \quad \forall i$$

  and in our case Slater's condition requires that there exists some point ω^* and b^* such that:

  $$-\left(y^{(i)}\left(\omega^{*T} \cdot x^{(i)} + b^*\right) - 1\right) < 0 \quad \forall i$$

  This condition is trivially satisfied in our case because we have assumed that the **data is linearly separable**. Given linear separability, we can always find a choice of ω^* and b^* such that the inequalities are strictly satisfied for all data points. Thus, Slater's condition is also satisfied.

With strong duality established, we now need to ensure that the KKT conditions are satisfied for guarantee the optimality of our solutions:

- **Stationarity**:
  We need to find the stationary points of the Lagrangian by taking the partial derivatives of $L(\omega, b, \alpha)$ with respect to ω and b, and setting them equal to zero:

- $\frac{\partial L}{\partial \omega} = \omega^* - \sum_{i=1}^{m} \alpha_i^* y^{(i)} x^{(i)} = 0 \qquad \Rightarrow \boldsymbol{\omega^* = \sum_{i=1}^{m} \alpha_i^* y^{(i)} x^{(i)}}$

  This is a great result because vector ω^* is just a linear combination of the samples $x^{(i)}$ (weighted by the corresponding Lagrange multipliers α_i). Importantly you should start to notice that **NOT ALL** the samples contribute to ω^*, but only the ones which have a Lagrange multipliers $\alpha_i^* \neq 0$ (soon we will see that they will be just the samples lying on the support vectors).

- $\frac{\partial L}{\partial b} = -\sum_{i=1}^{m} \alpha_i^* y^{(i)} = 0 \qquad \Rightarrow \boldsymbol{\sum_{i=1}^{m} \alpha_i^* y^{(i)} = 0}$
  This will be useful later during the calculation.

- **Primal feasibility** is held because it's directly tied to the primal problem's constraints. The optimization process ensures these constraints are respected by the optimal solution. Therefore, the condition $-y^{(i)}\left(\omega^{*T} \cdot x^{(i)} + b^*\right) - 1 \leq 0$ holds for all the samples i, as ensured by the structure of the problem.

- **Dual feasibility** holds because the dual problem is constructed with the condition that the Lagrange multipliers α_i^* must be non-negative. Therefore, $\alpha_i^* \geq 0$ holds for all i, as ensured by the structure of the dual formulation.

- **Complementary Slackness**: The complementary slackness condition requires that:

  $$\alpha_i^* \cdot h_i(\omega^*) = 0, \quad \forall i$$

  which in our case means:

  $$\alpha_i^* \cdot \left[y^{(i)}\left(\omega^{*T} \cdot x^{(i)} + b^*\right) - 1\right] = 0, \quad \forall i$$

  This condition can be explicitly stated as:

  - If the constraint is **active**, i.e., $y^{(i)}\left(\omega^{*T} \cdot x^{(i)} + b^*\right) - 1 = 0$, then the corresponding Lagrange multiplier must be non-zero value, i.e. $\alpha_i^* \neq 0$.
  - If the constraint is **inactive**, i.e., $y^{(i)}\left(\omega^{*T} \cdot x^{(i)} + b^*\right) - 1 < 0$, then the corresponding Lagrange multiplier must be zero, i.e., $\alpha_i^* = 0$.

  Now from the stationarity condition, we found that:

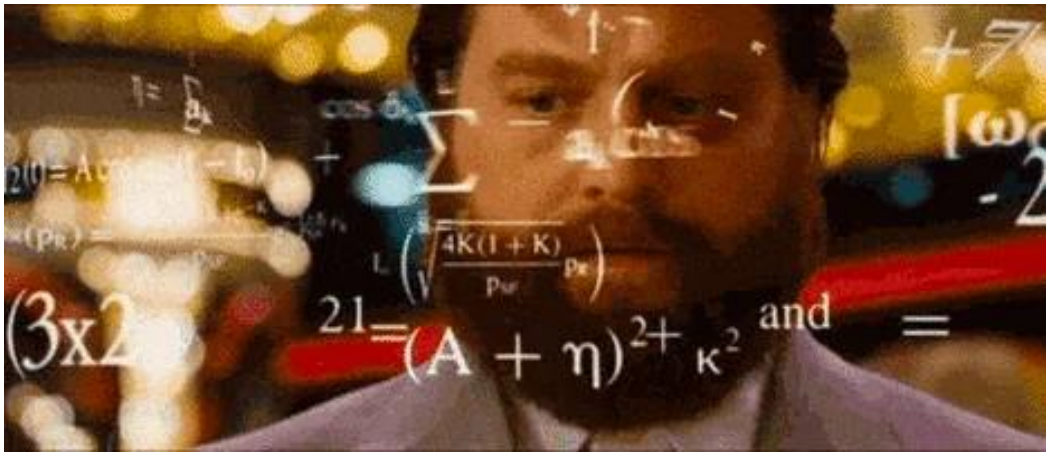
  $$\omega^* = \sum_{i=1}^{m} \alpha_i^* y^{(i)} x^{(i)}$$

  and this result has indicated to us that **not all data points** contribute to ω^* (and therefore to the construction of the optimal hyperplane), but only the ones which have a Lagrange multipliers $\alpha_i^* \neq 0$. Therefore, to respect the complementary slackness condition we are constrained to consider just the samples for which $y^{(i)}\left(\omega^{*T} \cdot x^{(i)} + b^*\right) - 1 = 0$ since they are the only ones that ensure $\alpha_i^* \neq 0$. But the samples for which $y^{(i)}\left(\omega^{*T} \cdot x^{(i)} + b^*\right) - 1 = 0$ are exactly the samples that lie on the **support vectors** and so they are the only ones that actively contribute to the determination of the optimal hyperplane.

  Therefore, what can be stated is:

  > **"The only samples that contribute to the minimization of the Lagrangian function and consequently to the determination of the optimal hyperplane are the samples that lie on the support vectors. As a result, the corresponding Lagrange multipliers α_i^* for these samples will be non-zero, while for all other samples that do not belong to the support vectors, the corresponding α_i^* will be zero."**

Me at this point:



We are almost there! Now, let's further simplify the formulation to derive the form of the problem we are ultimately interested in — one that is expressed only in terms of Lagrange multipliers α.

We first substitute the expression for ω^* that we derived earlier into the original Lagrangian function:

$$\frac{1}{2}\|\omega\|^2 - \sum_{i=1}^{m} \alpha_i \left[y^{(i)} \left(\omega^T \cdot x^{(i)} + b \right) - 1 \right] =$$

$$= \frac{1}{2} \left(\sum_{i=1}^{m} \alpha_i^* y^{(i)} x^{(i)} \right) \cdot \left(\sum_{j=1}^{m} \alpha_j^* y^{(j)} x^{(j)} \right) - \left(\sum_{i=1}^{m} \alpha_i^* y^{(i)} x^{(i)} \right) \cdot \left(\sum_{j=1}^{m} \alpha_j^* y^{(j)} x^{(j)} \right) - b \sum_{i=1}^{m} \alpha_i^* y^{(i)} + \sum_{i=1}^{m} \alpha_i^*$$

From earlier, we also derived that $\sum_{i=1}^{m} \alpha_i^* y^{(i)} = 0$, which implies that the term involving b vanishes.

$$= \frac{1}{2} \left(\sum_{i} \alpha_i^* y^{(i)} x^{(i)} \right) \cdot \left(\sum_{j} \alpha_j^* y^{(j)} x^{(j)} \right) - \left(\sum_{i} \alpha_i^* y^{(i)} x^{(i)} \right) \cdot \left(\sum_{j} \alpha_j^* y^{(j)} x^{(j)} \right) - b \underbrace{\sum_{i} \alpha_i^* y^{(i)}}_{0} + \sum_{i} \alpha_i^* =$$

$$= -\frac{1}{2} \left(\sum_{i} \alpha_i^* y^{(i)} x^{(i)} \right) \cdot \left(\sum_{j} \alpha_j^* y^{(j)} x^{(j)} \right) + \sum_{i} \alpha_i^* =$$

$$= \sum_{i} \alpha_i^* - \frac{1}{2} \sum_{i} \sum_{j} \alpha_i^* \alpha_j^* y^{(i)} y^{(j)} \, x^{(i)} \cdot x^{(j)} = \sum_{i} \alpha_i^* - \frac{1}{2} \sum_{i} \sum_{j} \alpha_i^* \alpha_j^* y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle$$

This gives us a simplified formulation for our **dual function** that is expressed only in terms of Lagrange multipliers α (and not anymore on the other parameters ω and b):

$$\mathcal{D}(\alpha) = \sum_{i} \alpha_i^* - \frac{1}{2} \sum_{i} \sum_{j} \alpha_i^* \alpha_j^* y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle$$

More importantly this formulation reveals that the optimization depends only on the **inner products** between pairs of samples $\langle x^{(i)}, x^{(j)} \rangle$ (obviously just the one lying on the support vectors). This is extremely important since it will make computationally efficient and pivotal incorporate kernel methods as required from Kernel SVM (that we will see soon).

The **dual optimization problem** is therefore defined as:

$$\max_{\alpha} \mathcal{D}(\alpha) = \max_{\alpha} \left(\sum_i \alpha_i^* - \frac{1}{2} \sum_i \sum_j \alpha_i^* \alpha_j^* y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle \right)$$

$$s.t. \quad \alpha_i^* \geq 0 \quad and \quad \sum \alpha_i^* y^{(i)} = 0 \quad \forall i, j = 1, \dots, m$$

The solution to this dual problem provides the optimal α_i^*. Practical algorithms like **Sequential Minimal Optimization (SMO)** can be used for efficient computation of α_i^*, however we won't cover the details here (if interested, refer to Andrew Ng's notes for a detailed explanation).

Then, once the α_i^* values have been obtained (e.g. trough SMO algorithm), we can recover the equation of the hyperplane $\omega \cdot x + b$ and substitute ω^* that we have derived:

$$\omega^* \cdot x + b^* = \sum_i \alpha_i^* y^{(i)} x^{(i)} \cdot x + b^* = \sum_i \alpha_i^* y^{(i)} \langle x^{(i)}, x \rangle + b^*$$

where, obviously, only the samples lying on the support vectors (samples for which $\alpha_i \neq 0$) contribute to the sum. This gives us the equation of optimal hyperplane, i.e. the separation **hyperplane with maximum margin**.

To fully define the optimal hyperplane, we also need to calculate the optimal intercept term b^*. The intercept term is the distance from the origin to the hyperplane, and it ensures that the hyperplane is positioned correctly between the two classes. There are several methods to compute it, and we will discuss two common approaches:

1. The first approach to calculate b^* is by using any data point $x^{(i)}$ lying on one of the two support vector and then solve for b^* using:

   $$b^* = y^{(i)} - \omega^* \cdot x^{(i)}$$

   This approach directly computes b^* based on a single support vector.

2. A second and more accurate approach is to calculate b^* as follows:

   $$b^* = -\frac{\max\limits_{i:y^{(i)}=-1} \omega^* \cdot x^{(i)} + \min\limits_{i:y^{(i)}=1} \omega^* \cdot x^{(i)}}{2}$$

   Here:

   - $\max\limits_{i:y^{(i)}=-1} \omega^* \cdot x^{(i)}$ is the largest value of $\omega^* \cdot x^{(i)}$ for all samples lying on the support vector where $y^{(i)} = -1$, i.e., the support vector of the negative class.
   - $\min\limits_{i:y^{(i)}=1} \omega^* \cdot x^{(i)}$ is the smallest value of $\omega^* \cdot x^{(i)}$ for all samples lying on the support vector where $y^{(i)} = 1$, i.e., the support vector of the positive class.

   The reason for this formulation is that the margin between the two classes should be symmetric around the hyperplane, so by taking the maximum and minimum dot products, we are determining the position of the hyperplane such that it is equidistant from the support vectors on both sides. Therefore, this approach takes into consideration both the two support vectors ensuring that the margin is symmetric with respect to the hyperplane. I suggest you use this one.

SVM Optimization Recap:

We began by transforming the constrained primal optimization problem:

$$\min_{\omega,b} \frac{1}{2}\|\omega\|^2$$

$$s.t. \quad y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1 \geq 0 \quad \forall i = 1, \ldots, m$$

in an unconstrained one using the Lagrange multipliers. The Lagrangian is defined as:

$$L(\omega, b, \alpha) = \frac{1}{2}\|\omega\|^2 - \sum_{i=1}^{m} \alpha_i \left[y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1\right]$$

Next, we ensured that **strong duality** and the **Karush-Kuhn-Tucker (KKT)** conditions were satisfied. This guaranteed two things respectivelly:

1. Solving the dual problem is equivalent to solving the primal problem.
2. The solutions obtained from the dual problem are optimal.

During this derivation process, we observed the following:

- $\boldsymbol{\alpha_i^* = 0}$ for all samples that do not lie on the support vectors.
- $\boldsymbol{\omega^* = \sum_{i=1}^{m} \alpha_i^* y^{(i)} x^{(i)}}$ which implies that ω^* is a linear combination of just the sample lying on the support vectors (since they are the only ones that have $\boldsymbol{\alpha_i^* \neq 0}$).

We arrived at the following expression for the **dual function**:

$$\mathcal{D}(\alpha) = \sum_i \alpha_i^* - \frac{1}{2}\sum_i \sum_j \alpha_i^* \alpha_j^* y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle$$

This showed that the optimization depends **only on the inner products** between pairs of samples, $\langle x^{(i)}, x^{(j)} \rangle$, that constitute the support vectors. This makes it computationally efficient for kernel methods.

The dual optimization problem is therefore defined as:

$$\boldsymbol{\max_{\alpha} \mathcal{D}(\alpha) = \max_{\alpha} \left(\sum_i \alpha_i^* - \frac{1}{2}\sum_i \sum_j \alpha_i^* \alpha_j^* y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle \right)}$$

$$\boldsymbol{s.t. \quad \alpha_i^* \geq 0 \quad and \quad \sum \alpha_i^* y^{(i)} = 0}$$

By solving this maximization (e.g., using the SMO algorithm), we determine the optimal dual variables α_i^*.

Given these optimal dual variables α_i^*, the separation **hyperplane with maximum margin** can be obtained as:

$$\omega^* \cdot x + b^* = \sum_i \alpha_i^* y^{(i)} \langle x^{(i)}, x \rangle + b^*$$

where the intercept term b^* can be computed as:

$$b^* = -\frac{\max\limits_{i:y^{(i)}=-1} \omega^* \cdot x^{(i)} + \min\limits_{i:y^{(i)}=1} \omega^* \cdot x^{(i)}}{2}$$

ensuring the hyperplane is symmetrically positioned between the two classes.

8.5 SVM Cost Function

The cost function of SVM is quite similar to that of Logistic Regression, but with key differences in how it handles classification.

Hypothesis in Logistic Regression vs. SVM

In Logistic Regression, the hypothesis is computed by applying the **sigmoid function** to the raw model output $\theta^T x$. The sigmoid function then maps this value to a probability, which can be interpreted as the likelihood of a sample belonging to the positive class (y = 1):

$$h_\theta(x) = \sigma(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

In contrast, the **hypothesis for SVM** is much simpler. SVM directly predicts the class label based on the raw output $\theta^T x$ (here we use θ^T instead of ω^T just to have a same notation for both). If $\theta^T x \geq 0$, it predicts $y = 1$ (positive class), otherwise, it predicts $y = 0$ (negative class). This can be written as:

$$h_\theta(x) = \begin{cases} 1, & if\ \theta^T x \geq 0 \\ 0, & otherwise \end{cases}$$

Cost Function in Logistic Regression vs. SVM

Remember that the cost function for logistic regression is given by the negative log-likelihood:

$$-\big(y \log h_\theta(x) + (1-y) \log\big(1 - h_\theta(x)\big)\big) =$$

$$= -y \log \frac{1}{1 + e^{-\theta^T x}} - (1-y) \log\left(1 - \frac{1}{1 + e^{-\theta^T x}}\right)$$

We can split the contribution of the cost function separately for $y = 1$ and $y = 0$.

- **Loss for $y = 1$**: $-\log \frac{1}{1+e^{-\theta^T x}}$
- **Loss for $y = 0$**: $-\log\left(1 - \frac{1}{1+e^{-\theta^T x}}\right)$

On the other hand, the SVM cost function can be written as:

$$cost(h_\theta(x), y) = \begin{cases} max(0, 1 - \theta^T x), & if\ y = 1 \\ max(0, 1 + \theta^T x), & if\ y = 0 \end{cases}$$

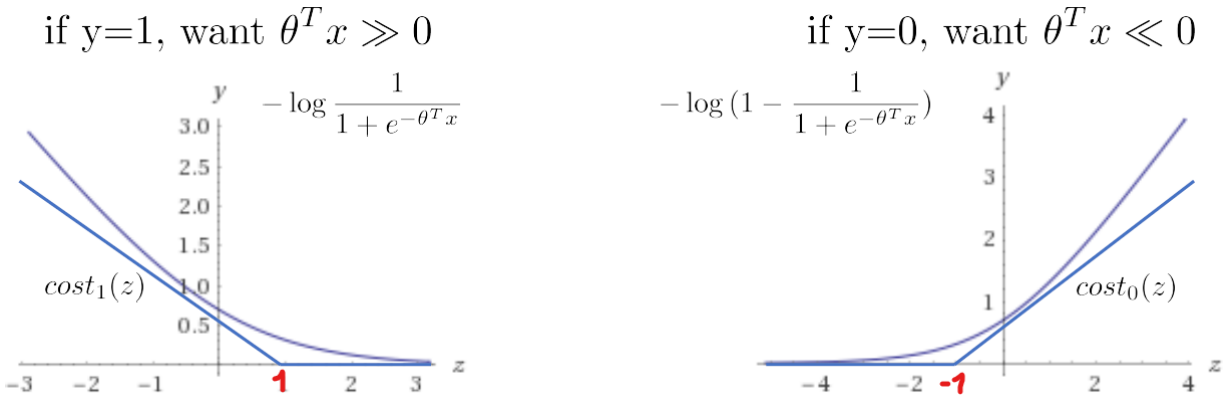
This cost function applies a margin-based penalty to misclassified points and points that are within the margin (where $0 < \theta^T x < 1$).

- **Loss for $y = 1$**:
  - If $\theta^T x \geq 1$, there is no penalty, meaning the model is confidently predicting the correct class (no cost).
  - If $\theta^T x < 1$, the penalty increases as $\theta^T x$ moves further away from 1 (decreased). This is because the model's prediction is either incorrect or falls within the margin.

Note: Why does the cost start to increase from 1 instead of 0? Because as we saw above in SVM we consider a margin to be defined by our support vectors which we spaced by 1.

- **Loss for $y = 0$:**
  - Similarly, if $\theta^T x \leq -1$, there is no penalty for a correct prediction.
  - If $\theta^T x > -1$, the penalty increases as $\theta^T x$ moves further away from -1 (increases). This is because the model's prediction is either incorrect or falls within the margin.

In the plots below are shown the cost function terms for $y = 1$ and $y = 0$ separately, where the purple line is the cost function of Logistic Regression, and the blue line is for SVM. Please note that the X-axis here is the raw model output, $\theta^T x$ (z).



You can notice that:

- **When $y = 1$**:
  - Logistic Regression's cost decreases smoothly as $\theta^T x$ increases.
  - SVM's cost is **piecewise linear**: it drops to zero directly when $\theta^T x \geq 1$.
- **When $y = 0$**:
  - Logistic Regression again decreases smoothly.
  - SVM drops to zero only directly when $\theta^T x \leq -1$.

So, basically, we can observe that an SVM cost function seeks to approximate the logistic function with a piecewise linear.

For a dataset with m samples, the overall cost function for SVM can be expressed as:

$$J(\theta) = \sum_{i=1}^{m} y^{(i)} cost_1\left(\theta^T x^{(i)}\right) + \left(1 - y^{(i)}\right) cost_0\left(\theta^T x^{(i)}\right) =$$

$$= \sum_{i=1}^{m} y^{(i)} max\left(0, 1 - \theta^T x^{(i)}\right) + \left(1 - y^{(i)}\right) max\left(0, 1 + \theta^T x^{(i)}\right)$$

We can also add regularization to SVM. For example, adding L2 regularized term to SVM, the cost function becomes:

$$\boldsymbol{J(\theta) = C\left[\sum_{i=1}^{m} y^{(i)}\, cost_1\left(\theta^T x^{(i)}\right) + \left(1 - y^{(i)}\right) cost_0\left(\theta^T x^{(i)}\right)\right] + \frac{1}{2}\sum_{j=1}^{n} \theta_j^2}$$

where m is the number of samples and n is the number of features.

Here there is a comparison between Logistic regression and SVM cost functions:

Logistic regression:

$$\min_{\theta} \frac{1}{m}\left[\sum_{i=1}^{m}\left(y^{(i)} \log h_\theta(\mathbf{x^{(i)}}) + (1 - y^{(i)}) \log\left(1 - h_\theta(\mathbf{x^{(i)}})\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

Support vector machine:

$$\min_{\theta}\ \mathrm{C}\sum_{i=1}^{m}\left[y^{(i)} cost_1(\theta^T \mathbf{x^{(i)}}) + (1 - y^{(i)}) cost_0(\theta^T \mathbf{x^{(i)}})\right] + \frac{1}{2}\sum_{j=1}^{n} \theta_j^2$$

Different from Logistic Regression which use $\boldsymbol{\lambda}$ as the parameter in front of regularized term to control the weight of regularization, correspondingly, SVM uses $\boldsymbol{C}$ in front of fit term. Intuitively, the fit term emphasizes fit the model very well by finding optimal coefficients.

C in the SVM cost function plays a critical role:

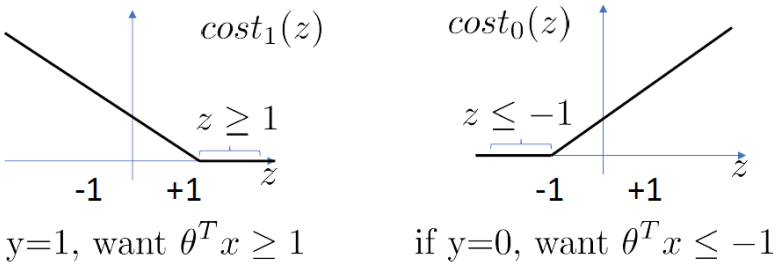
- A **large $\boldsymbol{C}$** (similar to no regularization) prioritizes fitting the training data well, making the model sensitive to outliers and potentially overfitting.
- A **small $\boldsymbol{C}$** enforces stronger regularization, encouraging a larger margin which might generalize better on unseen data.

In this sense, C plays a role similar to $1/\lambda$ in regularized Logistic Regression.

At the end we get for SVM:

Support vector machine:

$$\min_{\theta}\ \mathrm{C}\sum_{i=1}^{m}\left[y^{(i)} cost_1(\theta^T \mathbf{x^{(i)}}) + (1 - y^{(i)}) cost_0(\theta^T \mathbf{x^{(i)}})\right] + \frac{1}{2}\sum_{j=1}^{n} \theta_j^2$$

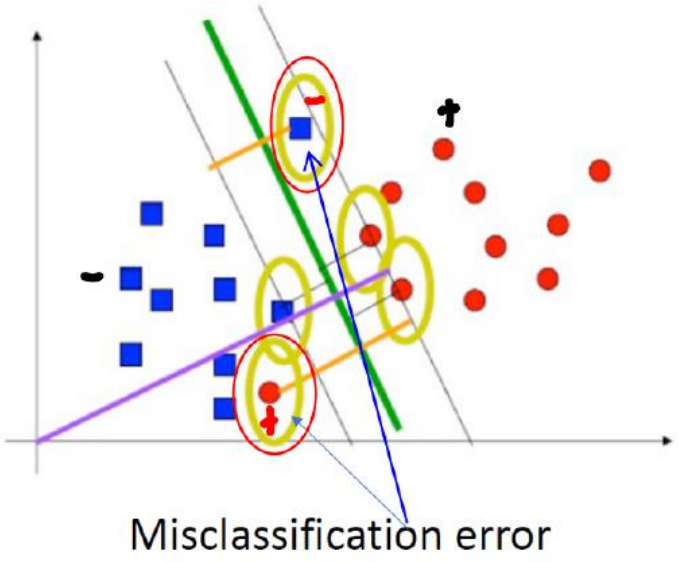


if y=1, want $\theta^T x \geq 1$ if y=0, want $\theta^T x \leq -1$

8.6 Soft-Margin SVM

Hard margin SVM seeks to find an optimal hyperplane that perfectly separates classes while maximizing the margin between them. However, real-world datasets are often not perfectly linearly separable and, in such cases, hard margin SVM will fail to find a valid solution and will lead to **misclassification errors**.

In this situation it is possible to carry out a separation of the classes through a linear hyperplane only by accepting that, after having determined the separating hyperplane, some patterns of the training set with positive label are classified as negative and vice versa. We must accept, therefore, that some constraints are **violated**.

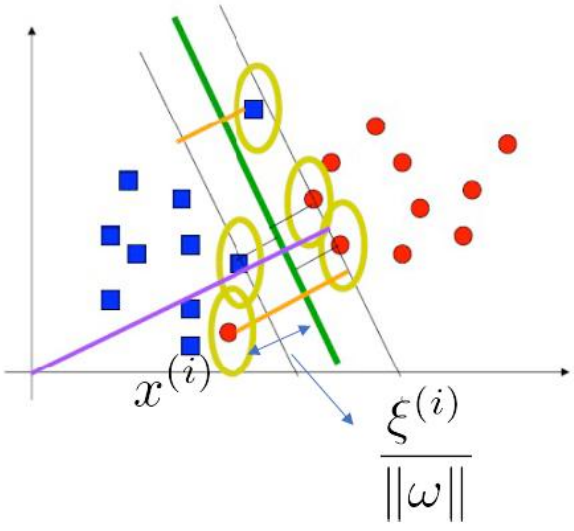


Soft Margin SVM introduces flexibility by allowing some margin violations (misclassifications) to handle cases where the data is not perfectly separable. To do it introduces a penalty term for misclassifications, allowing for a trade-off between a wider margin and a few misclassifications.

More specifically:

- A **slack** variable ξ_i is introduced for each constraint, in order to allow an **error tolerance**:
$$y^{(i)}\left(\omega^T x^{(i)} + b\right) \geq 1 - \xi^{(i)}$$
- An additional term C is introduced in the cost function to **penalize misclassification** errors:
$$\frac{1}{2}\|\omega\|^2 + C\sum_{i=1}^{m}\xi^{(i)}$$

$\xi^{(i)}$s are cost variables proportional to how much the anomalous point deviates from the hyperplane, they represent the distance from the optimal hyperplane, therefore $\xi^{(i)} > 1$ indicates a misclassification error.



$\boldsymbol{C}$ (**regularization** parameter) is used to control the trade-off between the complexity of the hypothesis space and the admissible number of errors (number of non-separable examples). A big value for C gives a stronger penalization to errors.

So, the optimization problem to solve becomes:

$$\min_{\omega,b}\left(\frac{1}{2}\|\omega\|^2 + C\sum_{i=1}^{m}\xi^{(i)}\right)$$

$$s.t. \quad y_i\left(\omega^T x^{(i)} + b\right) \geq 1 - \xi^{(i)}$$

And the dual problem becomes:

$$\max_{\alpha}\left(\sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y^{(i)}y^{(j)}\langle x^{(i)}, x^{(j)}\rangle\right)$$

$$s.t. \quad 0 \leq \alpha_i \leq \boldsymbol{C} \quad and \quad \sum_{i}^{m}\alpha_i y^{(i)} = 0$$

Notice that the dual variables α_i are now bound with C.

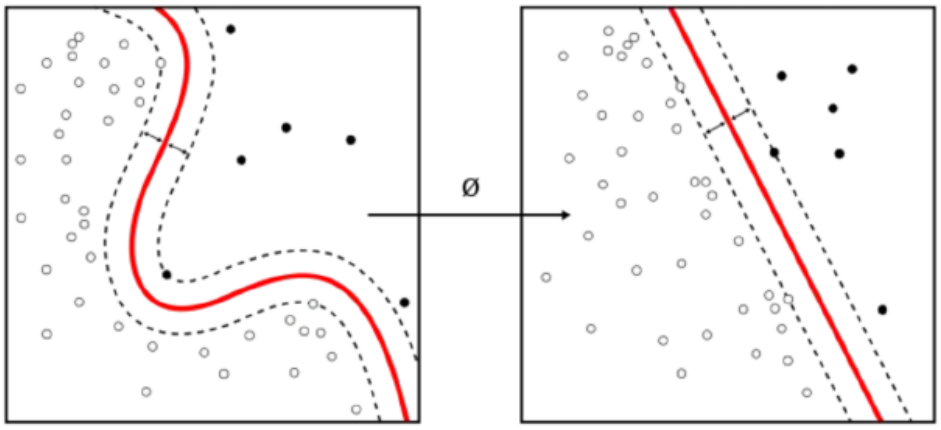
However, the proposed solution could not be enough. It does not perform well with samples distributed with a complex fashion in a non-linearly separable space since a hyperplane can only represent a dichotomy (rigid division into two parts) in the space of instances/patterns. This is where **Kernel SVM** comes into action.

8.7 Kernel SVM

Looking back at what we've derived, it is clear that we are only using a linear hyperplane defined as $\omega^T x^{(i)} + b$. That means SVM performs best **when the data is linearly separable**, and this assumption can be a significant limitation when dealing with non-linear data.

Cover's theorem on the separability of patterns states that "*A complex non-linear pattern-classification problem cast (transformed) in a high-dimensional space is more likely to be linearly separable than in a lowdimensional space*".

So, the idea is to map the data into a higher-dimensional space where the datapoints become linearly separable making so possible to apply a simple linear hyperplane to bisect the two classes.

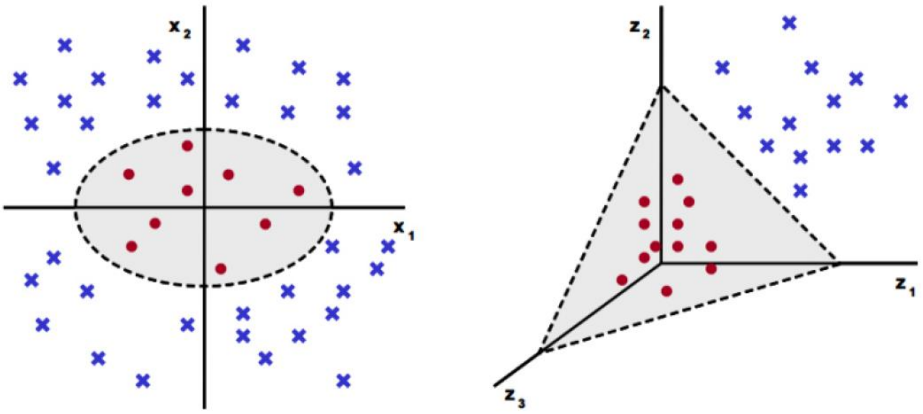


To better illustrate this, consider the following example shown in the Figure below: we have data points in a **2-dimensional space** defined by the features x_1 and x_2. In this original 2D space, it's impossible to separate the two classes with a linear hyperplane because the data is not linearly separable.

However, we can **engineer new features** from the existing ones, e.g. x_1^2, $\sqrt{2}x_1x_2$ and x_2^2, effectively transforming the data into a **3-dimensional space**. We can see that in this higher-dimensional space, the data becomes linearly separable, allowing us to use a simple linear hyperplane to distinguish the two classes.

$$\varphi : \mathbb{R}^2 \to \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) = \left(x_1^2, \sqrt{2}x_1x_2, x_2^2\right)$$



This demonstrates a powerful idea: by mapping the data to a more convenient higher-dimensional space using a **transformation function**, we can simplify the problem and make it solvable with linear methods.

Now, suppose we want to consider all possible interactions between two input features, including quadratic terms. For example, starting with two features x_1 and x_2, we can engineer new features as follows:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \to \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix}$$

This transformation takes the data from a **2-dimensional space** to a **6-dimensional space**. Similarly, if we include cubic interactions, the dimensionality of the transformed space will increase even further.

In general, such transformations can be written compactly as:

$$x \to \Phi(x)$$

So, here Φ is a transformation function that we apply on the original input data that maps them into a higher-dimensional space.

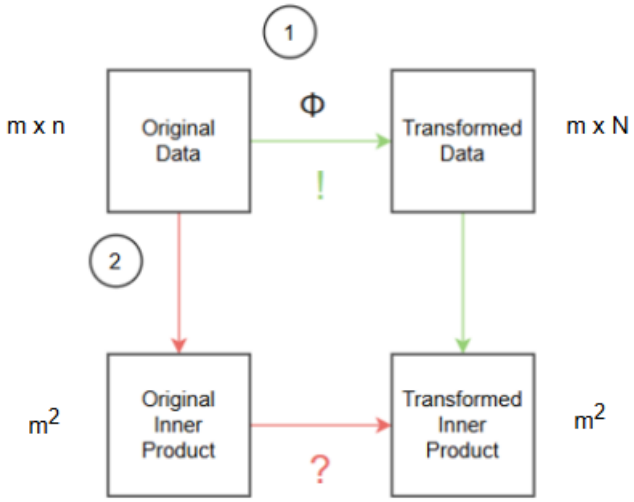
We saw that if we represent SVM problem in a dual form all we need to care about is the inner product of our data. So, what we can think to do is equivalently to work directly with the transformed data so that we still care only about the inner product between the transformed data:

$$\Phi\left(x^{(i)}\right) \cdot \Phi\left(x^{(j)}\right)$$

and then run our SVM regularly.

Kernel Trick

Look at the diagram shown in the Figure below to better understand at a high-level what we are doing.



We begin with our **original data** (at top-left box) and we aim to compute the **transformed inner product** (bottom-right box).

What we saw above is the Path 1: we first applied the transformation Φ to the data (e.g., mapping the data from a 2-dimensional space to a 6-dimensional space) and then we calculated the inner product of the transformed data so that we obtain the transformed inner product. While this method works, it poses a major challenge: as the dimensionality increases, the size of the transformed data grows rapidly. While go from 2 to 6-dimensional space seems not to be a huge deal, you can imagine that if you consider more and more interactions the new size of the space can grow very fast. This can lead to inefficiencies, particularly in terms of storage and computation.

For example, if the original data has m samples and n features (e.g., $n = 2$), the transformed data could have N features, where N becomes very large due to high-order interactions. Storing and processing such a large dataset is both time- and space-intensive.

So, here we passed from our original data that is $m \times n$ (m observations in n features) to transformed data that are $m \times N$, and the N that is very large will cause inefficiencies.

At this point we ask: "Is there a more efficient path to get from the original data to the transformed inner product?"

Yes, there is a more efficient alternative (Path 2). Instead of directly transforming the data, first we compute the inner product of the original data, i.e., $x^{(i)} \cdot x^{(j)}$, resulting in an $m \times m = m^2$ matrix (this because we make the inner product between each of the m data point vectors with every other of the m data point vectors) and then, we apply the transformation to these inner products, that still is m^2 (that's because in transforming the data we are not adding any new sample, so they are always that one's but just transformed).

Why Is This More Efficient?

- The naive approach would result in a transformed dataset of size $m \times N$, where N could be extremely large.
- By first computing the inner product in the original space, we deal with an $m \times m$ matrix. This is computationally more efficient, especially when $N \gg m$.

! By the way it is fundamental to notice that this efficiency is only possible if there exist a special function that can get us from the original inner product $x^{(i)} \cdot x^{(j)}$ directly to the transformed inner product $\Phi(x^{(i)}) \cdot \Phi(x^{(j)})$, allowing us to bypass explicitly computing the transformation Φ.

Let's do an experiment to see if we can get it.

Let's revisit the earlier example with quadratic transformations. In this case our goal is to obtain the transformed inner product $\Phi(x^{(i)}) \cdot \Phi(x^{(j)})$ directly from the original inner product, $x^{(i)} \cdot x^{(j)}$, without explicitly performing the transformation Φ. But before doing this let's see what transformed inner product we would get by following the first approach with the explicit transformation:

$$\Phi(x^{(i)}) \cdot \Phi(x^{(j)}) = 1 + x_1^{(i)}x_1^{(j)} + x_2^{(i)}x_2^{(j)} + x_1^{(i)}x_1^{(j)}x_2^{(i)}x_2^{(j)} + \left(x_1^{(i)}\right)^2\left(x_1^{(j)}\right)^2 + \left(x_2^{(i)}\right)^2\left(x_2^{(j)}\right)^2$$

This result provides the six terms that define the inner product in the transformed space. Now we want to reach the same result, so obtain the same six terms, but without doing the explicit transformation following the second approach.

Let's consider a "magical" function defined as:

$$K_{Poly(2)}(x^{(i)} \cdot x^{(j)}) = (1 + x^{(i)} \cdot x^{(j)})^2$$

Notice that this is basically just a function of the inner product of the original data, so it doesn't perform any transformation on the original data itself!

Expanding this we get:

$$= \left(1 + x_1^{(i)}x_1^{(j)} + x_2^{(i)}x_2^{(j)}\right)^2 =$$

$$= 1 + x_1^{(i)}x_1^{(j)} + x_2^{(i)}x_2^{(j)} + x_1^{(i)}x_1^{(j)}x_2^{(i)}x_2^{(j)} + \left(x_1^{(i)}\right)^2\left(x_1^{(j)}\right)^2 + \left(x_2^{(i)}\right)^2\left(x_2^{(j)}\right)^2$$

Looking what we got … we arrived at the same six terms as the explicit transformation.

Let's reflect on what this means for our problem. We've discovered a more efficient way where we have not sent our data in a higher dimensional space, but we kept all the benefits as like we sent our data in a higher dimensional space. In particular we were able to do it by using particular types of functions called Kernel functions, so that we can simply run the kernel function up to the inner product of the original data and get the same result of doing inner product of the transformations of the data but without the computational burden of performing the actual transformation.

This might seem almost magical, but it works—and this approach is known as the **kernel trick**: instead of computing explicitly the transformation $\Phi(x)$, which challenges because the learning algorithm must handle high-dimensional vectors, we can make a **projection** into the higher-dimensional space **implicitly**, using specialized functions called **kernel functions**.

A **kernel function** is any function that, by operating solely on the inner product of the original data points, computes the equivalent of the inner product in the transformed (higher-dimensional) space:

$$\boldsymbol{k(x^{(i)}, x^{(j)}) = \Phi(x^{(i)}) \cdot \Phi(x^{(j)})}$$

Crucially, this is done without ever explicitly constructing or interacting with the transformed data itself. This makes the kernel trick both powerful and efficient, enabling SVMs to handle complex, non-linear relationships in the data while avoiding the computational pitfalls of high-dimensional transformations. The explicit form of Φ is therefore ignored.

So, we can apply this to our starting dual optimization problem. By simply choosing a kernel function such that:

$$k(x^{(i)}, x^{(j)}) = \Phi(\vec{x}^{(i)}) \cdot \Phi(\vec{x}^{(j)})$$

the optimization problem simply becomes:

$$\max_{\alpha} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y^{(i)} y^{(j)} \boldsymbol{k}\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right)$$

$$s.t. \quad 0 \leq \alpha_i \leq C \quad and \quad \sum_{i}^{m} \alpha_i y^{(i)} = 0$$

Additional Observations:

Unlike in neural networks where the non-linearities are not introduced in the model (via activation functions), here the non-linearities are introduced directly in the transformations of the parameter space, in fact:

1. Patterns (input space) are mapped into a space with (much) higher dimension (feature space) through kernel functions;

2. The optimal hyperplane is defined within this feature space.

Kernel Functions

There are several kernel functions used for SVMs. Some of the most popular ones are:

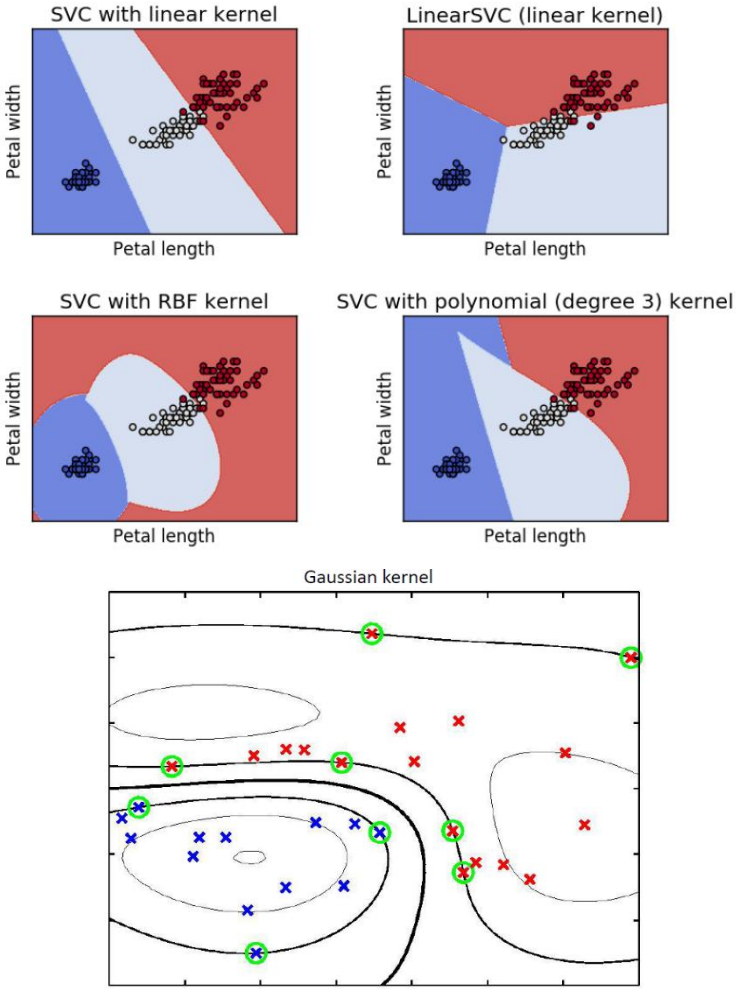
- Linear kernel
  $(\vec{u} \cdot \vec{v})$

- Polinomial kernel of degree d (this is the one that we saw in the example above with $r = 1$ and $d = 2$)
  $(\vec{u} \cdot \vec{v} + r)^d$

- Multi-layer Perceptron tanh
  $tanh\,(\gamma(\vec{u} \cdot \vec{v}) + r)$

- Radial basis function (RBF) kernel
  $e^{-\gamma \frac{\|x^{(i)} - x^{(j)}\|}{\sigma}}$

- Gaussian Radial basis function kernel
  $e^{-\frac{\left(x^{(i)} - x^{(j)}\right)^2}{2\sigma^2}}$



Parameters tuning

To use Support Vector Machines you have to define:

- the kernel function;
- potential parameters of the kernel function;
- the value for the regularization parameter C.

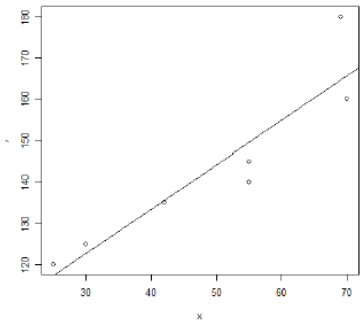
General rules for the set up do not exist, but you should make your choice on a validation set, usually through **cross validation**.

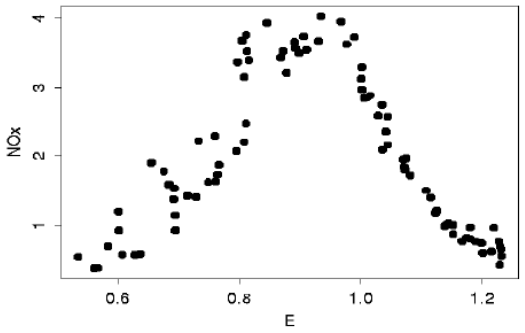
Advantages of SVMs

- There are **no local** minima, just a single global minimum (the optimization problem is quadratic-> ∃! optimal solution)
- The optimal solution can be found in polynomial time.
- There are few parameters to set up (C, type of kernel and specific kernel parameters)
- Solution is stable (ex. there is no problem of randomly initializing of weights just as in Neural Networks)
- Solution is sparse: it just involves support vectors.

8.8 Generalized Linear Models (GLMs)

We have seen how to model linear relationships, where inputs and outputs are linearly related. However, it is usual that our data cannot be approximated by a linear function. What we need is to find a way to model nonlinear relations without increasing too much the complexity of the algorithm.





The main and limiting characteristics of linear regression is the linearity of parameters:

$$h_\theta(x) = \sum_{i=1}^{m} \theta_i \cdot x^{(i)} = \theta^T x$$

$h_\theta(x)$ keeps a linear relation also with the features space X. This linearity represents a limitation of the expressiveness of the model because the hypothesis $h_\theta(x)$ is only able to approximate linear functions of the input.

The **Generalized Linear Models (GLMs)** represent an extension to linear models that allow non-linear transformations of the **input**:

$$h_\theta(x) = \sum_{i=1}^{m} \theta_i \cdot \Phi\left(x^{(i)}\right) = \theta^T \Phi(x)$$

so, we always consider linear combinations in the parameters, but they are made with respect to non-linear transformations of the input.

The functions **Φ** are called **basic functions**.

$$Example \quad x_1, x_2, x_3 \rightarrow x_1, x_2, x_3, x_1x_2, x_1x_3, x_1^2, x_2^2, x_3^2$$

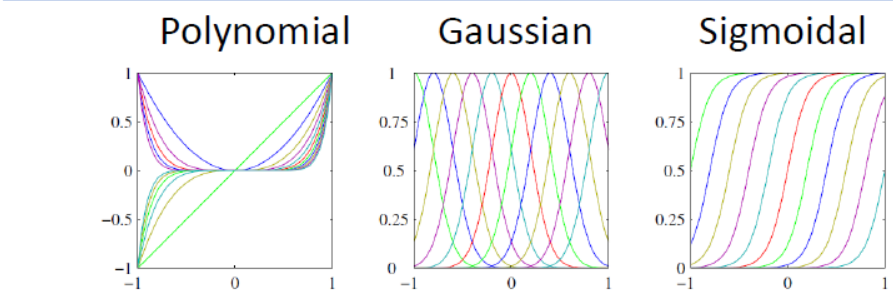
The main kinds of basic functions are:

$$Linear \qquad \phi_i(x) = x_i$$

$$Polynomial \qquad \phi_i(x) = x_i^p$$

$$Gaussian \qquad \phi_i(x) = e^{-\frac{x-\mu_i}{2\sigma^2}}$$

$$Sigmoidal \qquad \phi_i(x) = \frac{1}{1+e^{-\frac{x-\mu_i}{2\sigma^2}}}$$



9 Decision Trees

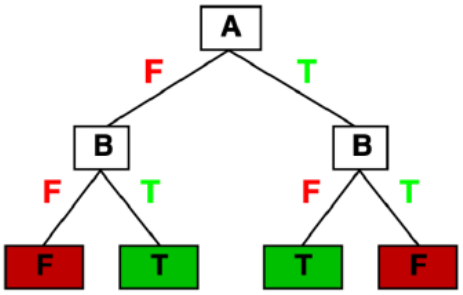
Neural networks and SVMs are highly effective at making predictions for non-linearly separable data, but they have a significant drawback: they **lack clear and intuitive interpretability**, making it difficult to understand how they arrive at their results. On the other hand, linear regression models are inherently interpretable, providing straightforward insights into how input variables influence the prediction. However, they struggle to handle non-linearly separable data effectively. As we will explore, **Decision Trees** bridge this gap by combining predictive power with **interpretability**, offering a model that is both intuitive and capable of handling complex relationships in the data.

In general, a **Decision Tree** is a tree made of condition statements (nodes) which then makes a decision based on whether or not that condition is respected.

- When a decision tree classifies things into discrete categories (e.g. True or False) it is called **Classification Tree**.
- When a decision tree predicts continuous (numeric) values is called **Regression Tree**.

A decision tree can express any function of the input attributes. For example, consider the Boolean function A XOR B, starting from the truth table we obtain a decision tree that is built on each row from the root to the leaves as shown here:

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F



As can be seen from the graphical representation (in this case a classification tree), there are two types of nodes:

- **Decision nodes**: key nodes employed to describe the condition statements (root node is one of them).
- **Leaf nodes**: nodes that represent the final results obtained.

Trivially we can observe that, unless having to describe a non-deterministic function in the attributes, there is a consistent decision tree for each training set, which in the most trivial case will have a path from the root to a leaf for each different example.

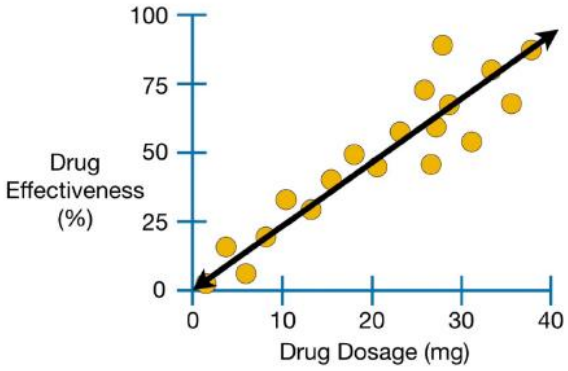
This tree, although consistent, does not have generalization qualities with respect to examples unknown to the training set, on the contrary it is certainly affected by **overfitting**, so what we are looking for is a **compact formulation of the decision tree** that is able to generalize.

Moreover, we can ask: how many distinct decision trees can be learned for a domain described by n Boolean attributes?

→ The number of possible decision trees with n attributes is equal to the number of possible Boolean functions with n attributes. So, if we have n attributes we will have a truth table with 2^n rows and so as consequence we will have 2^{2^n} possible Boolean functions → 2^{2^n} distinct decision trees.

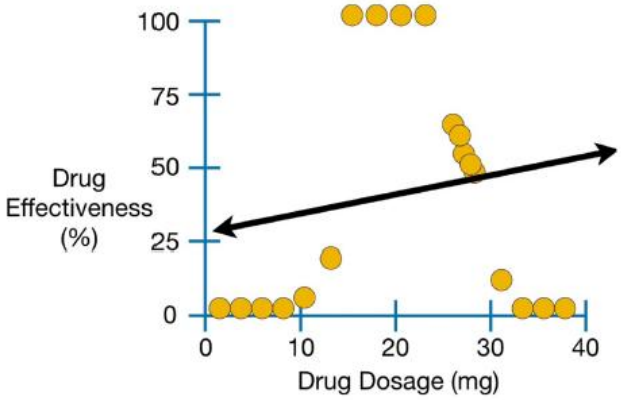
9.1 Regression Tree

Consider the following example. Suppose we developed a new drug to cure the common cold. However, we don't know the optimal dosage to give to patients, so we do a clinical trial with different dosages (drug dosages) and measure how effective each dosage is (drug effectiveness).

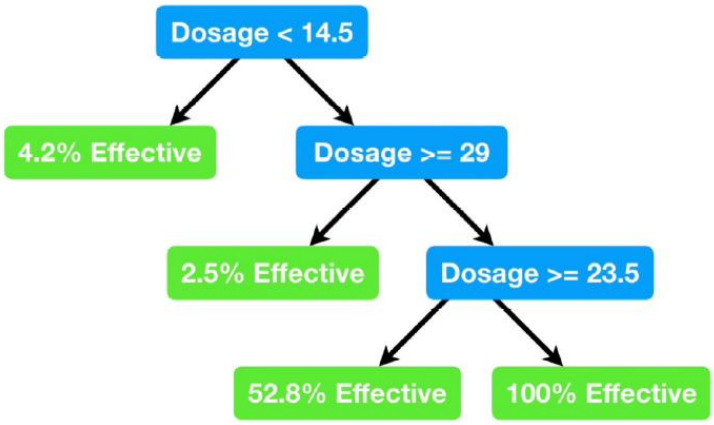


In such a situation where the data looks like the Figure on side, wanting to describe the data trend we could easily fit a straight line to the data, so it is perfectly plausible to use a linear regression model.

However, what if the data looked like the ones shown in the Figure below? Clearly a linear regression model would not be adequate to approximate the data trend in this case. The situation obviously worsens when considering multiple features.



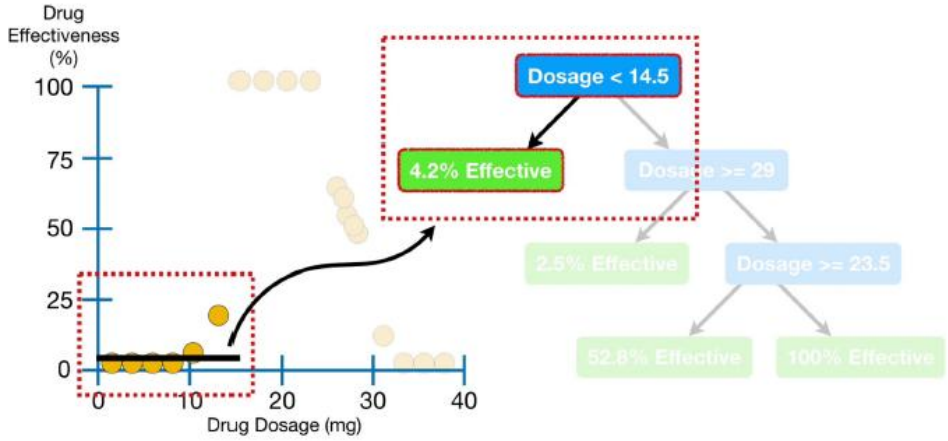
This underlines the big **limits** of using linear models such as linear regression, so we need to use something other than a straight line to make predictions. For this reason, we adopt the **Regression Tree** algorithm (which we remember to be a type of Decision Tree in which each leaf represents a **numeric value**). More specifically we aim to build a tree that looks like:



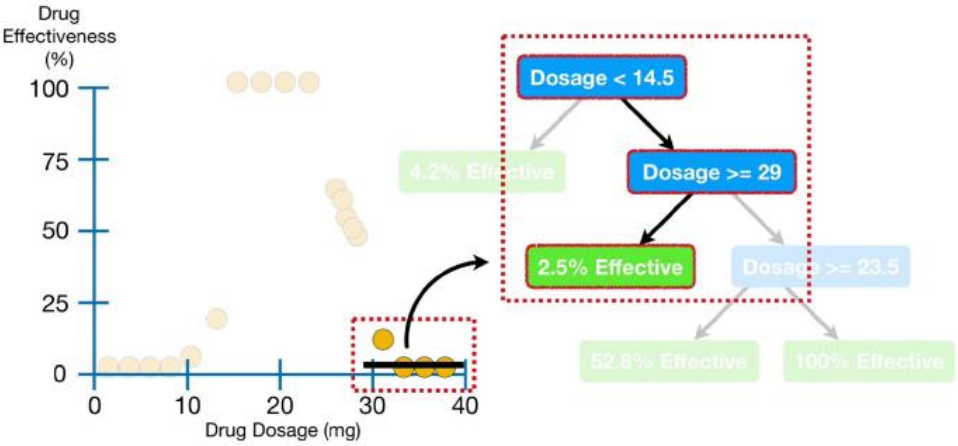
The tree is traversed starting from the **root node,** continuing along its structure according to the conditions imposed by the decision nodes, **until it reaches a leaf node**.

The process of crossing the regression tree illustrated above is as follows:

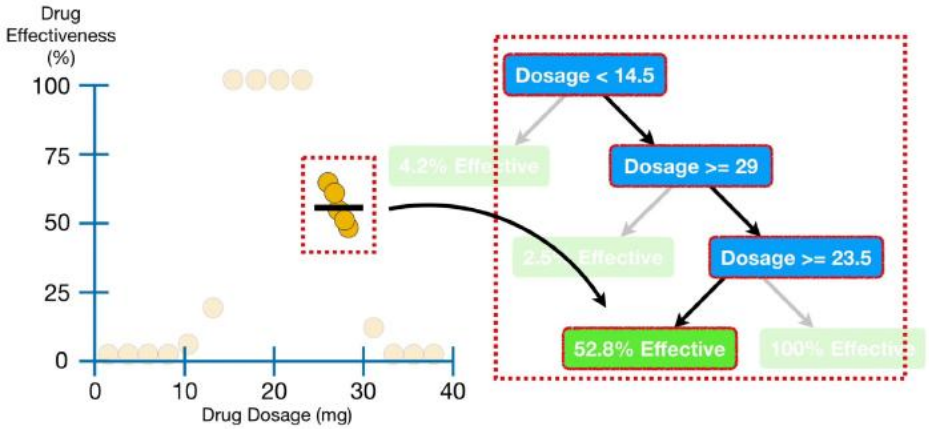
- We start by first asking if the dosage is < 14.5 (first six observations in the training data) in which case the average drug effectiveness is 4.2%. So, the tree uses the **average** drug effectiveness value 4.2% as its prediction for people with dosage < 14.5.



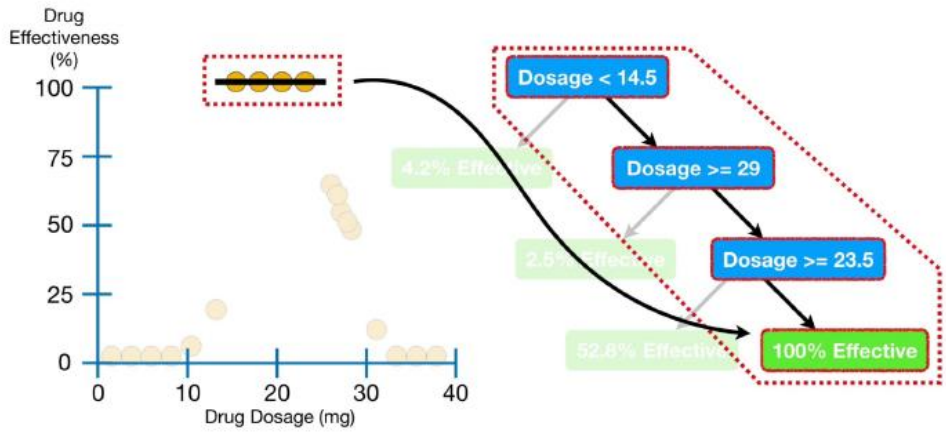
- If the dosage is >= 29 (last four observations in the training dataset) the average drug effectiveness is 2.5%. So, tree uses the average drug effectiveness value 2.5% as its prediction for people with dosage >= 29.



- If the dosage >= 23.5 and <29 the average drug effectiveness is 52.8%. So, tree uses the average drug effectiveness value 52.8% as its prediction for people with dosage >= 23.5 and < 29.



- If the dosage >= 14.5 and < 23.5 the average drug effectiveness is 100%. So, tree uses the average drug effectiveness value 100% as its prediction for people with dosage >= 14.5 and < 23.5.



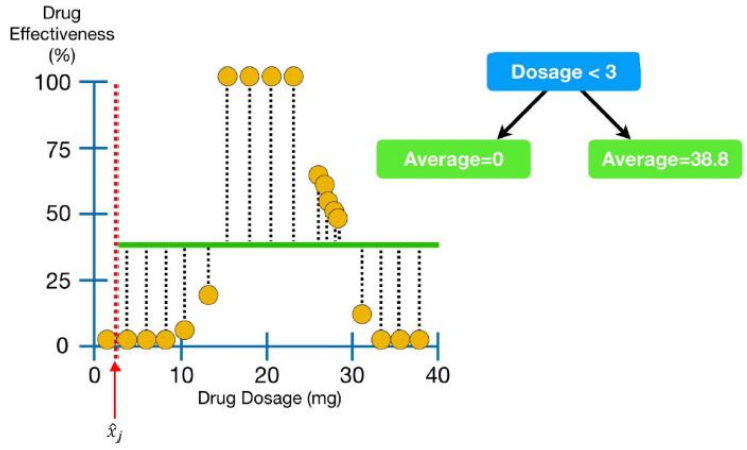
So, we can observe that a tree does a better job reflecting the data than straight line.

At this point you might be thinking, "The Regression Tree is cool, but I can also predict drug effectiveness just by looking at the graph, so why make a big deal about the Regression Tree? When the data are super simple and we are only using one predictor, drug dosage in this case, to predict drug effectiveness, making predictions by eye isn't terrible. But when we have multiple predictors, like drug dosage, age and sex, to predict drug effectiveness, drawing a graph is very difficult, if not impossible. Regression Tree can easily accommodate multiple features as we will see next.

Regression Tree with One Feature

Let's now see how to **construct** this tree. We should ask ourselves how we choose the decisions that split the tree, e.g. why do we start by asking if dosage < 14.5?

Going back to the graph of the data (Figure below) let's for example focus on the two observations with smallest dosages (the first two data). Their average dosage ($\hat{x}_j$) is 3, and that corresponds to the red dotted line. Now we can build a very simple tree that splits the observations into two groups based whether or not dosage < 3. The point on the far left is the only one with dosage < 3 and the average drug effectiveness for that point is 0%, while all the other points have dosages >= 3 and the **average** drug effectiveness for all that points is 38.8% (the green line).

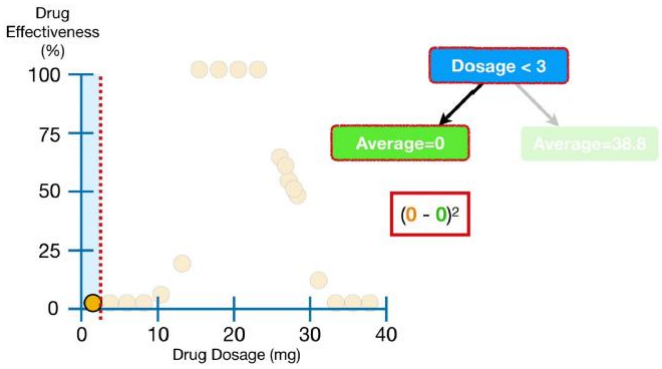


While for the point on the far left (with dosage < 3) the tree predicts a drug effectiveness value of 0%, that is pretty good since it is the same as the observed value, for a point with a dosage > 3 the tree predicts a drug effectiveness

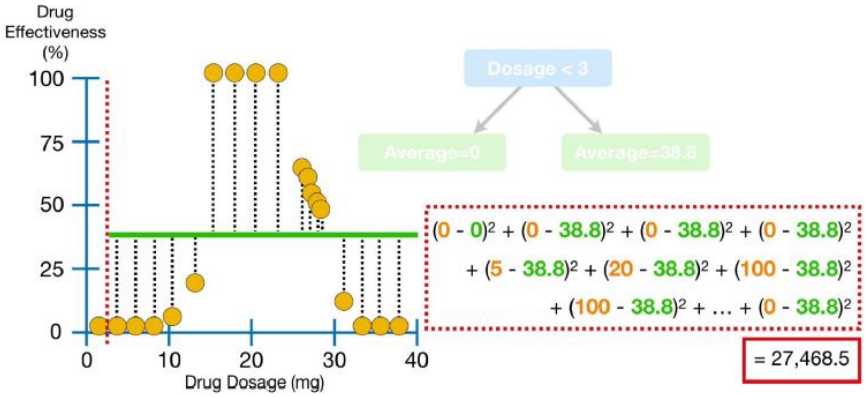
value of 38.8%, that is not very good since it is far from the observed drug effectiveness (this is especially true for instance when we consider a point with drug dosage=20 which has an observed drug effectiveness=100%).

The black dotted lines permit us to visualize how bad predictions are between the observed and predicted values. These black dotted lines are called **residuals**. So, for each point in the data, we can draw its **residual**, the **difference between the observed and predicted values** and we can use them to quantify the quality of these predictions. More specifically, what we need to do is calculate for each point the difference between its observed drug effectiveness and the predicted effectiveness (the residual) and then square this difference. So basically, what we need to do is calculate the **squared residual**.

So, let's start by calculating the squared residual for the first point.



Then we calculate and add the squared residuals for the remaining points with dosages >= 3 until we added squared residuals for every point. This sum is known as **Residual Sum of Squares** (we already talk about it in the first chapter, but we called it SSR).



Residual Sum of Squares (RSS)

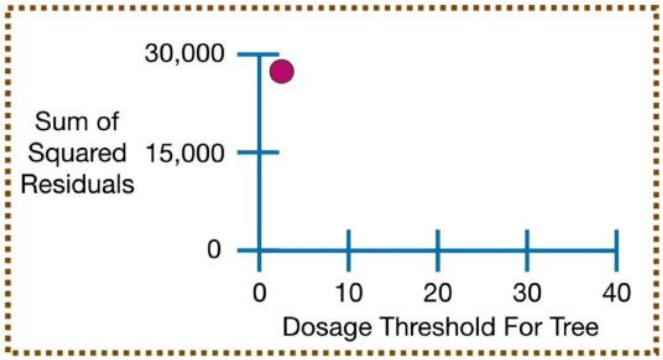
The **Residual Sum of Squares (RSS)**, also known as **Sum of Squared Residuals (SSR)**, calculates the sum of the squared residuals:

$$h(x^{(i)}) = \sum_{\hat{x}_{j-1}<x^{(i)}<\hat{x}_j} \frac{y^{(i)}}{\left|x^{(i)}: \hat{x}_{j-1} < x^{(i)} < \hat{x}_j\right|}$$

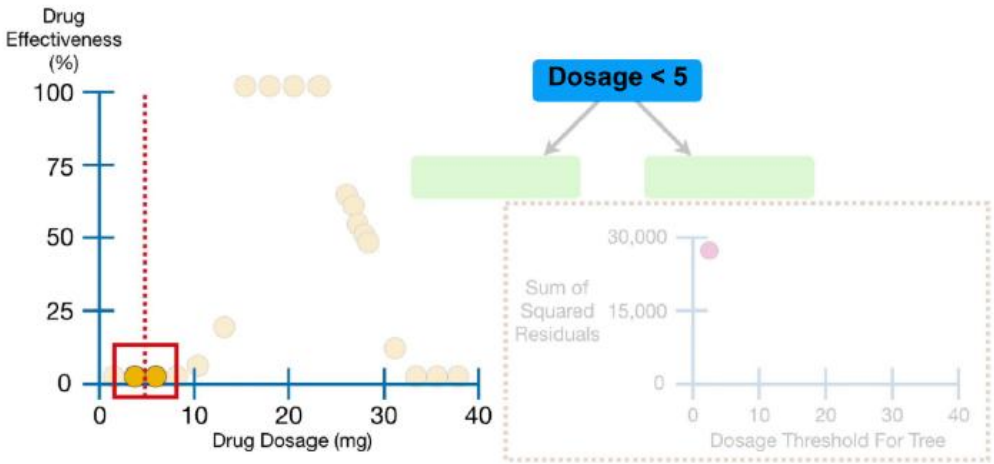
$$\boldsymbol{RSS = \sum_{i=1}^{m} \left(h(x^{(i)}) - y^{(i)}\right)^2}$$

The smaller the residual sum of squares, the better your model fits your data (of course the larger the worse).

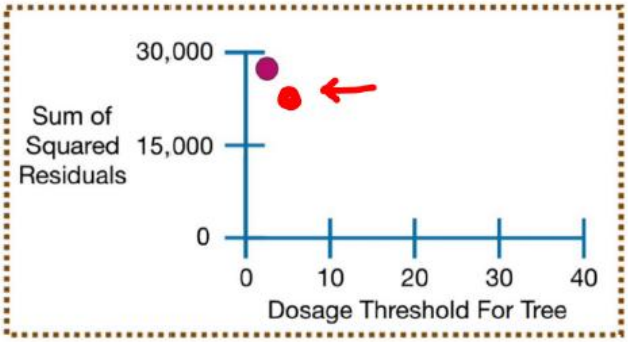
We can plot the SSR on the following graph. The y-axis corresponds to the SSR and the x-axis corresponds to dosage thresholds.



In this case we considered a dosage threshold of 3, but if we focus on the next two data points and we calculate their **average** dosage, which is 5, then we can use **dosage < 5** as **new threshold**.

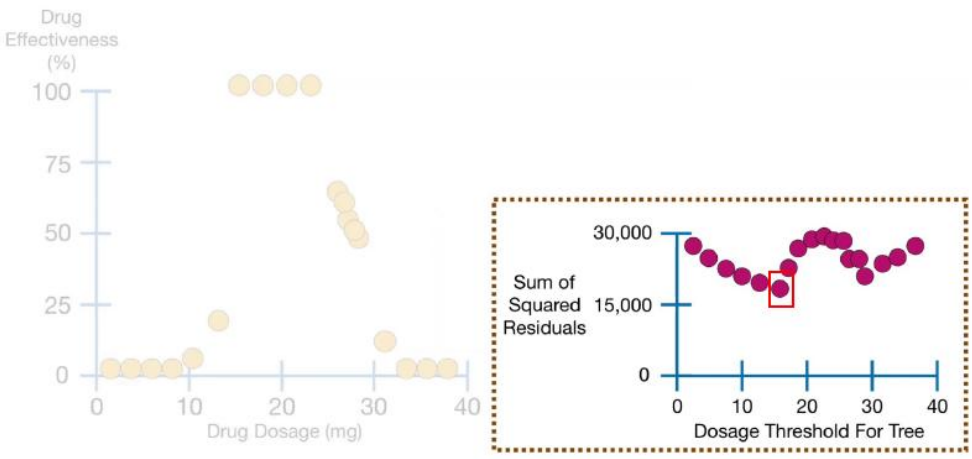


Using dosage < 5 gives us new predictions and new residuals, and that means we can add a new SSR to our graph.



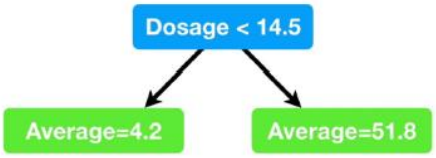
In this case, the new threshold, dosage < 5, results in a smaller SSR and that means using that threshold resulted in **better predictions** overall.

We repeat this process until we have calculated the SSR for all the remaining thresholds.



At this point we use the following criterion to choose the best value to employ as a condition (threshold): choose the value of the threshold that resituates the **smallest RSS**.

In this case the dosage < 14.5 had the smallest SSR, so it will be **root** of our tree, which not surprisingly is exactly the value chosen in the initial example illustration.



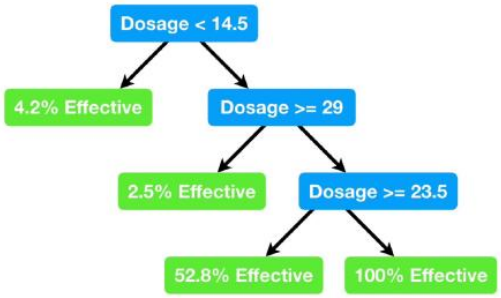
Theoretically, we could proceed in the construction of the tree (and so obtain the regression tree that we had at the beginning) by splitting the observations into smaller groups and founding the smallest SSR and the corresponding threshold for each subgroup.

When we divide the dataset into subgroups to form the branches of the tree we must be careful of the number of data we are going to enclose in each subgroup as this could lead the model to **overfit** the data (the model fits the training data perfectly) and will not perform well with new data → the model has no Bias, but potentially **large Variance** !!

To prevent our tree from overfitting the training data there are bunch of techniques, the simplest is to only split observations when there are more than some minimum number. Typically, the **minimum number of observations** to allow for a split is **20**.

- ➔ We continue with the addition of decision nodes and the formation of additional subtrees, in the presence of a number of data points per side approximately equal to **20**.

However, since this example doesn't have many observations, we set the minimum to 7. Taking these considerations into account we finally obtain the regression tree proposed at the beginning.



Regression Tree with Multiple Features

So far, we have built a tree using a single predictor (single feature), i.e. dosage, to predict drug effectiveness. However, in the case of multiple predictors (multiple features), the process of constructing the tree is practically

identical, except that, since multiple features are present, it is necessary to figure out which of those features is the best to be utilized as decision node.

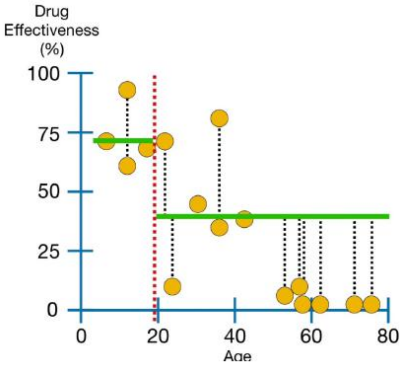
To this end, we proceed by first calculating for each feature the RSS for each threshold and choosing the one with the lowest value, as we seen above; then having the best RSS for each feature we choose the feature with the lowest RSS overall as the one to be included as the first decision node. Let's see this process in detail.

Suppose that we want to use dosage, age and sex, to predict drug effectiveness.

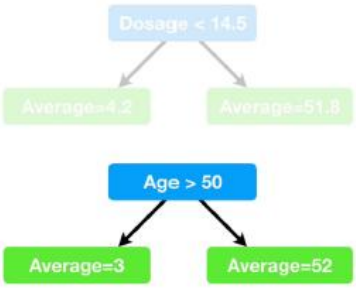
Just like before, we start by using **dosage** to predict drug effectiveness, we try it for different thresholds and calculate the SSRs at each step. Then we pick the threshold that gives us the minimum SSR, which we find to be dosage <14.5. This threshold becomes a **candidate** for the root.

Dosage	Age	Sex	Drug Effect.
10	25	Female	98
20	73	Male	0
35	54	Female	6
5	12	Male	44
etc...	etc...	etc...	etc...

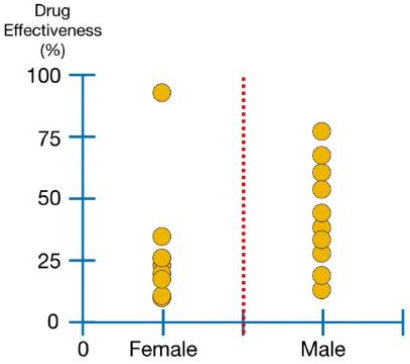
Then we focus on using **age** to predict drug effectiveness. Just like with dosage, we try different thresholds for age and calculate the SSRs at each step. We pick the one that gives us the minimum SSR, which we find to be 50.



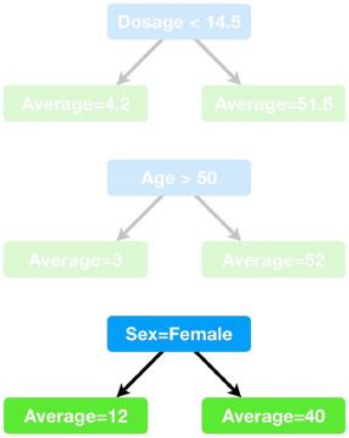
Dosage	Age	Sex	Drug Effect.
10	25	Female	98
20	73	Male	0
35	54	Female	6
5	12	Male	44
etc...	etc...	etc...	etc...



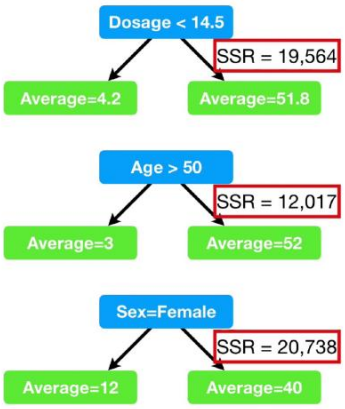
Then we focus on using **sex** to predict drug effectiveness. With sex, there is only one threshold to try, sex=female, so we use that threshold to calculate the SSR and that becomes another candidate for the root.



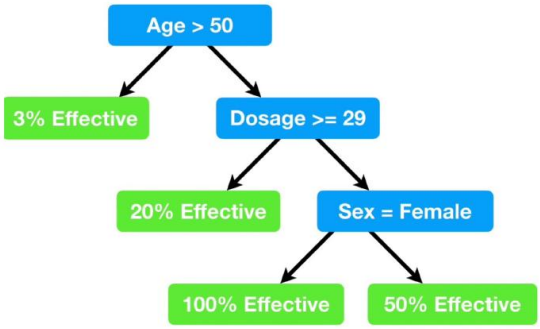
Dosage	Age	Sex	Drug Effect.
10	25	Female	98
20	73	Male	0
35	54	Female	6
5	12	Male	44
etc...	etc...	etc...	etc...



Now we compare the SSRs found for each candidate and **pick the candidate with the lowest SSR value**.



Since **Age > 50** is the threshold with the lowest SSR, it becomes the root of the tree. Then we grow the tree just like before, except now we compare the lowest SSR from each prediction keeping in mind that when a leaf has less than a minimum number of observations (which is usually 20, but here we are using 7) we stop trying to divide them. The final tree obtained proceeding like that is shown in the Figure below.



Dosage	Age	Sex	Drug Effect.
10	25	Female	98
20	73	Male	0
35	54	Female	6
5	12	Male	44
etc…	etc…	etc…	etc…

One observation to be made here concerns the fact that, within the final tree there are exactly three decision nodes, one for each feature. This is clearly just a case, as it could very well be the case that there are multiple conditions for each feature present along the tree structure. In fact, the operation to be performed whenever a new decision node needs to be inserted is always to determine the feature among those available characterized by the smallest RSS (the **optimal** one).

Summary:

- **Regression Trees** are a type of Decision Tree where each leaf represents a **numeric value**.
- We determine how to divide the observations by trying different thresholds and calculating the sum of squared residuals (RSS) at each step. The threshold with the **smallest RSS** becomes a **candidate** for the root of the tree.
- If we have more than one feature (multiple features case), we find the optimal threshold for each one and we pick the candidate with the smallest SSR to be the root.
- Then if we have fewer than some **minimum number** of observations (commonly **20**) in a node then that node becomes a leaf, otherwise we repeat the process to split the remaining observations until we can no longer split the observations into smaller groups.

9.2 Classification Tree

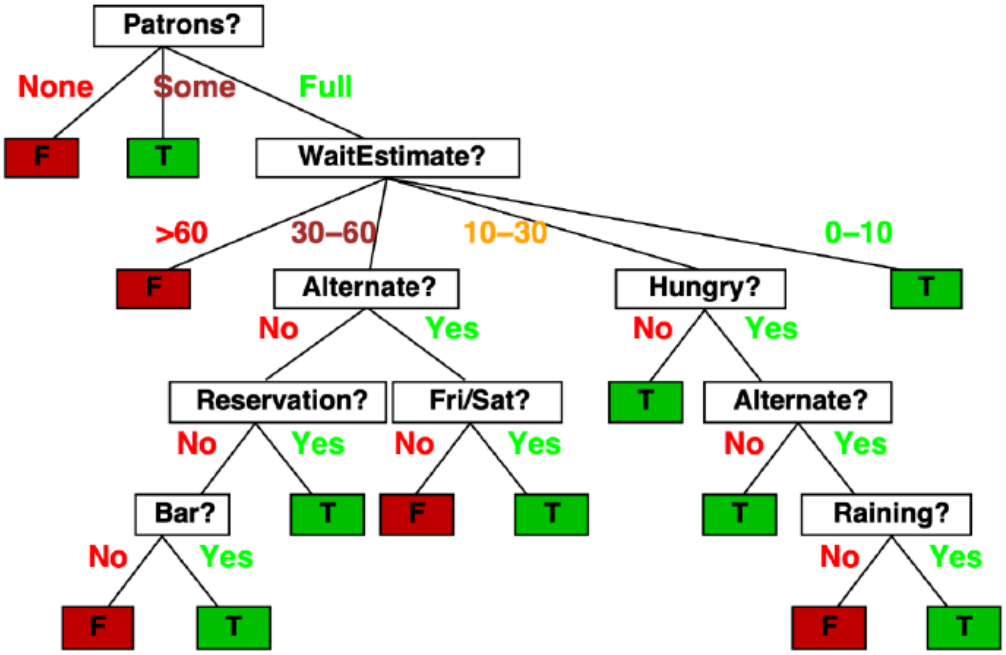
Let us now consider a classification task characterized by the following data:

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

In this case, the goal is to classify whether a user will choose to wait or not for a table at a restaurant based on several factors such as cost, waiting time, type of cuisine, etc. We aim to create a model that can both **predict the user's behavior** (wait or not wait) and **explain the reasoning behind it**.

For such a problem we can think about using a **Classification Tree**. Rember that classification tree is a decision tree used for classification problems characterized by the fact that each leaf node corresponds to one of the final decision outcomes, in this case T (True) for waiting or F (False) for not waiting. More generally, the outcomes correspond to discrete categories.

The final classification tree that we want to build (learned from all the attributes in the data table) will be as follows:

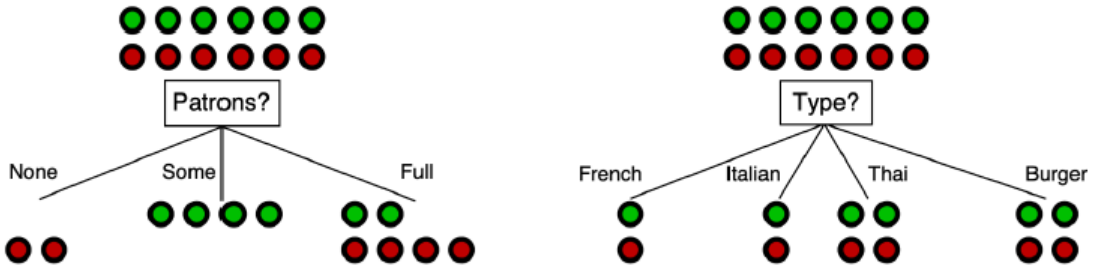


Again, here we will focus on trying to understand how to construct such a tree, beginning with the selection of the root decision node, followed by the selection of subsequent nodes.

As with regression trees, the first challenge is how to **define in which order to consider the features** in order to obtain the most compact and effective tree possible. The principle should be to select features based on how **discriminatory** they are (e.g. select a feature first, when it is most discriminatory)

Intuition → A good feature splits the dataset examples into subsets that are ideally **pure**, so subsets where all samples belong to the same class (e.g. "all positive" or "all negative").

One possible idea would be to select the feature that can return the **minimum number of misclassification errors**. As can be seen from the Figure below, selecting the "Patrons" feature is much more efficient than selecting the "Type" feature because it returns more information regarding classification, characterized by fewer misclassified examples. For this reason, Patrons would be certainly selected first among the two.



Entropy & Information Gain

The selection criterion seen above is then based on the obtainable information related to the classification, so how do we go about quantifying this type of information? We need a way to quantify the information contained in a dataset. To this end, we introduce the concept of **Entropy***: a measure of the informativeness of a set of events.

Given n events, we need $\log_2 n$ bits to represent them. Equivalently, this can be expressed as:

$$\log_2 n = -\log_2 \frac{1}{n}$$

so $-\log_2 \frac{1}{n}$ represents the information needed to represent the n events.

Each event will have a certain probability of occurring. If, for example, we consider the case of a uniform distribution then this probability will be equal to $1/n$ for each event (in a uniform distribution the events all have the same probability of occuring = $1/n$). So, if we want to measure the significance of the information related to the variable and so calculate the entropy H associated with that variable in case of uniform distribution will be given by:

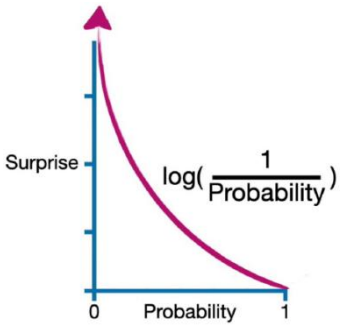
$$H(n\ events) = -\sum_{(n\ events)} \frac{1}{n} \log_2 \frac{1}{n}$$

In the more general case, with a generic probability p_i for each event i, we would have:

$$H(n\ events) = \sum_{i=1}^{n} p_i \log_2 p_i$$

This is the actual Entropy* formulation.

Analyzing the graph on the side we can realize that if the probability of an event is 1, i.e., it certainly happens, the term tends to zero; conversely, if the probability of an event is very low, the term tends to infinity.



Entropy = Expected Surprise

This behavior is explanatory of the nature of Entropy, in that, if an event with probability 1 occurs, this thing does not surprise us (low entropy), conversely, if an event with very low probability occurs, this phenomenon is indicative of something extremely unexpected and, consequently, surprises us (high entropy). For this reason, entropy can be interpreted as the **expected surprise** associated with a variable.

Entropy

For a random variable with n possible values, where each value i occurs with a probability p_i, the **entropy** $\boldsymbol{H}$ is defined as:

$$\boldsymbol{H(p_1, p_2, \dots, p_n) = \sum_{i=1}^{n} p_i \log_2 p_i}$$

It is expressed in **bits** and represents the amount of information needed to encode or describe the random variable's outcomes. Moreover, entropy can be interpreted as a measure of the level of **uncertainty** or equivalently the **expected surprise** associated with a random variable.

- **Low Entropy**: If one outcome dominates (p_i is close to 1), the entropy is low, as there is little uncertainty or low surprise about the outcome.
- **High Entropy**: If the probabilities are evenly distributed (the p_is are lower, so close to 0), the entropy is higher, reflecting greater uncertainty or high surprise about the outcome.

Entropy provides a theoretical foundation for selecting features in classification tasks by quantifying their informativeness. **Features that divide the dataset into subsets with low entropy are preferred, as they reduce uncertainty and provide clearer distinctions between classes**.

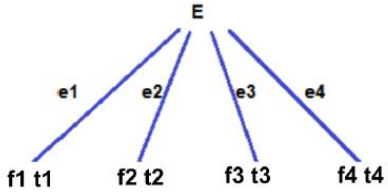
Using Entropy for Feature Evaluation

To understand how entropy is applied in classification tasks, consider a target attribute X with a number t of true elements (class 1) and a number f of false elements (class 0). If we ask what is the probability that an element of that classification attribute is true it is $\frac{t}{t+f}$, while the probability that an element is false is $\frac{f}{t+f}$, so the information-related entropy (calculated on the target attribute X) is:

$$H\left(\frac{t}{t+f}, \frac{f}{t+f}\right) = -\frac{t}{t+f}\log_2\frac{t}{t+f} - \frac{f}{t+f}\log_2\frac{f}{t+f}$$

This represents the **initial entropy** $\boldsymbol{H_0}$ of the dataset before considering any split.

Suppose we introduce a feature E and wish to evaluate its discriminatory power against X. Assume E can take four possible values e_1, e_2, e_3, e_4, splitting the dataset into four subsets E_1, E_2, E_3, E_4. For each subset E_i, let:



- t_i: the number of positive samples (class 1) for the subset E_i
- f_i: the number of negative samples (class 0) for the subset E_i

The entropy of subset E_i, denoted as $H(E_i)$, is calculated as:

$$\boldsymbol{H(E_i) = H\left(\frac{t_i}{t_i + f_i}, \frac{f_i}{t_i + f_i}\right) = -\frac{t_i}{t_i + f_i} \log_2 \frac{t_i}{t_i + f_i} - \frac{f_i}{t_i + f_i} \log_2 \frac{f_i}{t_i + f_i}}$$

Now what we want to do is to calculate the entropy $H(E)$. To do it we start by calculating the entropy related to each of all the possible four feature values e_1, e_2, e_3 and e_4 one by one.

For example, if we choose e_1 we will have for it a number t_1 of true elements and a number f_1 of false elements and its entropy will be:

$$H(E_1) = H\left(\frac{t_1}{t_1 + f_1}, \frac{f_1}{t_1 + f_1}\right) = -\frac{t_1}{t_1 + f_1} \log_2 \frac{t_1}{t_1 + f_1} - \frac{f_1}{t_1 + f_1} \log_2 \frac{f_1}{t_1 + f_1}$$

than we do the same for e_2, e_3, e_4.

To calculate the total entropy $H(E)$ for feature E, we have to add the entropies calculated with respect to the single attribute values ($H(E_1)$, $H(E_2)$, $H(E_3)$and $H(E_4)$), each of them **weighted** according to the probability of taking that feature value, so:

$$H(E) = H(E_1, E_2, E_3, E_4) = \sum_{i=1}^{4} \frac{t_i + f_i}{t + f} H(E_i)$$

This represents the **final entropy $\boldsymbol{H(E)}$** after splitting the dataset using feature E.

More in general, the **final entropy after splitting based on a feature $\boldsymbol{E}$**, i.e. $\boldsymbol{H(E)}$, is the weighted sum of the entropies of its k subsets:

$$\boldsymbol{H(E) = \sum_{i=1}^{k} \frac{t_i + f_i}{t + f} H(E_i)}$$

So, starting from the initial entropy H_0 and choosing E as the discriminating feature, we obtain a set of subsets with a new final entropy $H(E)$.

At this point it is possible to calculate the so-called **Entropy Gain**, also known as **Information Gain**:

$$\boldsymbol{IG(E) = H_0 - H(E)}$$

which measures the reduction in uncertainty (or entropy) of a dataset after splitting it based on a specific feature (in this case E).

In the context of decision trees, we will use it to identify which feature provides the most valuable information for classifying the data, helping to determine the splits at each node.

E.g. **when multiple features are available, we choose the feature with the highest IG as it provides the greatest reduction in uncertainty**.

Let's now see how to apply what we have seen so far to our starting example.

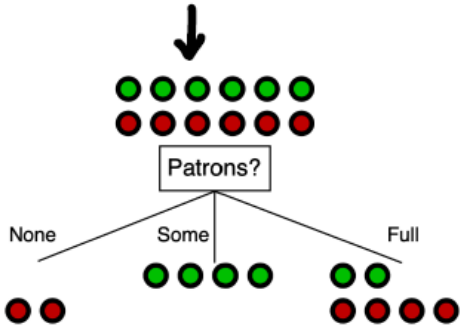
Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
X_5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*F*	*T*	*French*	*>60*	*F*
X_6	*F*	*T*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Italian*	*0–10*	*T*
X_7	*F*	*T*	*F*	*F*	*None*	*$*	*T*	*F*	*Burger*	*0–10*	*F*
X_8	*F*	*F*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Thai*	*0–10*	*T*
X_9	*F*	*T*	*T*	*F*	*Full*	*$*	*T*	*F*	*Burger*	*>60*	*F*
X_{10}	*T*	*T*	*T*	*T*	*Full*	*$$$*	*F*	*T*	*Italian*	*10–30*	*F*
X_{11}	*F*	*F*	*F*	*F*	*None*	*$*	*F*	*F*	*Thai*	*0–10*	*F*
X_{12}	*T*	*T*	*T*	*T*	*Full*	*$*	*F*	*F*	*Burger*	*30–60*	*T*

We can start by calculating the **initial entropy H_0** of the dataset. In this case considering that our **target** variable ("*WillWait*") has $t = f = 6$, we have:

$$H_0 = H\left(\frac{t}{t+f}, \frac{f}{t+f}\right) = -\frac{6}{12}\log_2\frac{6}{12} - \frac{6}{12}\log_2\frac{6}{12} = 1$$

Then let's calculate the information-related entropy of "Patrons" feature. It has three possible values (None, Some, Full) and therefore leads to the splitting of the samples into three classes.

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
X_5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*F*	*T*	*French*	*>60*	*F*
X_6	*F*	*T*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Italian*	*0–10*	*T*
X_7	*F*	*T*	*F*	*F*	*None*	*$*	*T*	*F*	*Burger*	*0–10*	*F*
X_8	*F*	*F*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Thai*	*0–10*	*T*
X_9	*F*	*T*	*T*	*F*	*Full*	*$*	*T*	*F*	*Burger*	*>60*	*F*
X_{10}	*T*	*T*	*T*	*T*	*Full*	*$$$*	*F*	*T*	*Italian*	*10–30*	*F*
X_{11}	*F*	*F*	*F*	*F*	*None*	*$*	*F*	*F*	*Thai*	*0–10*	*F*
X_{12}	*T*	*T*	*T*	*T*	*Full*	*$*	*F*	*F*	*Burger*	*30–60*	*T*



We calculate the entropy associated with each of them so to be able to compute $H(Patrons)$:

$$H(None) = H\left(\frac{t_{None}}{t_{None}+f_{None}}, \frac{f_{None}}{t_{None}+f_{None}}\right) = H(0,1) = -\frac{0}{2}\log_2\frac{0}{2} - \frac{2}{2}\log_2\frac{2}{2} = 0$$

$$H(Some) = H\left(\frac{t_{Some}}{t_{Some}+f_{Some}}, \frac{f_{Some}}{t_{Some}+f_{Some}}\right) = H(1,0) = -\frac{4}{4}\log_2\frac{4}{4} - \frac{0}{4}\log_2\frac{0}{4} = 0$$

$$H(Full) = H\left(\frac{t_{Full}}{t_{Full}+f_{Full}}, \frac{f_{Full}}{t_{Full}+f_{Full}}\right) = H\left(\frac{2}{6}, \frac{4}{6}\right) = -\frac{2}{6}\log_2\frac{2}{6} - \frac{4}{6}\log_2\frac{4}{6} = 0.9183$$

$$\Rightarrow H(Patrons) = \frac{t_{None}+f_{None}}{t+f}H(None) + \frac{t_{Some}+f_{Some}}{t+f}H(Some) + \frac{t_{Full}+f_{Full}}{t+f}H(Full) =$$
$$= \frac{2}{12}0 + \frac{4}{12}0 + \frac{6}{12}0.9183 = 0.4591$$

and then we calculate the IG:

$$IG(Patrons) = H_0 - H(Patrons) = 1 - 0.4591 = 0.5409 \approx \mathbf{0.541}$$

Next, we can calculate the information-related entropy to the "Type" feature and its IG.

"Type" has four possible values (French, Thai, Burger, Italian) and therefore leads to the splitting of the samples into four classes. Following the same procedure seen for "Patrons" we obtain:

$$IG(Type) = H_0 - H(Type) = 1 - \left[\frac{2}{12}H\left(\frac{1}{2},\frac{1}{2}\right) + \frac{2}{12}H\left(\frac{1}{2},\frac{1}{2}\right) + \frac{4}{12}H\left(\frac{2}{4},\frac{2}{4}\right) + \frac{4}{12}H\left(\frac{2}{4},\frac{2}{4}\right)\right] = \mathbf{0}$$

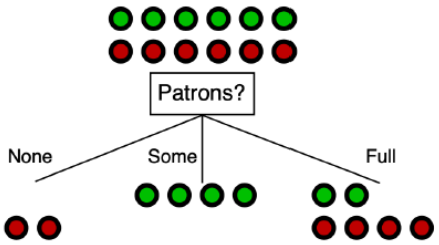
Since "Patrons" has a higher IG than Type than it's "Patrons" the best first choice as discriminating feature between the two.

Note: You should notice that by applying the classification through "Patrons" the **entropy is decreased** (=0.4591) with respect to the initial total entropy (=1), and this corresponds to a smaller number of bits necessary for the classification.

Now repeating this procedure for the other features, we will find that the one with highest IG is "Patrons" and so it will be selected to be the root of the tree.

➔ **Feature selection criterion**: **Select the feature that returns the highest Information Gain (IG)**.

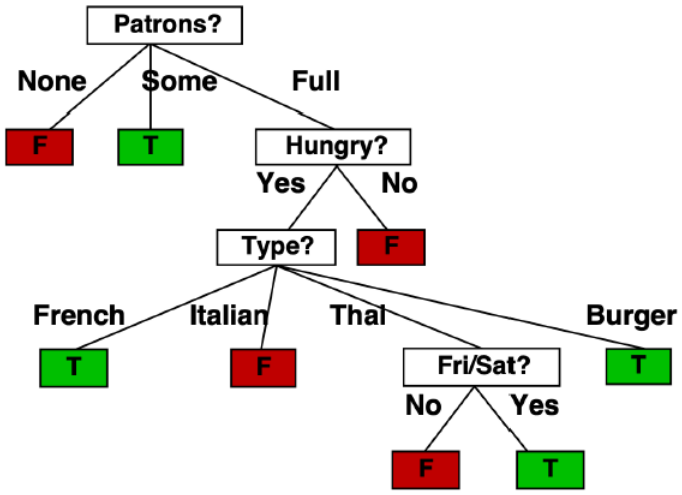
At this point, after selecting the "Patrons" feature, we are in the situation just reported. So, how do you choose the next feature? What we do is **remove the correctly classified examples from the dataset** (marked in red)**, to focus only on the examples characterized by misclassification**. So just considering the ones that end up in **impure leaves**, e.g. in this case Full class.



Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	~~T~~	~~F~~	~~F~~	~~T~~	~~Some~~	~~$$$~~	~~F~~	~~T~~	~~French~~	~~0–10~~	~~T~~
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	~~F~~	~~T~~	~~F~~	~~F~~	~~Some~~	~~$~~	~~F~~	~~F~~	~~Burger~~	~~0–10~~	~~T~~
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	~~F~~	~~T~~	~~F~~	~~T~~	~~Some~~	~~$$~~	~~T~~	~~T~~	~~Italian~~	~~0–10~~	~~T~~
X_7	~~F~~	~~T~~	~~F~~	~~F~~	~~None~~	~~$~~	~~T~~	~~F~~	~~Burger~~	~~0–10~~	~~F~~
X_8	~~F~~	~~F~~	~~F~~	~~T~~	~~Some~~	~~$$~~	~~T~~	~~T~~	~~Thai~~	~~0–10~~	~~T~~
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	~~F~~	~~F~~	~~F~~	~~F~~	~~None~~	~~$~~	~~F~~	~~F~~	~~Thai~~	~~0–10~~	~~F~~
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

At this point, the entropy associated with the subset consisting only of the remaining samples is calculated again and then the IG of the remaining features is calculated in a similar way as described above, until the overall classification of all data points is obtained (i.e. there are no more impure leaves).

At the end of this process, you should end with a final classification tree as the one given in the Figure below.



9.3 Numerical Values in Classification Trees

Above we saw how to calculate the Information Gain (IG) in order to choose which feature will be used as root of a classification tree (feature with the highest IG). More specifically, we saw how to do it when we had to compare features having categorical values, but how do we proceed if in the dataset we find a **feature with numerical values**? Here we will go through it and see how to handle this case.

Suppose we have a dataset, as the one shown in the Figure below, where there is a feature with numerical values, i.e. the "Age" feature. To handle it we proceed by taking these steps:

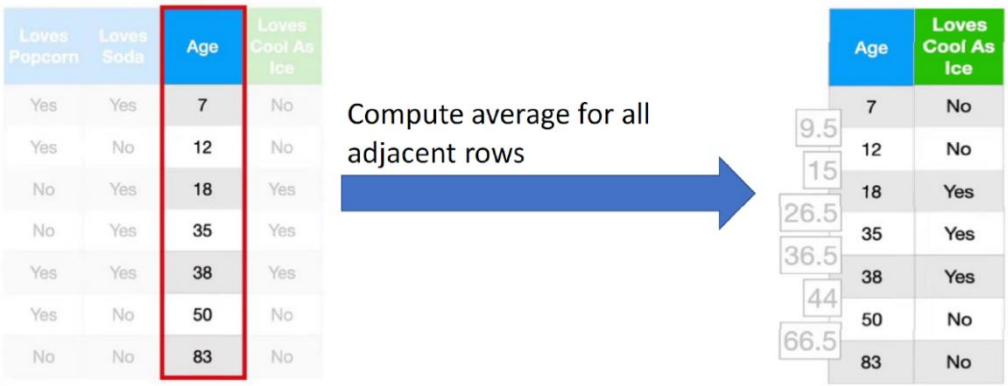
- **Step 1**: We sort the rows of the Age feature from the lowest value to the highest.

Loves Popcorn	Loves Soda	Age	Loves Cool As Ice
Yes	Yes	7	No
Yes	No	12	No
No	Yes	18	Yes
No	Yes	35	Yes
Yes	Yes	38	Yes
Yes	No	50	No
No	No	83	No

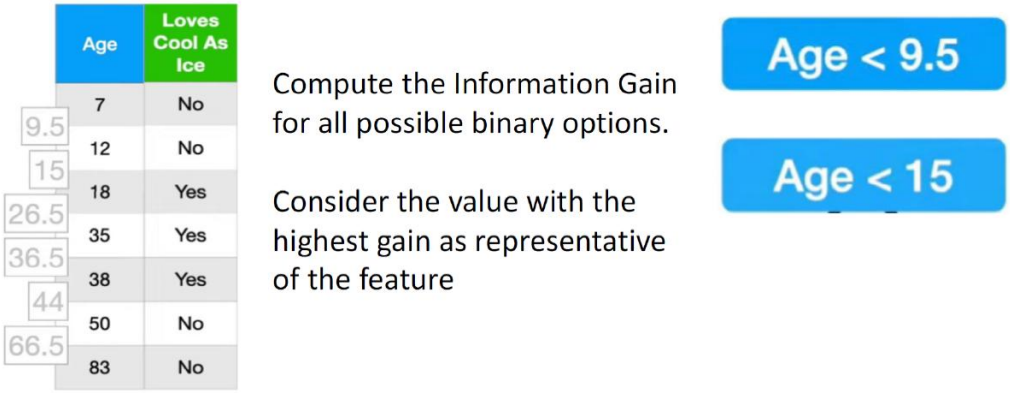
Sort rows

Loves Popcorn	Loves Soda	Age	Loves Cool As Ice
Yes	Yes	7	No
Yes	No	12	No
No	Yes	18	Yes
No	Yes	35	Yes
Yes	Yes	38	Yes
Yes	No	50	No
No	No	83	No

- **Step 2**: We compute the average for each couple of adjacent rows.

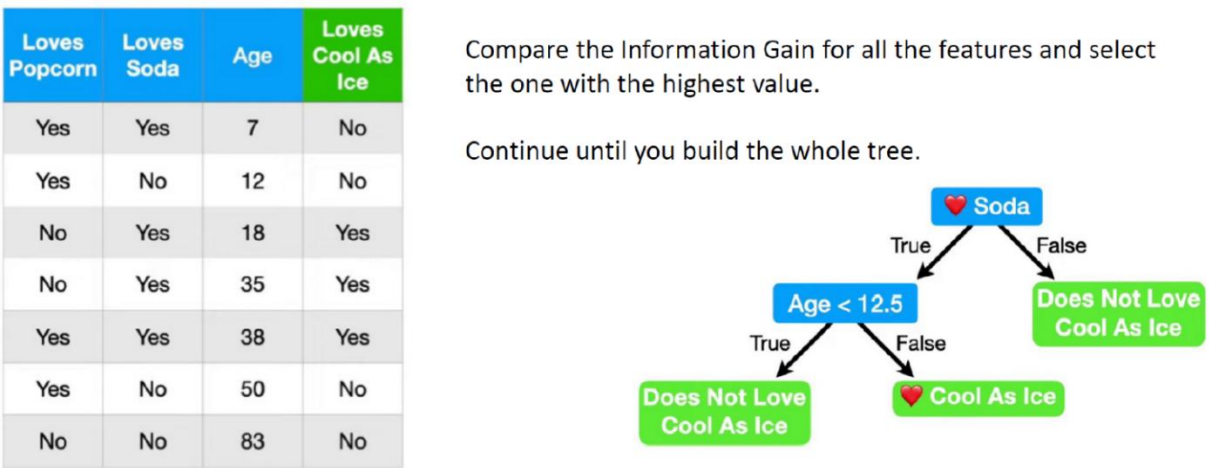


- **Step 3**: We compute the Information Gain for all possible binary options (Age < 9.5, Age < 15 and so on). After computing all of them we consider the value with the highest IG as representative of the feature.



- **Step 4**: We compare the IG for all the other features (Loves Popcorn, Loves Soda, etc.) and then we select the one with highest IG value.

We continue like this until we build the whole three.



End.

9.4 Missing Values in Decision Trees

If we are missing some feature values, there are a lot of ways to guess what it might be.

Suppose we want to predict when a person might have or not a *Heart Disease* and we have the dataset shown in the Figure below. As we can see we have a **missing value** for the *Blocked Arteries* feature.

Chest Pain	Good Blood Circulation	Blocked Arteries	Heart Disease
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	No	No
Yes	No	???	Yes
etc...	etc...	etc...	etc...

The most common option is to fill that value with the **most frequent value** in the column.

For example, in this case "Yes" occurs more times than "No", so we could put "Yes".

Chest Pain	Good Blood Circulation	Blocked Arteries	Heart Disease
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	No	No
Yes	No	**YES**	Yes
etc...	etc...	etc...	etc...

Alternatively, we could find a **correlated feature and use it as guideline**.

For example, in this case Chest Pain and Blocked Arteries are often very similar. Looking at the first row both are "No", in the second row both are "Yes" and so on. So, for the fourth row since Chest Pain is "Yes" we'll make Blocked Arteries "Yes" as well.

Chest Pain	Good Blood Circulation	Blocked Arteries	Heart Disease
No	No	No	No
Yes	Yes	Yes	Yes
No	Yes	No	No
Yes	No	**???**	Yes
etc...	etc...	etc...	etc...

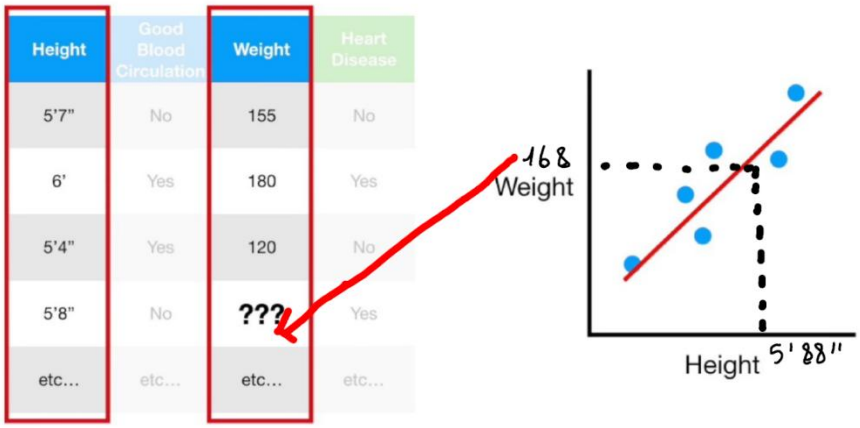
And what to do if we are missing a continuous value (e.g. a numeric value)? Also here we have few ways to guess what might be the missing value.

Imagine we had Weight feature instead of Blocked Arteries feature and we are missing a weight value (continuous value).

Height	Good Blood Circulation	Weight	Heart Disease
5'7"	No	155	No
6'	Yes	180	Yes
5'4"	Yes	120	No
5'8"	No	???	Yes
etc...	etc...	etc...	etc...

We could replace this missing value with the **mean** or the **median**.

Alternatively, we could find a feature that has the highest correlation with it, **do a linear regression** on the two features and use the least square line to predict the missing value.



For example, we found that Height is high correlated with Weight, we did a linear regression on Weight and Height and we used the least square line to predict the value that for 5' 88'' is 168. So, we replace the missing value with 168.

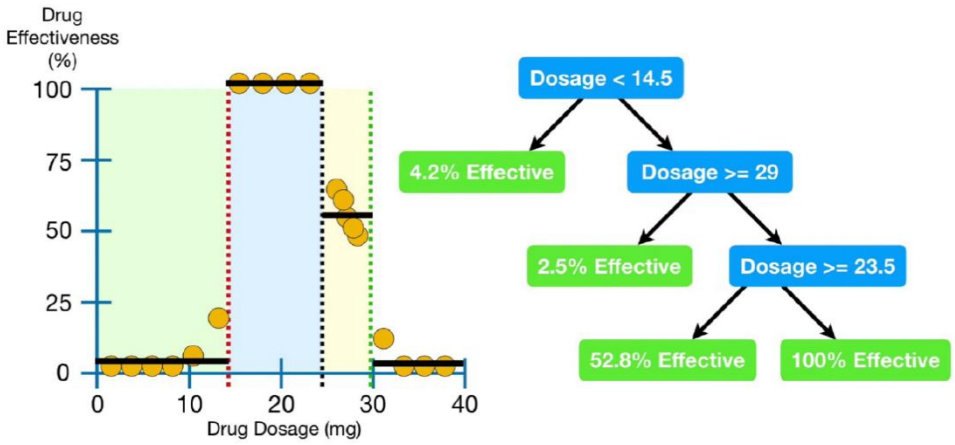
9.5 Pruning Regression Trees

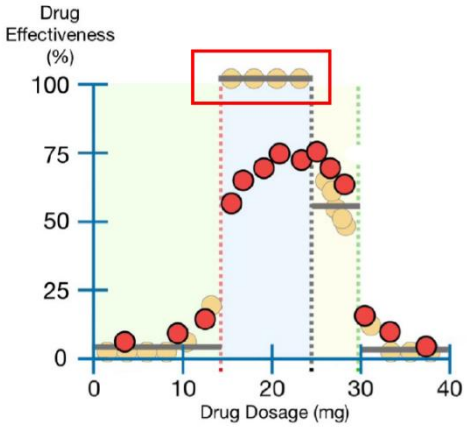
There are several methods for pruning Regression Trees, the one that we will see here is called ***Weakest Link Pruning***, also known as ***Cost Complexity Pruning***.

We'll start by giving a general overview of how *Weakest Link Pruning* works and then we'll describe how to use it to build Regression Trees.

In the Regression Tree section, we built a Regression Tree that, given different drug dosages, was able to predict drug effectiveness.

The tree did a pretty good job in reflecting the training data because each leaf represented a value that was close to the data.





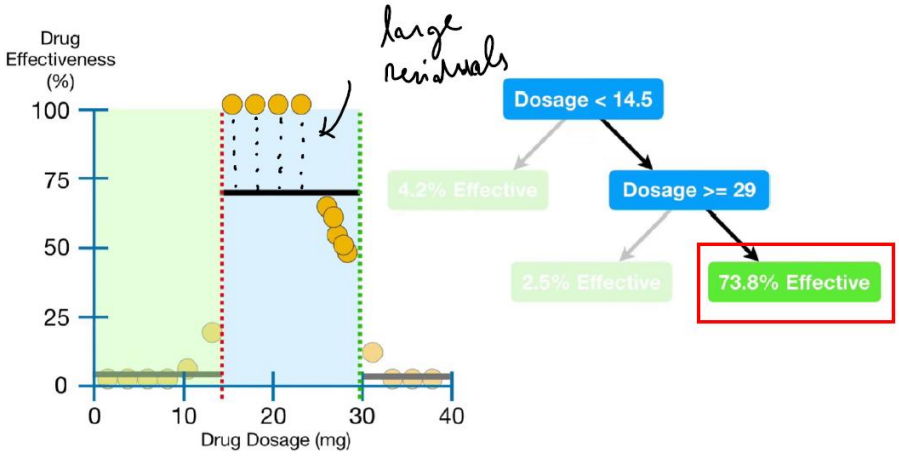
However, what if these new data represented in the Figure on side as red circles were in the Test set?

Some of them are pretty close to the predicted values so their residuals (difference between the observed and predicted values) are not very large.

However, residuals for some other of them are larger than before. In fact, it seems like the four observations from the training data with 100% drug effectiveness now look a little bit like outliers and that means we **overfit** the Regression Tree to the **training data**.

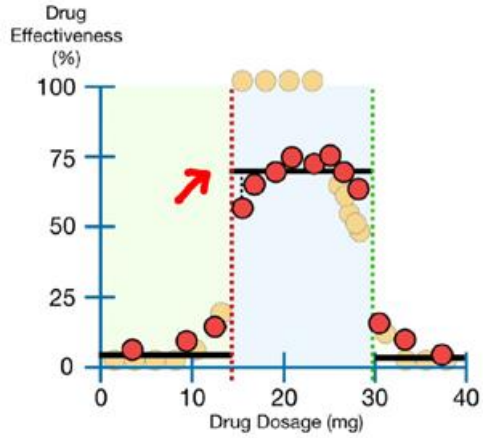
One way to prevent overfitting a regression tree to the training data is to **remove some of the leaves** and replace the split with a leaf that is the average of a larger number of observations.

For example, we can remove the last two leaves (52.8% and 100%) and replace them with a single one that is the average of the two (73.8). Doing that we end up with:



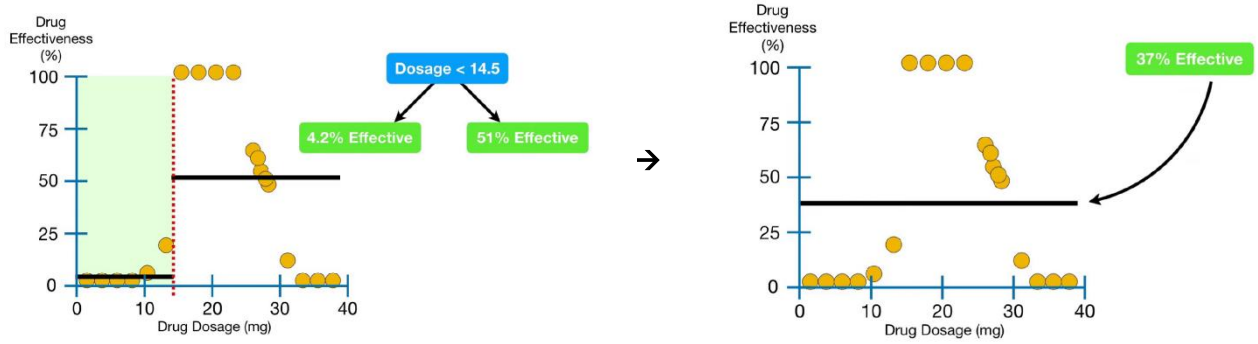
Now all of the observations between 14.5 and 29 go to the leaf on the far right.

The large residuals shown for the four observations with 100% drug effectiveness in this new configuration tell us that the new tree doesn't fit the training data as well as before, BUT the new sub-tree does a much better job with the **test data**.

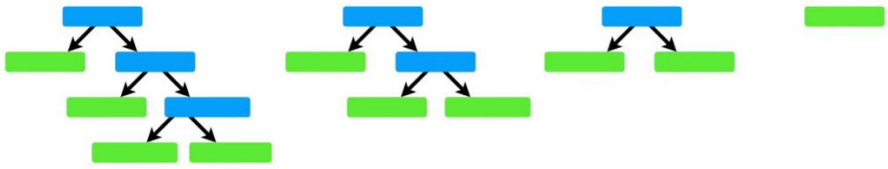


→ Thus, the main idea behind **pruning a Regression tree** is to prevent overfitting the training data so that the tree will do a better job with test data !

If we want, we can perform **further pruning** (prune the three more) by removing other two leaves and replacing the split with a leaf of a larger number of observations multiple times.

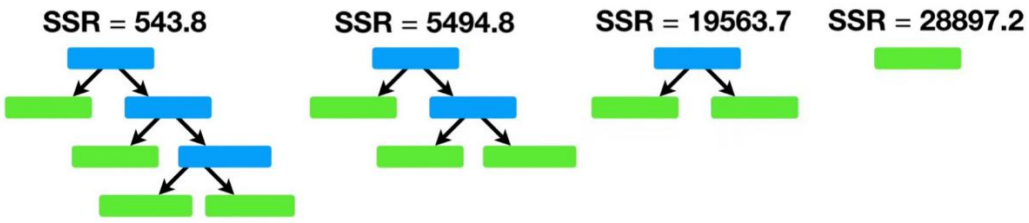


So, now the question is: **"How to decide which tree to use? Which tree is the best?"**



To answer this question, we'll have to apply ***Weakest Link Pruning***.

The first step in *Weakest Link Pruning* is to calculate the total **Sum of Squared Residuals** (**SSR**) for each tree. After computing them we will have:



Note that the SSR is relatively small for the original full-sized tree, but each time we remove a leaf, the SSR gets larger and larger. However, we knew that was going to happen because the whole idea was for the pruned trees to **NOT** fit the **training data** as well as the full-sized tree.

So how do we compare these trees?

Weakest Link Pruning works by calculating a **Tree Score** that is based on the **SSR** for the tree or sub-tree and a **Tree Complexity Penalty** ($\boldsymbol{\alpha T}$) that is a function of the number of leaves, or **T**erminal nodes, in the tree or sub-Tree.

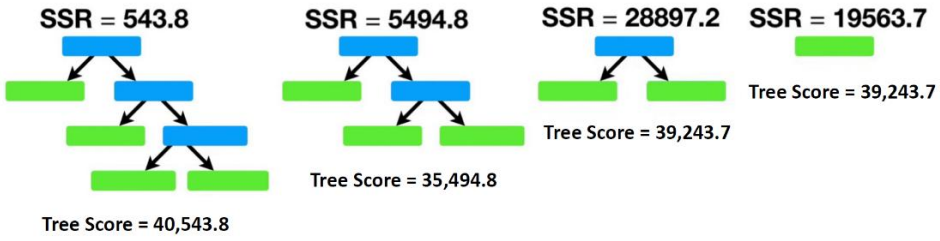
$$\boldsymbol{Tree\ Score = SSR + Tree\ Complexity\ Penality =}$$

$$\boldsymbol{= SSR + \alpha \cdot T}$$

where $\boldsymbol{\alpha}$ is a tuning parameter used to regularize this penalty, and it's found using **Cross Validation** as we'll see in a bit. → The Tree Complexity Penalty (αT) compensates for the difference in the number of leaves.

For now, let's let $\alpha = 10000$ and let's calculate the Tree Score for each tree. After calculating all of them, using the formula that we saw above, we obtain:

Tree Score = SSR + $\alpha \cdot T$
Example with $\alpha = 10.000$

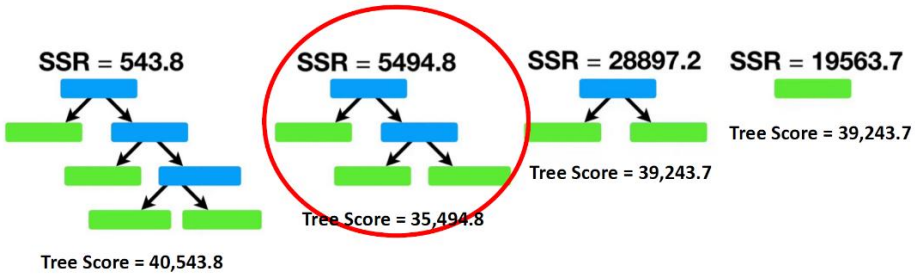


Remember that T is the total number of leaves, so for example for the first tree T=4.

Note: More leaves a tree has and larger will be the Complexity Penalty.

So, we pick the first sub tree because it has the lowest Tree Score.

Tree Score = SSR + $\alpha \cdot T$
Example with $\alpha = 10.000$



➔ In general, to choose the best tree we apply the Weakest Link Pruning, computing the Tree Score for the tree and each sub-tree and picking the one that has the **lowest Tree Score**.

How to evaluate the best α?

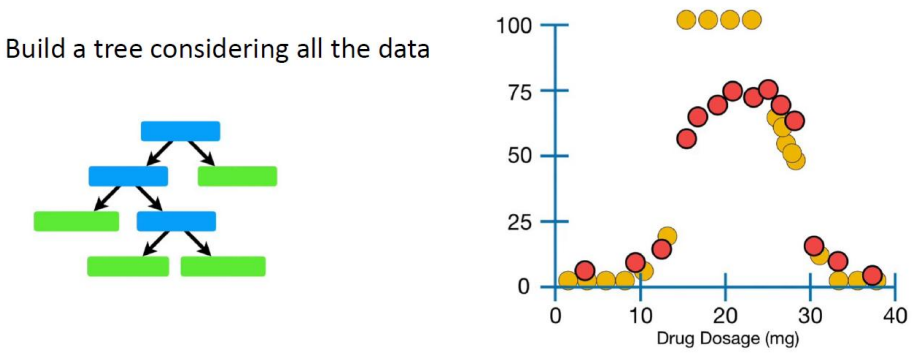
If we **change** α (for example $\alpha = 20000$) and calculate again the tree scores then we could notice that another tree has the lowest tree score thus, the value for α makes an **important difference** in our choice of the best sub-tree.

So now let's see how to build a pruned regression tree and how to find the best value for α.

1. Finding the Best values for α

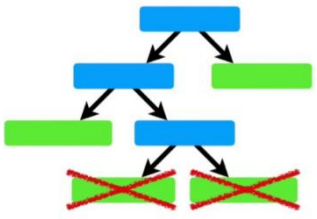
First, we build a full-sized regression tree using **all** the data (both training data and test data).

Note: This full-size tree is different than the one we have seen before because it has to fit all data, not just the training data !!!



This full-sized tree has the lowest Tree Score when $\alpha = 0$. This because when $\alpha = 0$, the tree complexity penalty becomes 0 and the Tree Score is just the SSR ($Tree\ Score = SSR$).

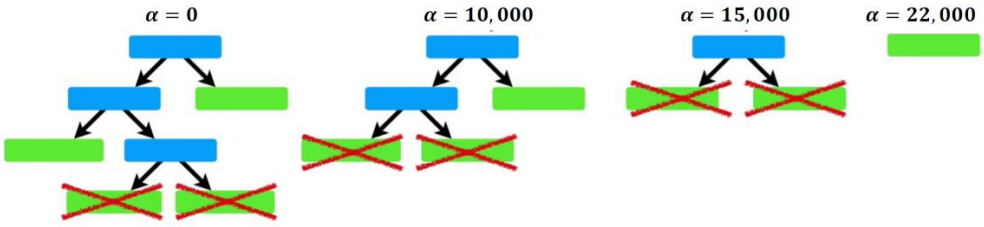
Now we increase α until pruning leaves will give us a lower Tree Score.



So, we start by removing the last two leaves and increasing α.

In this case, when $\alpha = 10000$ we'll get a lower Tree Score and so we use this new subtree.

We increase α again until pruning leaves will give us a lower Tree Score obtaining a new subtree that has the best Tree Score for that specific α value. We repeat this process until we removed all leaves. In the end, different values for α give us a sequence of trees that minimize the Tree Score.



➔ We get a sequence of sub-trees, each optimal for a specific α.

2. Evaluating α values: Cross-Validation to find the optimal α

Once we identify potential values of α from the pruning process, we evaluate them using **10-Fold Cross-Validation** to determine the optimal overall value. The data (both training and test) is initially divided into **10 folds**. Each fold is used as a test (validation) set once, while the remaining 9 folds are used as the training set for that iteration.

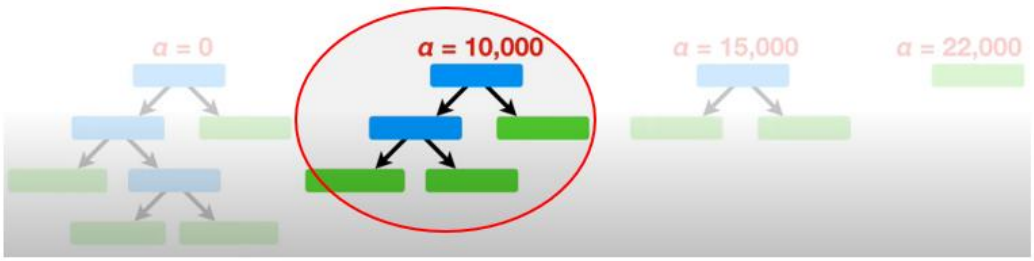
During the procedure all values of α found are kept **fixed**.

1. We take **only the training portion** of the given fold configuration, and we use it to **train** the full-size tree and the sequence of sub-trees.
2. Next, using **only the test portion** of the current fold configuration, we compute for each new tree the SSR and the Tree Score (using the fixed α values). We vote for the tree with the **lowest Tree Score** for this fold and we take note of the α for that tree.

We repeat this process for each of the 10 folds.

At the end, the value for α that, on **average**, gave us the **lowest SSR** across the 10 folds is selected as the **final α**. In this case, the optimal trees built with $\alpha = 10000$ had on average the lowest SSR, so $\boldsymbol{\alpha = 10000}$ is our final value.

Lastly, once we determined the optimal α, we can return to the original tree and sub-trees made from the full dataset (the one we found during "Finding the Best values for α" step) and we pick the tree that corresponds to the final value for α that we found ($\alpha = 10000$).



So, this sub-tree will be the **final, pruned tree**.

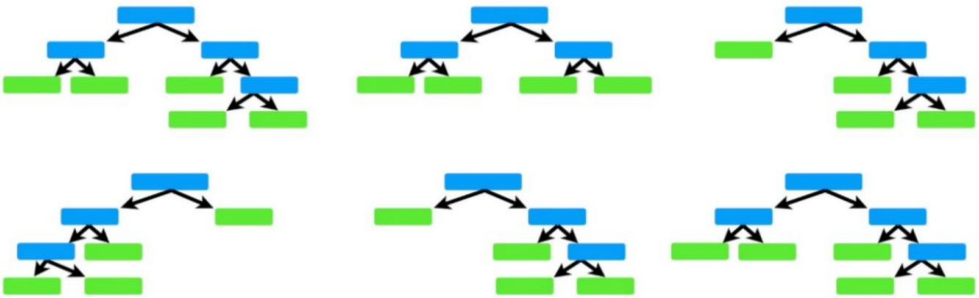
10 Random Forests

Decision Trees are easy to build, easy to use and easy to interpret, but in practice they have one aspect that prevents them from being the ideal tool to make predictions, namely **overfitting**. Decision Trees work great with the data used to create them, but **they are not flexible when it comes to classifying new samples**, so they don't generalize well, leading to a decrease in accuracy.

Random Forests, on the other hand, is an **ensemble learning** method that addresses this limitation by combining multiple decision trees to create a more robust and reliable predictive model. Instead of relying on a single decision tree, Random Forests generate a **"forest" of decision trees** during training and aggregate their outputs during prediction:

- For classification tasks, they take a majority vote among the trees.
- For regression tasks, they compute the average of all tree predictions.

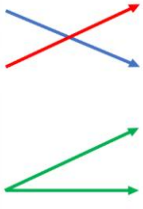
As we will see soon, thanks to the randomness introduced in both data sampling and feature selection during the individual tree construction, Random Forests enhance tree diversity, producing a more resilient model that is less prone to overfitting and which results in a **vast improvement in accuracy**.



Step 1: Create a bootstrapped dataset

Given a dataset, the first step necessary for building different trees is to create the so-called Bootstrapped datasets. A Bootstrapped dataset is a dataset that has exactly the same size (same number of samples and features) as the original dataset, with samples selected randomly from the original dataset with replacement. This means that the samples chosen randomly could also be picked more than once (green arrows)!

Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
No	No	No	125	No
Yes	Yes	Yes	180	Yes
Yes	Yes	No	210	No
Yes	No	Yes	167	Yes



Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
Yes	Yes	Yes	180	Yes
No	No	No	125	No
Yes	No	Yes	167	Yes
Yes	No	Yes	167	Yes

Step 2: Create a decision tree using a random subset of features (from the bootstrapped dataset) at each step

Starting from this bootstrapped dataset, the idea is to select a subset of the features and start the procedure of constructing the corresponding tree; we then continue in a similar manner by considering the other features (except the one chosen in the previous step) as shown in the following example.

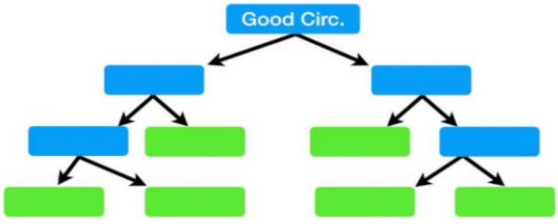
We start by selecting a subset of the features, in this case Good Blood Circulation and Blocked Arteries, as candidates for the root node.

Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
Yes	Yes	Yes	180	Yes
No	No	No	125	No
Yes	No	Yes	167	Yes
Yes	No	Yes	167	Yes

Imaging Good Blood Circulation did the best job separating the samples it becomes the root node, so we go ahead and select randomly other two features, in this case Chest Pain and Weight, as candidates to continue the building of the tree. Note that Good Blood Circulation is excluded from this and the next selections as we already use it.

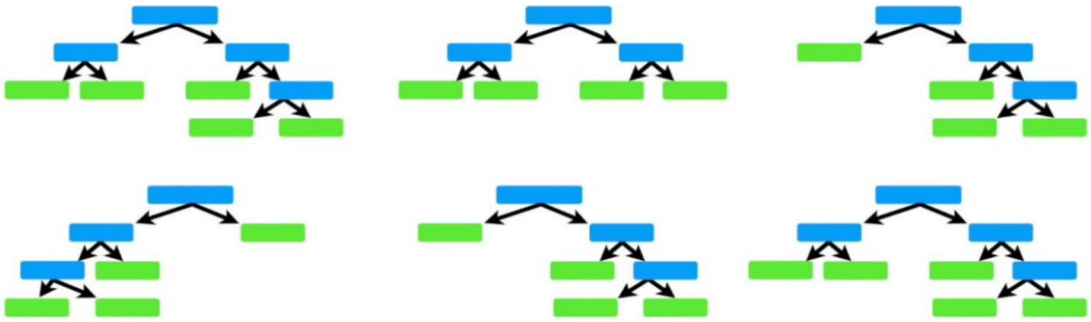
Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
Yes	Yes	Yes	180	Yes
No	No	No	125	No
Yes	No	Yes	167	Yes
Yes	No	Yes	167	Yes

We repeat this process considering a random subset of features from the remaining ones at each step until we build the whole tree.



Now go back to Step 1 and Repeat: Make a **new bootstrapped dataset** and **build a new tree** considering a subset of features at each step.

Ideally, we should do this 100 times, here we only have space to show 6 … but we get the idea.



Using different bootstrapped datasets and considering only a subset of features at each step results in wide variety of different trees (since based on the different data and features) and creates what is called a Random Forest.

10.1 Classify a New Sample with "Bagging"

Let's first see how to classify a new sample in a Random Forest.

Given a new sample we want to predict its classification value. As first thing we just take the new sample, we run it down in the first tree that we made, and we keep track of the result. Then we repeat this process for all other trees we made keeping track of the results.
After running the sample down all of the trees in the random forest, we see which classification option received more votes.

Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
Yes	No	No	168	

Heart Disease	
Yes	No
5	1

For example, in this case, "Yes" received the most votes, so we will conclude that the patient has heart disease.

More specifically, **b**ootstrapping the data plus using **agg**regate to make a decision is called "**Bagging**". So, what we did was classify a new sample with Bagging.

- ➔ **Bagging** (***Bootstrap Aggregating***) is when we predict the value of a target variable associated with a particular sample by **aggregating** the predictions obtained from different models built from **bootstrapped** datasets. In our case, such models are the trees that make up the forest, but more in general can be also other machine learning models.

10.2 Evaluate the Accuracy of a Random Forest: Out-of-Bag Error

Let's now see how to evaluate a Random Forest. Remember when we created the bootstrapped dataset, we allowed duplicate entries in it and, as result, one entry of the example wasn't included in the bootstrapped dataset.

Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
No	No	No	125	No
Yes	Yes	Yes	180	Yes
Yes	Yes	No	210	No
Yes	No	Yes	167	Yes

Typically, about **1/3** of the original data does not end up in the bootstrapped dataset.

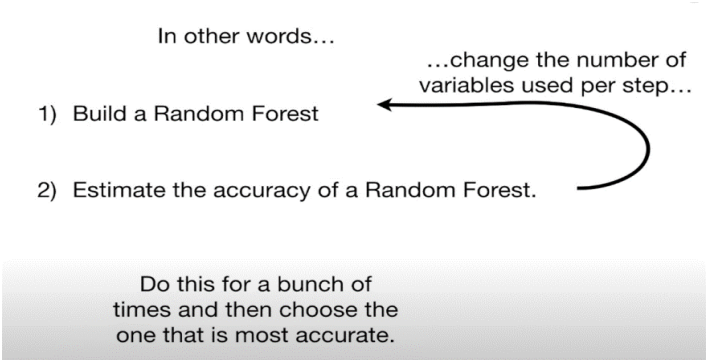
All the entries that didn't end up in the bootstrapped dataset form the so called **Out-of-Bag (OOB) Data**.

Each OOB sample is passed through only the trees that did not include it in their bootstrapped dataset. The predicted value from each of these trees is then compared to the true value of the OOB sample.

Ultimately, we can measure the so called **Out-Of-Bag error** which is computed as the overall proportion of misclassified samples or the average prediction error, depending on whether we are considering a classification or a regression task respectively.

Optimizing a Random Forest

At the end another thing that we can do is to compare the Out-Of-Bag error for a random forest built using only 2 variables (features) per step (as we did until now) to a random forest built using 3 variables for step, we test a bunch of different settings for it and choose the most accurate Random Forest.



The **number of features to consider in a subset** when creating a tree is a **hyperparameter** of the model. Typically, we start by considering a number equal to the **square root of all the features present** and then we try a few settings above and below that value.

10.3 Missing Values in Random Forest

Let's see what to do when there are missing values. Random Forest considers two types of missing values:

1. Missing values in the original dataset used to create a random forest.
2. Missing values in a new sample that we want to categorize through a random forest.

Missing Values in the Dataset

Suppose we got data for four patients, but for 4th patient we've got some missing values, one for Blocked Arteries and one for Weight.

Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
No	No	No	125	No
Yes	Yes	Yes	180	Yes
Yes	Yes	No	210	No
Yes	No	???	???	No

The general idea for dealing with missing values in this context is to make an initial guess that could be bad, then gradually refine the guess until it is (hopefully) a good guess.

Because the 4th patient has NO as value for Heart Disease …

Initial Guess - Categorical

… for Blocked Arteries we can consider the **<u>most common value</u> found in the other samples** that have NO as value for Heart Disease as initial guess.

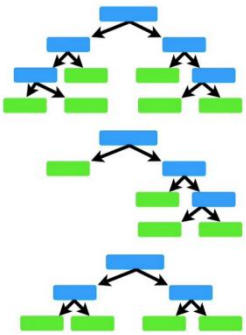
Initial Guess - Numerical

… for Weight we can consider the **<u>median value</u> among the ones found in the other samples** that have NO as value for Heart Disease as initial guess.

Now we want to refine these guesses.
We do this by first determining which samples are similar to the one with missing values.

To find similarity in samples we:

1. Build a random forest



2. Run all the data samples through each tree of the forest
3. Check if a data sample ended up at the same leaf of sample 4. In this case sample 3 ends up in the same leaf of sample 4, so it seems that they are in some way similar.

We keep track of similar samples using a Proximity Matrix.

A **Proximity Matrix** has a row for each sample and a column for each sample. In a proximity matrix we keep track of similarity by **adding a 1** to any pair of samples that end up in the same leaf node.

Running all data down the first tree we found that sample 3 and 4 both ended up in the same leaf node, so we put 1 to each pair of them:

	1	2	3	4
1				
2				
3				1
4			1	

Running all data down the second tree we found that samples 2,3 and 4 all ended up in the same leaf node, so we update the proximity matrix by adding 1 to each pair of them:

	1	2	3	4
1				
2			1	1
3		1		2
4		1	2	

We run the data down to all the other threes and the proximity matrix fills in:

	1	2	3	4
1		2	1	1
2	2		1	1
3	1	1		8
4	1	1	8	

Ultimately, we **normalize** the proximity matrix values <u>with respect to the total number of trees we have considered</u> (in this example, assume we had 10 trees):

	1	2	3	4
1		0.2	0.1	0.1
2	0.2		0.1	0.1
3	0.1	0.1		0.8
4	0.1	0.1	0.8	

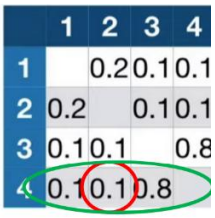
Now we use the proximity matrix obtained to compute the missing values for sample 4 to make better guesses.

- For Blocked Arteries, we calculate the **weighted frequency** of Yes or No, using proximity values as the weights:

Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
No	No	No	125	No
Yes	Yes	Yes	180	Yes
Yes	Yes	No	210	No
Yes	No	???	???	No

Compute the frequency
Yes = 1/3
No = 2/3

	1	2	3	4
1		0.2	0.1	0.1
2	0.2		0.1	0.1
3	0.1	0.1		0.8
4	0.1	0.1	0.8	



	1	2	3	4
1		0.2	0.1	0.1
2	0.2		0.1	0.1
3	0.1	0.1		0.8
4	0.1	0.1	0.8	

	1	2	3	4
1		0.2	0.1	0.1
2	0.2		0.1	0.1
3	0.1	0.1		0.8
4	0.1	0.1	0.8	

$$\text{Yes} = \frac{1}{3} \cdot proximity^{YES}$$

$$= \frac{1}{3} \cdot \frac{0.1}{01. + 0.1 + 0.8} = 0.03$$

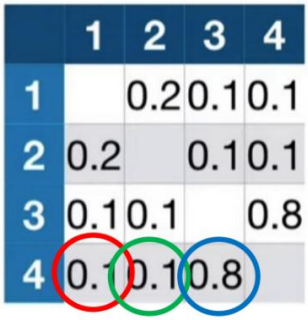
$$\text{No} = \frac{2}{3} \cdot proximity^{YES}$$

$$= \frac{2}{3} \cdot \frac{0.1 + 0.8}{01. + 0.1 + 0.8} = 0.6$$

"No" has a higher weighted frequency (=0.6), so we'll go with it. So, our new improved and revised guess based on proximity is "No" for Blocked Arteries.

- For Weight, we use the proximities to calculate the **weighted average**:

Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
No	No	No	125	No
Yes	Yes	Yes	180	Yes
Yes	Yes	No	210	No
Yes	No	???	???	No



	1	2	3	4
1		0.2	0.1	0.1
2	0.2		0.1	0.1
3	0.1	0.1		0.8
4	0.1	0.1	0.8	

$$w = 125 \cdot \frac{0.1}{0.1 + 0.1 + 0.8} + 180 \cdot \frac{0.1}{0.1 + 0.1 + 0.8} + 210 \cdot \frac{0.8}{0.1 + 0.1 + 0.8} = 198.5$$

So, our new improved and revised guess based on proximity is 198.5 for Weight.

Now that we've **refined** our guesses a little bit, we do the whole thing over again, so we build a new random forest, and we repeat all the previous steps* until the missing values converge, that is, until the values obtained no longer change, de facto testifying the convergence of the algorithm.

*Previous steps: run the data through the trees, recalculate the proximities and recalculate the new missing values guesses

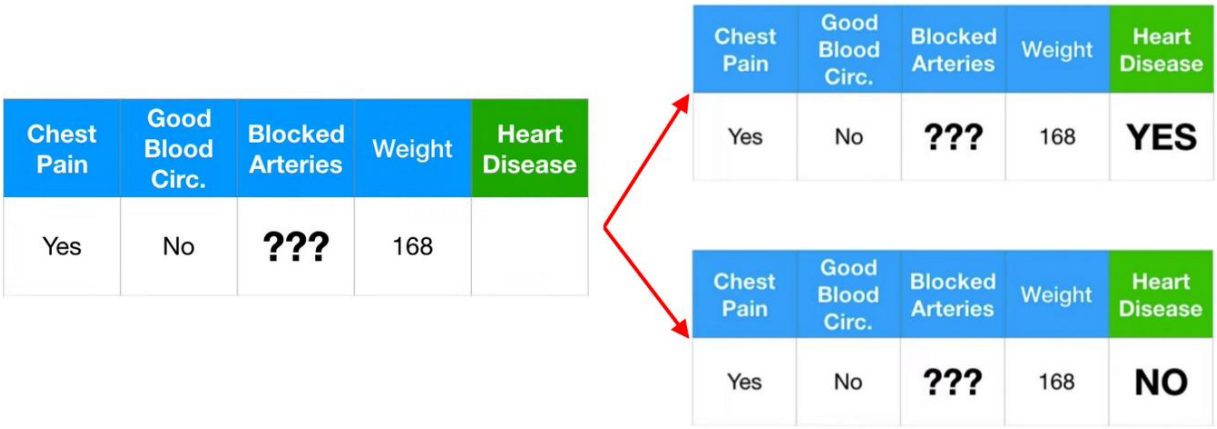
Missing Values in a New Sample

Imagine we had already a Random Forest with existing data and wanted to classify this new patient, but we have a missing value for Blocked Arteries.

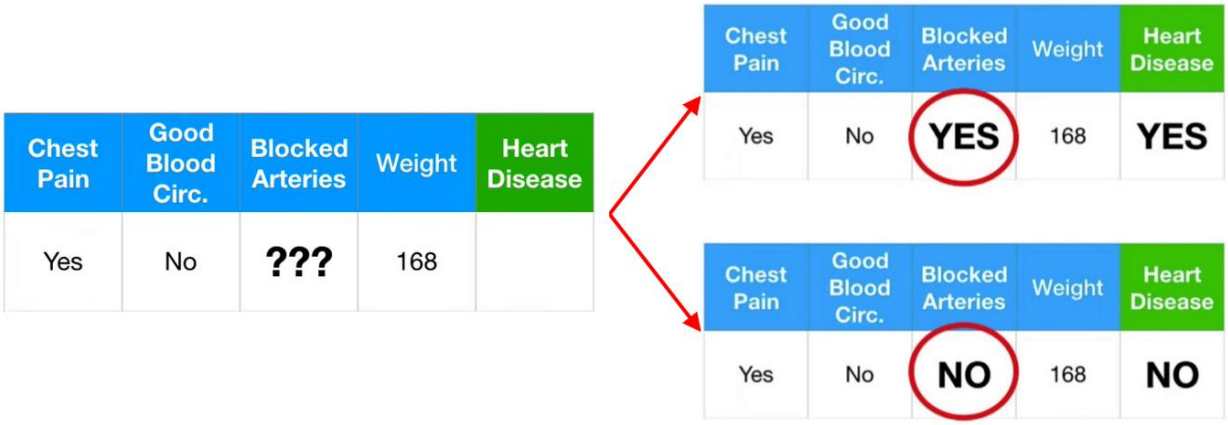
Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
Yes	No	???	168	

We need to make a guess about Blocked Arteries so we can run the patient down all the trees in the forest.

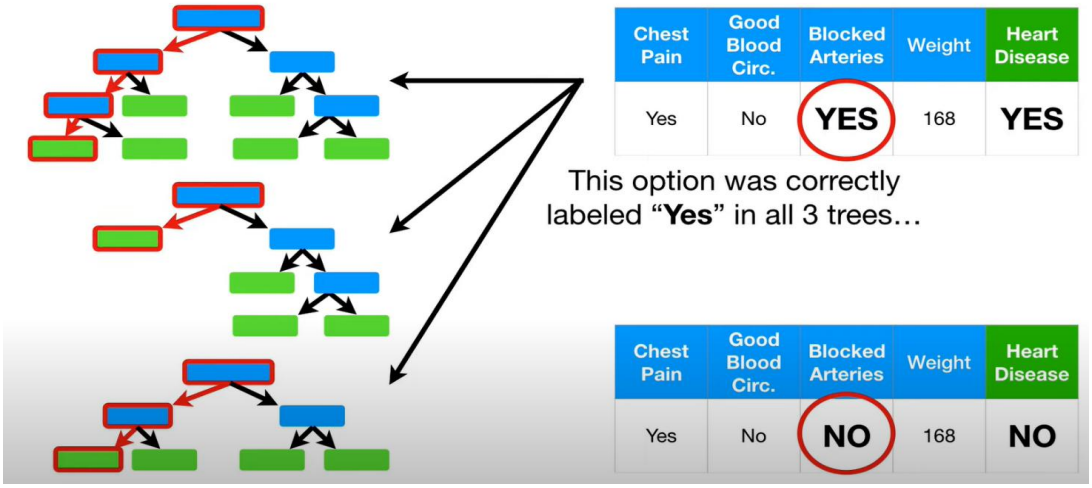
The first thing we do is create **two copies** of the data, one that has YES for Heart Disease and one that has NO for Heart Disease.

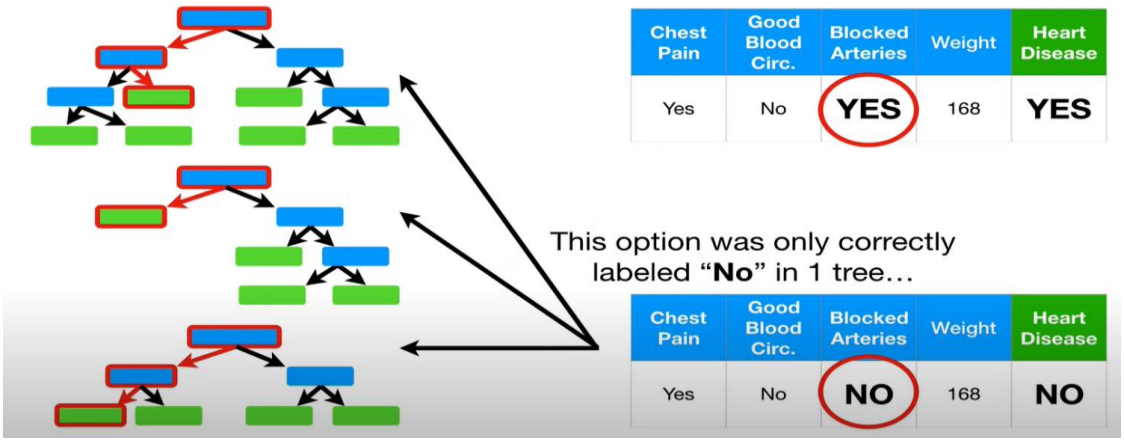


Then we use the iterative method for missing values in the dataset (the method we just talked about above) to make a good guess about the missing values. Suppose that these are the guesses that we come up with are:



Then we ran the two samples down the three in the forest and we count all the times the two samples are correctly labeled:





The option "Yes" wins because it was correctly labeled more than the other option, so we fill the missing value with Yes.

+ Boosting: XGBoost (extra)

Here I want also to introduce you to a powerful algorithm widely used to tackle machine learning tasks: **XGBoost**, short for **Ex**treme **G**radient **Boos**ting.

While bagging algorithms like Random Forest enhance accuracy by averaging predictions from multiple independently trained trees, XGBoost takes a different approach using a technique called **boosting**. In boosting, trees are built **sequentially**, with each new tree focusing on correcting the errors of its predecessors, resulting in a strong learner from a series of weaker ones.

XGBoost is an optimized implementation of gradient-boosted decision trees, offering several advanced features. These include regularization techniques to prevent overfitting, parallelized tree construction for faster training, and robust handling of missing data. Thanks to these enhancements, XGBoost has become a top choice for solving complex machine learning problems efficiently and effectively.

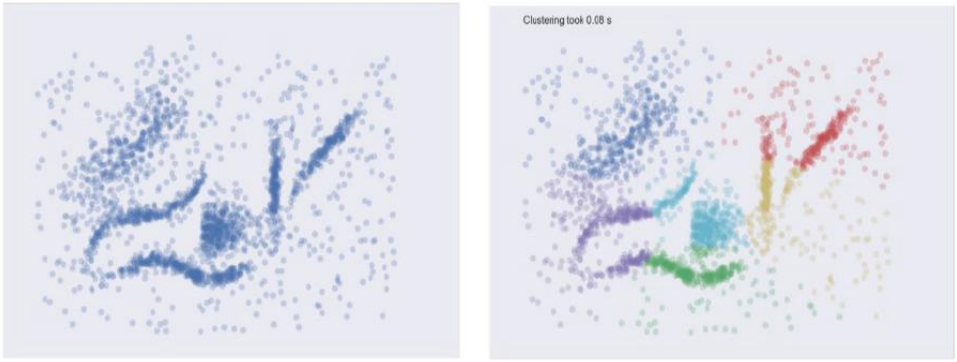
Unsupervised Learning

Unlike Supervised Learning, in **Unsupervised Learning** tasks there is no information about the classes belonging to the samples or in general on the output corresponding to a certain input (**no labels** are given).

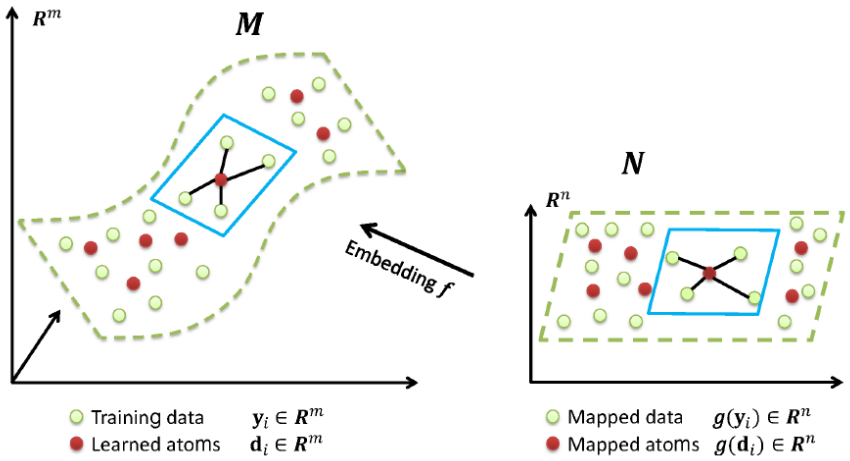
The available data only concerns the set of features that describe each sample.

The goal is to identify common patterns, behaviors or characteristics of the data. Some unsupervised learning tasks are:

- **Clustering**: an unsupervised learning task with the goal of group samples that have similar characteristics. Notably, this task is not deterministic; in fact, starting from the same dataset, using different clustering algorithms leads to very different results.



- **Dimensionality reduction**: an unsupervised learning task with the goal of reducing the dimensionality of the data while preserving the essential information.



11 Clustering

Clustering is an unsupervised learning task that has the purpose of dividing the samples into "natural" groups (clusters) such that:

- Samples in the same cluster are similar to each other (high intra-cluster similarity)
- Samples in different clusters are dissimilar (low inter-cluster similarity)

11.1 K-Means

K-means is one of the most popular clustering algorithms. The idea behind K-means is to determine the best approximation of the centroid (center point) of the clusters to determine consequently the best approximation of the clusters themselves.

The algorithm starts with an initial assumption, namely, that we have a certain number of "K" clusters (hyperparameter) and, consequently, that we necessarily have K **centroids** $\mu_1, \mu_2, \dots, \mu_K$, initialized in the space. So, it guesses the initial center of the clusters (centroids) by setting them equal to the values of K training examples chosen randomly.

At this point the algorithm assigns each point $x^{(i)}$ of the dataset to a given cluster k if its distance from the centroid of that cluster μ_k is the closest one compared to the centroids of the other clusters $\mu_1, \mu_2, \dots, \mu_K$.

Then it chooses as new centroids μ_k for each cluster so obtained the **midpoints** (the mean of the data points assigned to that cluster). These becomes the new geometrical centers of the clusters obtained.

At this point, the distance of all data points for each cluster from the new centroids is calculated again, and consequently, some data points will migrate from one cluster to another, and then again new centroids are assigned as midpoints of the cluster so obtained. This is repeated until the algorithm converges, that is, until there are no more switches of data points between clusters.

K-means clustering algorithm can be summarized as follows:

Randomly initialize K cluster centroids $\mu_1, \mu_2, \dots, \mu_K$.

Repeat {

for $i = 1$ to m

$$C_k^{(i)} := arg \min_{k \in \{1,\dots,K\}} d\left(x^{(i)}, \mu_k\right)$$

(Here $C_k^{(i)}$ represents the assignment of the point $x^{(i)}$ to the cluster of index k (C_k). It is assigned to the cluster having the closest centroid μ_k based on the distance metric d.

***Note**: Therefore, $C_k^{(i)}$ indicates the assignment of a point to cluster C_k, while C_k is the overall cluster, i.e. contains all the datapoints appertaining to that cluster)*

for $k = 1$ to K

$$\mu_k := \frac{1}{|C_k|} \sum_{x^{(i)} \in C_k} x^{(i)}$$

(Here μ_k represents the new centroid found for the cluster C_k. It is assigned as the mean of the points of cluster C_k.

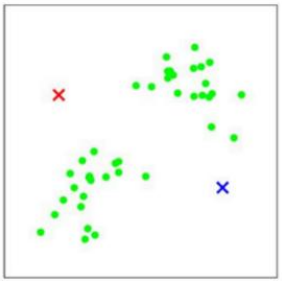
Note: $|C_k|$ is the module, i.e. it indicates the number of data points belonging to cluster C_k)

So fixed K, the number of clusters that we want to use to cluster the data, we initialize the cluster centroids by setting them equal to the values of K training examples chosen randomly (with $K < m$).

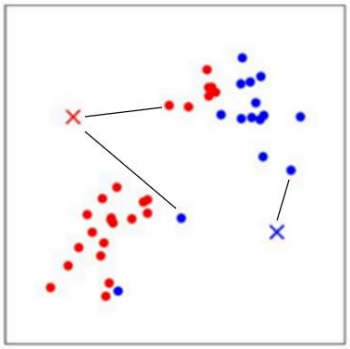
Then the inner-loop of the algorithm repeatedly carries out two steps:

- "Assigning" each training example $x^{(i)}$ to the cluster C_k with the closest cluster centroid μ_k.
- Moving each cluster centroid μ_k to the mean of the points assigned to it.

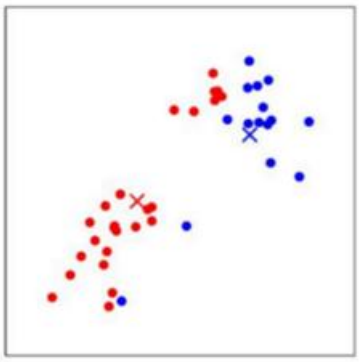
Example: Consider having the following data points shown in Figure and want to separate by means of 2 clusters ($K = 2$). As first step we start by randomly selecting two samples as centroids. In Figure, the data are represented as green dots and the first two centroids are represented as red and blue crosses.



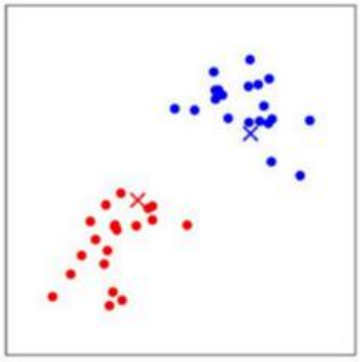
At this point we assign each point to a given cluster based on the **minimum distance** from the centroids of the two clusters. In Figure we paint each sample with the same color as the cluster centroid to which is assigned.



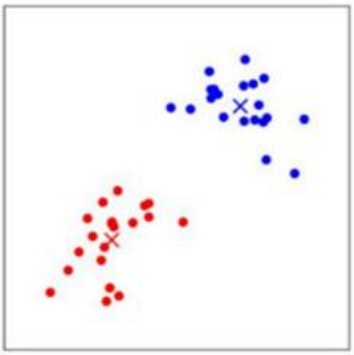
For each cluster, we compute the mean of its points to find its **new centroid**.



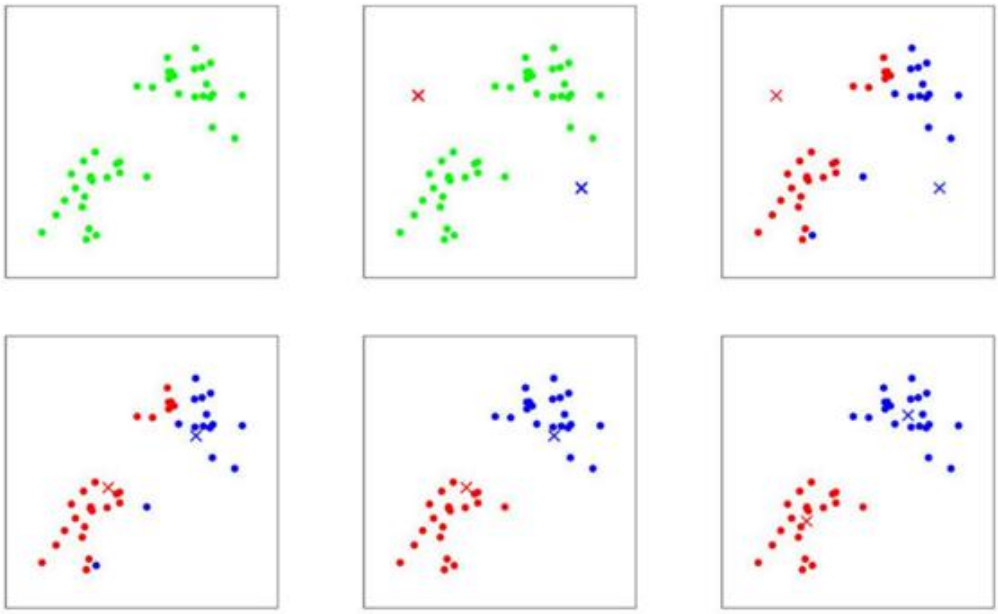
We recompute the distances of each point from the two clusters, and we reassign them to the closest cluster. As you can see some data points will migrate from one cluster to another.



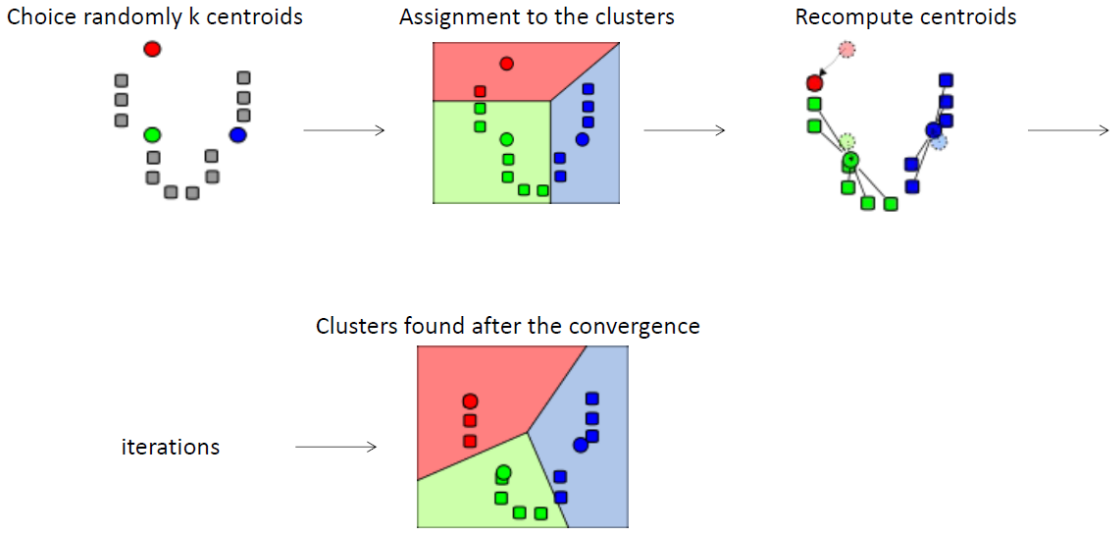
Again, we repeat this process until we **converge**, finding centroids for which there are no more switches of data points between clusters.



The complete illustration of running K-means algorithm is shown below:



In the Figure below is also shown another example with three clusters ($K = 3$):



K-Means - Expectation Maximization

The K-means algorithm itself can be interpreted as an application of the Expectation Maximization (EM) algorithm. I recommend revisiting this section after exploring the EM algorithm, which is introduced following the discussion on Gaussian Mixture Models.

From an EM viewpoint, K-means seeks to partition a dataset into K clusters by defining a binary assignment matrix $[r_{ij}]$ having one row for each of the m observations, and one column for each of the K clusters. Specifically, $r_{ij} = 1$ indicates that the data point i is assigned to cluster j, while $r_{ij} = 0$ means it is not assigned to that cluster. As we will see soon, we will aim to iteratively refine this r_{ij} by assigning each data point i to **exactly one cluster $\boldsymbol{k}$** for which only that r_{ik} is non-zero.

At this point we can define the cost function of K-means as the ***distortion measure***, given by:

$$\boldsymbol{J = \frac{1}{m}\sum_{k=1}^{K}\sum_{i=1}^{m} r_{ik}\left\|x^{(i)} - \mu_k\right\|^2}$$

which represents the sum of the squares of the distances of each data point to its assigned cluster centroid μ_k.

Our goal is to find values for r_{ik} and μ_k so as to minimize J. We can do this through an iterative procedure in which each iteration involves two successive steps corresponding to successive optimizations with respect to r_{ik} and μ_k. We start by choosing some initial values for the μ_k randomly and then we perform:

- **E-step**: In this first phase we minimize J with respect to the r_{ik}, keeping the μ_k fixed. Because J is a linear function of r_{ik}, this optimization can be performed easily to give a closed form solution. We can optimize for each i separately by choosing r_{ij} to be 1 for whichever value of j gives the minimum value of $\left\|x^{(i)} - \mu_j\right\|^2$. In other words, we simply assign the i-th data point to the closest cluster centroid. More formally, this can be expressed as:

$$\boldsymbol{r_{ik} \rightarrow \begin{cases} 1, & if\ k = \underset{j}{\operatorname{argmin}}\left\|x^{(i)} - \mu_j\right\|^2 \\ 0, & otherwise \end{cases}}$$

- **M-step**: In the second phase we minimize J with respect to the μ_k, keeping r_{ik} fixed. Since J is a quadratic function of μ_k, and it can be minimized by setting its derivative with respect to μ_k to zero giving:

$$2\sum_{i=1}^{m} r_{ik}\left(x^{(i)} - \mu_k\right)$$

  which we can easily solve for μ_l to give:

$$\boldsymbol{\mu_k = \frac{\sum_{i=1}^{m} r_{ik}x^{(i)}}{\sum_{i=1}^{m} r_{ik}}}$$

  The denominator in this expression is equal to the number of points assigned to cluster k (i.e. $|C_k| = \sum_{i=1}^{m} r_{ik}$), and so this result has a simple interpretation, namely set μ_k equal to the mean of all of the data points $x^{(i)}$ assigned to cluster k (as we already stated above):

$$\mu_k := \frac{1}{|C_k|}\sum_{x^{(i)} \in C_k} x^{(i)}$$

  Note: Moreover, is for this reason, that the procedure is known as the "*K-means*" algorithm.

We repeat these two steps until convergence is reached.

These two phases of *re-assigning data points to clusters* and *re-computing the cluster means* are repeated in turn until convergence is reached, i.e. until there is no further change in the assignments (or until some maximum number of iterations is exceeded).

K-Means – Random Initialization

One of the main limitations of the k-means clustering algorithm lies precisely in its strong dependence on the choice of starting centroids. The initialization of the latter is, in fact, done randomly. Starting from K randomly chosen data points, some of them are randomly selected as initial centroids depending on the number of clusters. In particular:

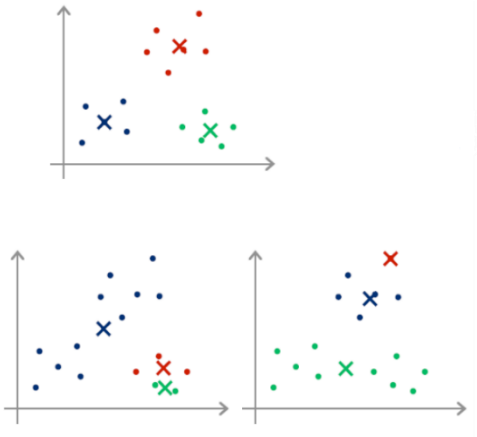
- We should have $K < m$ (of course, the number of clusters must always be less than the number of data points)
- Randomly pick K training examples
- Set the centroids equal to these examples

! Note that you set centroids equal to the randomly chosen examples because this avoids initially choosing centroids completely randomly in space, which may perhaps be in regions totally outside the space of residence of the data points.

K-means – Convergence

Is the K-means algorithm guaranteed to converge?

Because each phase reduces the value of the objective function J, **convergence of the algorithm is assured**. In fact, we saw above that K-means repeatedly minimizes J with respect to r_{ik} while holding μ_k fixed, and then minimizes J with respect to μ_k while holding r_{ik} fixed. Thus, J must monotonically decrease, and the value of J must converge. However, it may converge to a local rather than global minimum of J. In other words, k-means can be susceptible to **local optima**, so it will reach a good clustering but maybe it will not be the optimal clustering.



Very often k-means will work fine and come up with very good clusters despite this. But if you are worried about getting stuck in bad local minimum, one common thing to do is **run k-means many times**, e.g. 100 times (using different random initial values for the cluster centroids μ_k). Then, out of all the different clustering so obtained (→), pick the one that gives the lowest cost J.

K-means algorithm to avoid local-minima:

For $e = 1$ *to* 100 *{*

Randomly initialize k-means

Run k-means. Get all the final datapoints assignment and centroids values $C^{(m)}, \dots, C^{(m)}, \mu_1, \dots, \mu_K$

Compute $J\left(C^{(m)}, \dots, C^{(m)}, \mu_1, \dots, \mu_K\right)$

}

Choose clustering that gave the lowest cost J *over the 100 epochs.*

K-Means – PROS/CONS:

- PROS
  - Quite simple and computational efficient.
  - Always terminates.

- CONS:
  - Finding the optimal clustering is not guaranteed. This is because it may obtain different results for different runs on the same dataset because the first K centroids are chosen randomly.
  - Not applicable to categorical values (non-numeric).
  - Requires to specify K.
  - Sensible to noise and outliers.

11.2 K-Medoids

K-Medoids (also called **Partitioning Around Medoid** - **PAM**) is an algorithm similar to K-means but with a crucial difference: K-medoids chooses in each iteration actual data points as centers (medoids), in contrast to K-means where the center of a cluster is not necessarily one of the input data points (it is the mean between the points in the cluster).

The **medoid** of a cluster is defined as the point in the cluster whose sum of dissimilarities to all the points in the cluster is minimal, that is, it is a most centrally located point in the cluster. This behavior makes K-medoids algorithm more robust to noise than the K-means algorithm.

Different dissimilarity measures can be used, in general distance metrics such as Euclidean, Manhattan, Chebyshev and so on.

K-medoids Clustering algorithm:

1. *Randomly initialize K clusters medoids ($\mu^1, \dots, \mu^k, \dots, \mu^K$) and define $M = \bigcup_{k=1}^{K} \mu^k$*
2. *Associate each $x^{(i)} \in X$ to the cluster of index k having the closest medoid μ^k*
3. *$Cost$ = sum of the distances of samples from their medoids*
4. *$NewCost = 0$*
5. ***while** ($NewCost \leq Cost$) {*

   ***for** $k = 1$ **to** K*

   ***for** $x^{(i)} \in X - M$*

   swap $x^{(i)}$ and μ^k

   ***repeat** steps 2) and 3) to obtain $NewCost$ = sum of the distances of samples from their medoids*

   ***if** ($NewCost > Cost$) **then** undo swap*

   ***else** $Cost = NewCost$*

}

Explanation

1. Initialize: Randomly select K points out of the m data points as the medoids μ_k. Denote the set containing all medoids as $M = \bigcup_{k=1}^{K} \mu_k$.

2. Associate each data point $x^{(i)} \in X$ to the cluster of index k having the closest medoid μ^k by using any common distance metric method (e.g. Euclidean distance).

3. Compute the Cost as the sum of distances all data points to their assigned medoids.

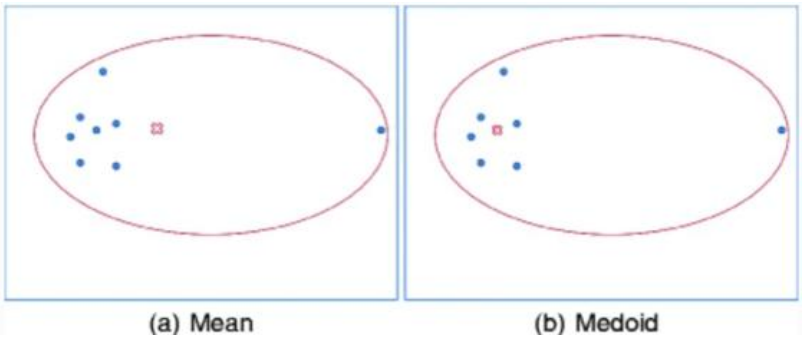
5. Create a counter variable NewCost and initialize it to zero (it will iteratively be updated and be used to track the new cost each time computed).

6. While the NewCost decreases: For each cluster k and for each data point $x^{(i)}$ which is not a medoid (don't appertain to M):

- Swap $x^{(i)}$ and μ_k, and repeat steps 2 and 3, which means based on this new set of medoids (obtained after the swap) perform the re-assignment and re-compute the cost to obtain the NewCost.
- If the NewCost is higher than that in the previous step (Cost) undo the swap, otherwise (confirm the swap) update the cost as equal to this lower NewCost.

K-Medoids – PROS/CONS

- PROS
  - It is simple to understand and easy to implement.
  - K-Medoid algorithm is fast and converges in a fixed number of steps.
  - It is less sensitive to outliers than K-means → The mean of the data points is a measure that gets highly affected by the extreme points, so in K-means algorithm the centroid may get shifted to a wrong position and hence result in incorrect clustering if the data has outliers because then other points will move away from . On the contrary, a medoid in the K-medoids algorithm is the most central element of the cluster, such that its distance from other points is minimum. Since medoids do not get influenced by extremities, the K-medoids algorithm is more robust to outliers and noise than K-means algorithm. The following Figure explains how mean's and medoid's positions can vary in the presence of an outlier.



Moreover, note that this example shows extremely well how with K-means can be chosen a point as centroid that actually is not an input data point, while in K-medoids we don't have this problem.

- CONS:
  - It is not suitable for clustering non-spherical (arbitrarily shaped) groups of objects. This is because it relies simply on minimizing the distances between the non-medoid points and the medoid, so it doesn't have a way to consider the geometry of the groups.

- Finding the optimal clustering is not guaranteed.
- Not applicable to categorical values (non-numeric).
- Requires to specify K.
- Yet sensible to noise and outliers.
- Fails for non-linear data set.
- Computationally more expensive than K-means.

11.3 Gaussian Mixture of Models (GMMs)

In the K-means and K-medoids algorithms we've discussed so far, there are two important limitations:

- Each data point is simply assigned to the nearest cluster based on its distance from centroid/medoid but what if data points form groups that overlap? In the intersection region it becomes hard to know which assignment is right since both are plausible.
- What if our clusters are defined by some non-circular shape?

The two algorithms in this case won't be able to discover this assignment. To face these issues Gaussian Mixture Models come in help.

Gaussian Mixture Models (GMMs) represent another common and important technique used in clustering to model clusters as a composition of Gaussian distributions, so we have not only their mean, but also a covariance that describes their non-spherical shapes.

The main idea is to model data with a mixture of gaussian distributions, creating a gaussian mixture model, and then fit the model by maximizing the likelihood of the observed data through the so-called Expectation Maximization (EM) algorithm. At the end of this process, we will obtain a probability model that will assign to each data point a probability to appertain to a specific cluster.

Let's delve deeper into how GMMs can be adopted to make clustering.

Recap of Gaussian Distribution

Since we are going to extensively use Gaussian distributions, let's quickly review their fundamentals. A **Gaussian distribution** (or **Normal distribution**) is a continuous probability distribution that is characterized by its symmetrical **bell shape**. If a variable $x \in \mathbb{R}$ follows a Gaussian distribution, its **probability density function** (**pdf**) is defined as:

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

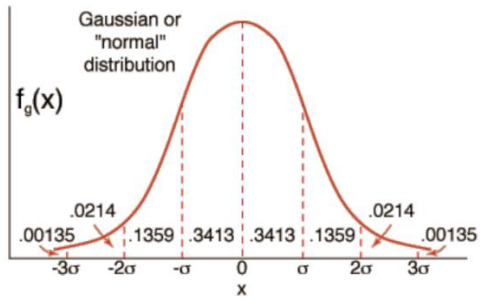
This can also be expressed compactly as:

$$x \sim N(\mu, \sigma^2)$$

where μ is the **mean** (expected value) and σ^2 is the **variance** of the distribution. Note that μ and σ^2 here are scalars.

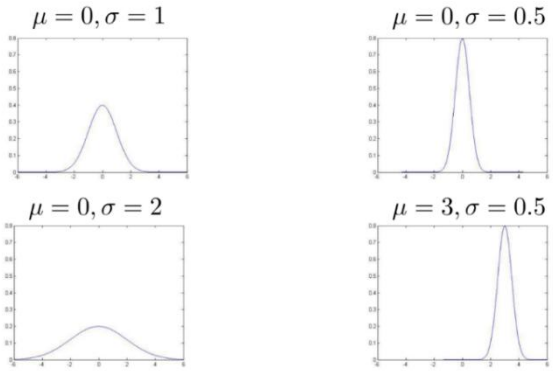
Note: We like Gaussians because they have several nice properties, for instance marginals and conditionals of Gaussians are still Gaussians. Thanks to these properties Gaussian distributions have been widely used in a variety of algorithms and methods.

In the Figure below is shown a normal distribution with $\mu = 0$.



However, changing μ and σ values we can see:

- change μ while keeping fixed σ changes the **width** of the gaussian.
- change σ while keeping fixed μ changes the **height** of the gaussian.



Univariate & Multivariate Gaussian Distribution

A random variable that follows a normal distribution ($x \sim N(\mu, \sigma^2)$) has parameters μ and σ computed as:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \qquad \sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu\right)^2$$

Since this Gaussian distribution is defined over a single random variable is called **univariate Gaussian distribution**.

The **multivariate Gaussian distribution** is a multidimensional generalization of the one-dimensional Gaussian distribution. It represents the distribution of a multivariate random variable, that is made up of multiple random variables which can be correlated with each other. The **multivariate** Gaussian distribution can be defined with its joint probability density function given by:

$$p(x; \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} e^{\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right)} = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} e^{\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right)}$$

Like the univariate normal distribution, the multivariate normal is defined by sets of parameters: the mean *vector* $\mu \in \mathbb{R}^n$ which gives us the expected value of the distribution and the covariance matrix $\Sigma \in \mathbb{R}^{n \times n}$, which each of its element measures how two random variables depend on each other and how they change together.

Note: The value $|\Sigma|$ is the determinant of Σ, and n is the number of dimensions $x \in \mathbb{R}^n$.

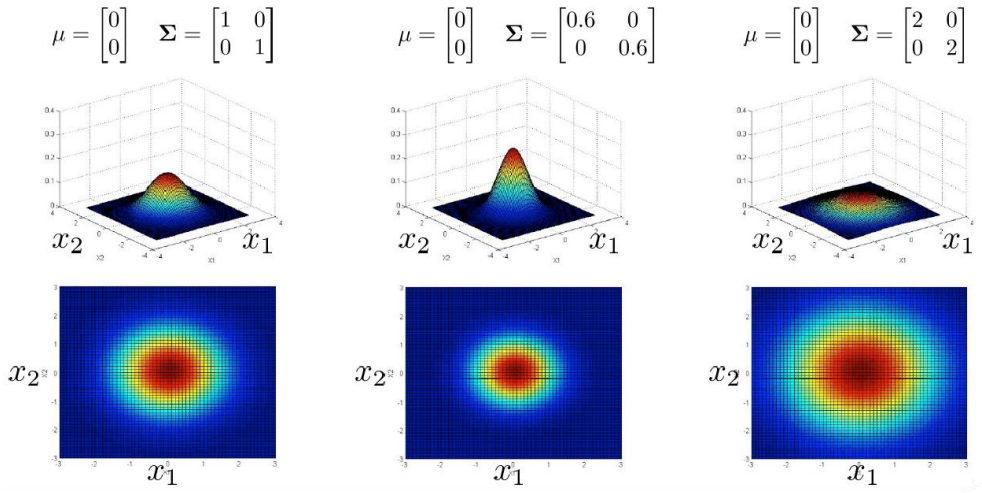
Plotting a multivariate distribution of more than two variables might be hard. Therefore, let's give an example of bivariate normal distribution.

Assume a 2-dimensional random vector $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ has a normal distribution $N(\mu, \Sigma)$ where:

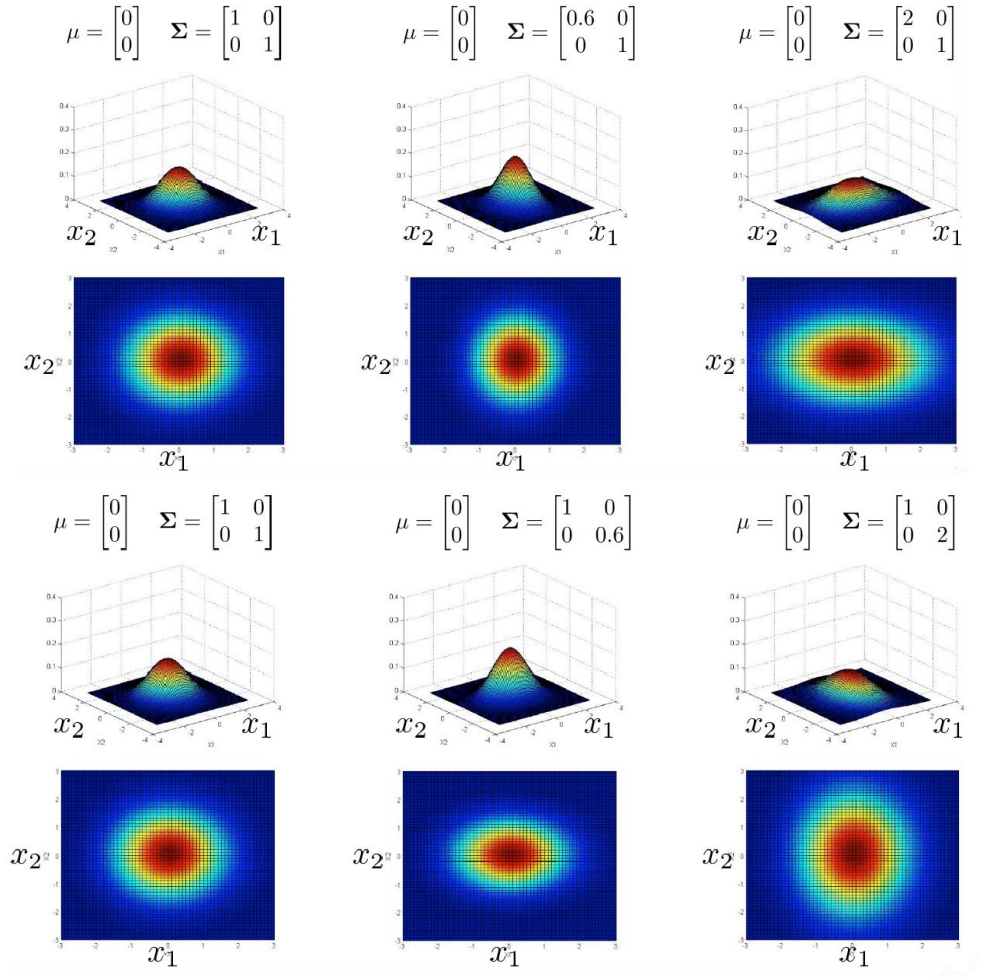
$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix} \qquad \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$$

Clearly, depending on the values in the mean vector and in the covariance matrix, the Gaussian distribution on the n-dimensional (hyper)plane of the data takes different courses:

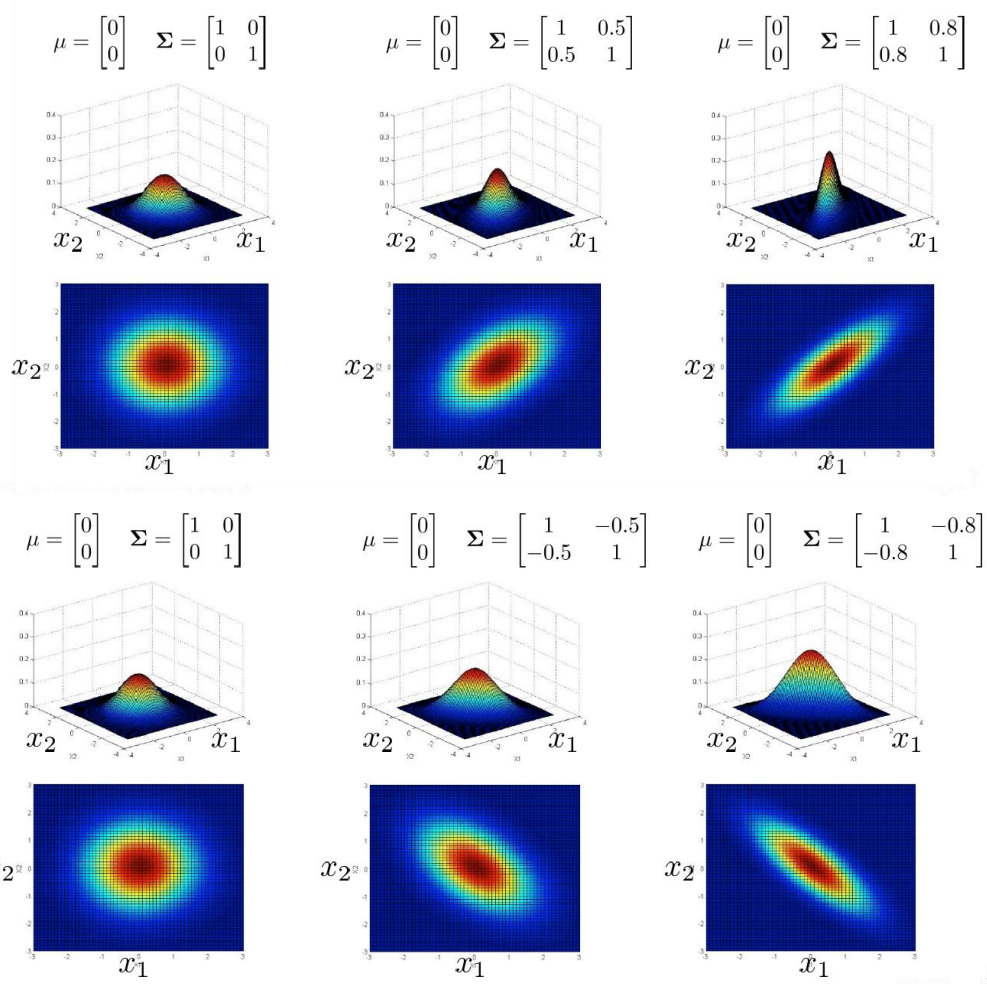
- Changing the **covariance matrix** has the effect of making the Gaussian more "spread-out" as Σ becomes larger, while it becomes more "compressed" as Σ becomes smaller. In particular:
    - The Gaussian is regular if the elements of the main diagonal are equal, and the secondary diagonal is zero.



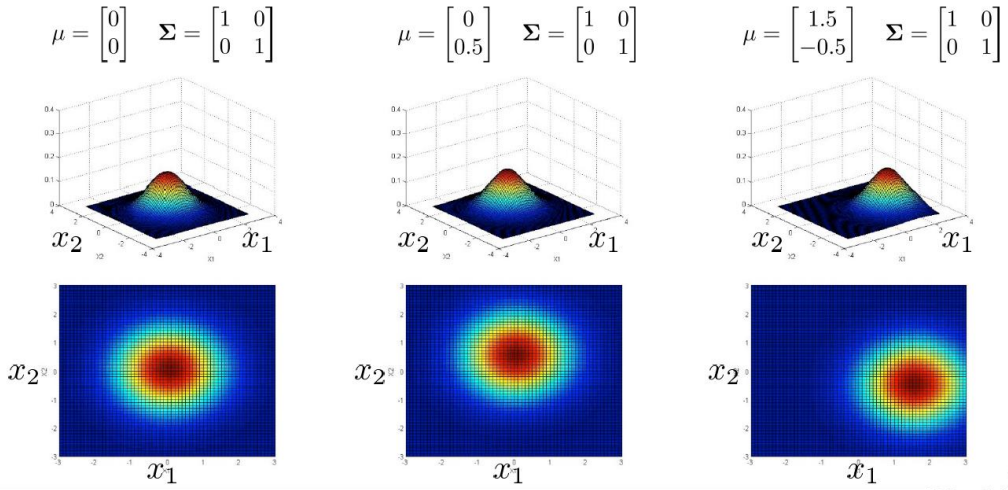
    - The Gaussian begins to be irregular if the elements of the main diagonal are different.



- The Gaussian begins to rotate in the presence of non-zero elements along the secondary diagonal.



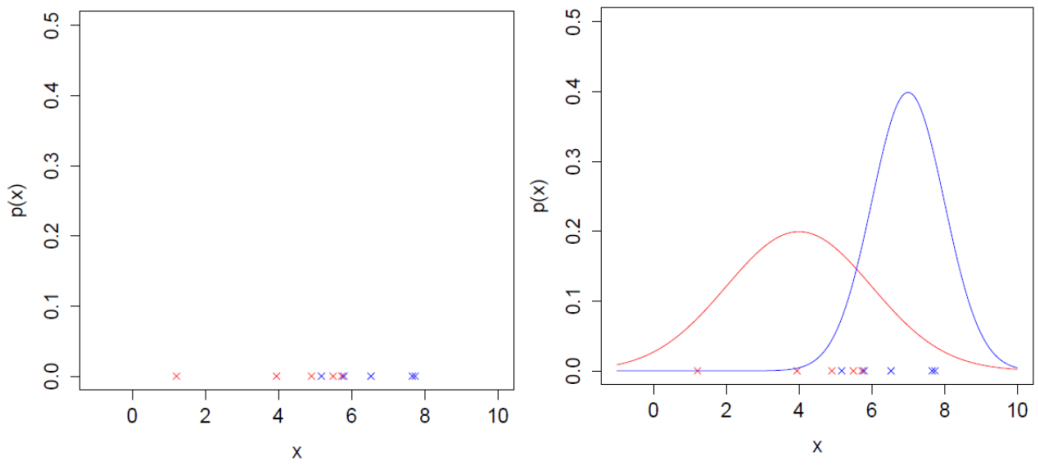
- While acting on the **mean vector** the Gaussian begins to **translate** (shift).



Mixture of Gaussians

In light of what has just been described, how then is it possible to use such distributions to perform a clustering?

The idea is to model the data as being generated from a mixture of multiple distributions, with each distribution modelling the probability of data belonging to a specific cluster. Specifically, in a **Gaussian Mixture Model** (**GMM**) we use a **mixture of Gaussians** where each Gaussian distribution will represent a particular cluster. Each of them is characterized by its own mean and variance (or covariance, in the multivariate case).



The fact that we represent the data as a mixture of Gaussians gives us the possibility to assign to each datapoint a given probability of appertaining to a given Gaussian (i.e. cluster).

For example, look at the Figure above. Considering the point at the value "4" in that case it would have a certain probability of belonging to the red cluster and a certain probability of belonging to the blue cluster. However, we assume that it actually belongs to the red cluster since the probability that it belongs to the red cluster is greater than the probability relating to the blue cluster.

Probabilistic Representation of the Mixture: Having assumed that the distribution that the data assumes is the sum of several gaussian distributions, we can then say that:

The probability of a single data point $x^{(i)}$ belonging to the mixture of Gaussians is equal to the sum of the joint probabilities of the same data point $x^{(i)}$ to belong to each of the K Gaussians of the mixture:

$$p\left(x^{(i)}\right) = \sum_{k=1}^{K} p\left(x^{(i)} \cap z = k\right)$$

Using Bayes' theorem, we can rewrite this probability explicitly as:

$$p\left(x^{(i)}\right) = \sum_{k=1}^{K} p\left(x^{(i)} | z = k\right) p(z = k)$$

At this point, we define:

- $p_k\left(x^{(i)}\right) = p\left(x^{(i)} | z = k\right)$ the probability density of the data point $x^{(i)}$ given that it belong to the k-th Gaussian.
- $\pi_j = p(z = k)$ the prior probability of picking the k-th Gaussian among all the K Gaussians. The parameters π_ks are called **mixing coefficients**. Note: These coefficients satisfy $\sum_{k=1}^{K} \pi_k = 1$.

From which, substituting, we get:

$$p\left(x^{(i)}\right) = \sum_{k=1}^{K} \pi_k \, p_k\left(x^{(i)}\right)$$

We can further make explicit the parameters of the gaussians (mean and variance), through a theta parameter:

$$p\left(x^{(i)}|\theta\right) = \sum_{k=1}^{K} \pi_j \, p_k\left(x^{(i)}|\theta_k\right)$$

where θ represents the parameters of all the gaussians and so θ_k represents the parameters of the k-th gaussian.

For example, if we have three gaussians will have

$$\theta = \{\mu_1, \mu_2, \mu_3, \sigma_1, \sigma_2, \sigma_3\}$$

$$\theta_1 = \{\mu_1, \sigma_1\} \qquad \theta_2 = \{\mu_2, \sigma_2\} \qquad \theta_3 = \{\mu_3, \sigma_3\}$$

Each of the multivariate gaussian under the mixture perspective can be defined as:

$$p_j(x; \theta_k) = p_j(x; \mu_k, \Sigma_k) = \mathcal{N}\left(x^{(i)}|\mu_k, \Sigma_k\right) = \frac{e^{\left(-\frac{1}{2}(x-\mu_k)^T \Sigma_k^{-1}(x-\mu_k)\right)}}{\sqrt{(2\pi)^n |\Sigma_k|}}$$

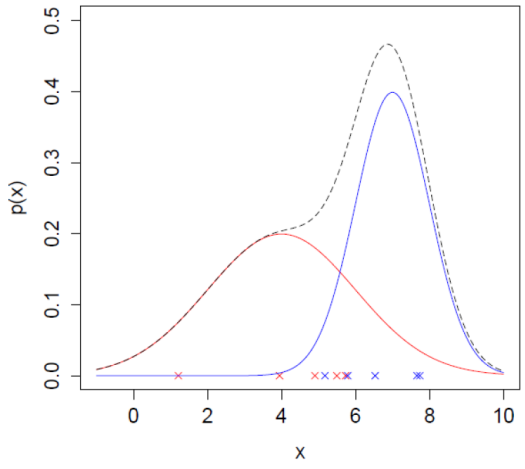
where:

- μ_k is the vector of the means of the k-th gaussian
- Σ_k is the covariance matrix of the k-th Gaussian
- n is the number of features

Note: Basically, we obtained that $p\left(x^{(i)}\right) = \sum_{k=1}^{K} \pi_j \, \mathcal{N}\left(x^{(i)}|\mu_k, \Sigma_j\right)$ where:

- each Gaussian density $\mathcal{N}\left(x^{(i)}|\mu_k, \Sigma_k\right)$ is called a ***component*** of the mixture and has its own mean μ_k and covariance Σ_k.
- π_k represents the **weight** of a given component indicating how much that Gaussian component contributes to the overall data distribution. For instance, if the first component has a weight of 0.2977, it will imply that roughly 29.77% of the data belongs to the first Gaussian component.

Ultimately, what we are interested in knowing is how to identify the appropriate Gaussian components that collectively form a mixture capable of accurately approximating the true data distribution.



This involves determining the parameters of each Gaussian, as these parameters define the shape and characteristics of the individual components. Specifically, we aim to estimate μ_k and Σ_k for each component $k = 1, \dots, K$. But also determining the mixing coefficients π_k.

To frame this task mathematically, we model the data using a likelihood function that captures the probability of the observed dataset under the Gaussian mixture model. If we assume the data points $X = \{x^{(1)}, x^{(2)}, \dots, x^{(i)}, \dots, x^{(m)}\}$ are drawn independently and identically distributed (i.i.d.), the likelihood of the data can be expressed as:

$$p(X|\pi, \mu, \Sigma) = \prod_{i=1}^{m} p\left(x^{(i)}|\theta\right)$$

Therefore, this likelihood serves as the **objective function** that we aim to maximize to find the optimal parameters of the GMM. By doing so, we determine the set of parameters μ_k, Σ_k *and* π_k that best explain the observed data distribution.

Let us take a step back to understand how these parameters are typically calculated.

Let us define C_k as the k-th cluster, and $|C_k|$ as the number of points that belong to it. Normally, if we already knew which point belongs to which cluster, we would straightforwardly calculate the parameters as:

- For univariate

$$\hat{\mu}_k = \frac{1}{|C_k|} \sum_{x^{(i)} \in C_k} x^{(i)} \qquad \hat{\sigma}_k^2 = \frac{1}{|C_k| - 1} \sum_{x^{(i)} \in C_k} \left(x^{(i)} - \hat{\mu}_k\right)^2$$

- and so, for multivariate:

$$\hat{\mu}_k = \frac{1}{|C_k|} \sum_{x^{(i)} \in C_k} x^{(i)} \qquad \hat{\Sigma}_k = \frac{1}{|C_k| - 1} \sum_{x^{(i)} \in C_k} \left(x^{(i)} - \hat{\mu}_k\right)\left(x^{(i)} - \hat{\mu}_k\right)^T$$

However, since we do not possess this information (we do not know the cluster memberships C_k of the data points), we can't directly compute these parameters using these simple closed-form expressions, so we need to look for an alternative way to determine them. For this purpose, the Expectation-Maximization Algorithm (EM) algorithm is used.

Expectation-Maximization Algorithm

The **Expectation-Maximization Algorithm (EM)** is an iterative method for finding the maximum likelihood estimates of a set of parameters. The algorithm consists of two steps: the **E-step** in which a function for the expectation of the likelihood is computed based on the current parameters, and an **M-step** where the parameters found in the first step are maximized. For convergence, we can check the likelihood and stop the algorithm when a certain threshold ϵ is reached, or alternatively when a predefined number of steps is reached.

Here we illustrate directly the EM algorithm applied to GMM, but it's important to underline that this algorithm can be used also in other contexts.

EM for GMM Algorithm:

1. *Initialize the parameters θ (μ_k, Σ_k and π_k) for each of the K gaussians to random values.*

***Note**: This is not so trivial as it may seem. We must be careful in this very first step, since the EM algorithm will likely converge to a local optimum. I suggest you initialize them such that they are not too far from the **data manifold**, in this way we can minimize the risk they get stuck over outliers.*

2. Repeat {

1. ***(E) Expectation step**: we fix the parameters θ^{t-1} and for each data point $x^{(i)}$ and each cluster C_k (cluster-point pair), we compute a **responsibility coefficient** r_{ik} that measures a relative probability that the data point $x^{(i)}$ belongs to cluster C_k. This coefficient gives an idea of how much the point is generated from the gaussian that is paired with it at the moment, w.r.t. the entire mixture:*

$$r_{ik} := p\left(z = k|x^{(i)}, \theta^{t-1}\right)$$

where θ^{t-1} are the parameters set in the previous iteration. We can make this formula more explicit as:

$$r_{ik} = \frac{\pi_k \, p_k\left(x^{(i)}|\theta_k^{t-1}\right)}{\sum_{l=1}^{K} \pi_l \, p_l\left(x^{(i)}|\theta_l^{t-1}\right)}$$

***Interpretation**: This equation calculates r_{ik} as the relative likelihood that the data point $x^{(i)}$ was generated by the k-th Gaussian component. It is the contribution of the k-th component to the overall probability of $x^{(i)}$, normalized by the total contribution of all K components.*

***Note**: r_{ik} is a numeric parameter ranging from **0 to 1**: If a Guaussian is not a very good explanation for $x^{(i)}$ it will typically have a small r_{ik}, conversely if it's the best explanation for $x^{(i)}$ it will have a big r_{ik}, close to 1.*

2. ***(M) Maximization step**: we fix the responsibilities r_{ik} found in the E-step, and we recompute the parameters as a function of the responsibility coefficient to maximize the likelihood:*

$$\pi_k = \frac{1}{m}\sum_{i=1}^{m} r_{ik}$$

$$\mu_k = \frac{\sum_i r_{ik} x^{(i)}}{\sum_i r_{ik}}$$

$$\Sigma_k = \frac{\sum_i r_{ik}\left(\left(x^{(i)} - \mu_k\right)\left(x^{(i)} - \mu_k\right)^T\right)}{\sum_i r_{ik}}$$

Here we can say that this actually help us to maximize the likelihood, since we know that the parameters are for sure better or at least equal in the steps. This is because an interesting property of EM is that during the iterative maximization procedure, the value of the likelihood will continue to increase after each iteration. In other words, the EM algorithm never makes things worse.

} until convergence

So, once we have got the parameters μ_k, Σ_k *and* π_k for each of the K components of our mixture via EM, we can then easily apply to a data point:

$$p\left(x^{(i)}|\theta\right) = \sum_{k=1}^{K} \pi_k \, p_k\left(x^{(i)}|\theta_k\right)$$

to get its probability to appertain to the mixture.

Gaussian Mixture Models – PROS/CONS

- Pros
  - It is a flexible algorithm that can approximate multivariate gaussian to fit data maximizing the likelihood.
  - Less sensible to outliers.
- Cons
  - The algorithm can diverge if there are only a few points w.r.t. the overall mixture.
  - All the flexibility of the model is always used, even if it is not needed.

+ Importance of Gaussian Mixture Models Beyond Clustering:

A remarkable property of GMMs is that by using a sufficient number of Gaussian components and carefully adjusting their means, covariances, and mixing coefficients, GMMs can approximate almost any continuous density distribution with great accuracy.

This flexibility makes GMMs a versatile modeling technique valuable in a wide range of applications:

- **Anomaly Detection**: GMMs are often used to identify anomalies in datasets by modeling the "normal" behavior of the data. Points with low likelihood under the fitted GMM are flagged as anomalies. We will see this at the end of the chapter.
- **Generative Models**: GMMs serve as probabilistic generative models. For instance, they are used in **Variational Autoencoders (VAEs)** to approximate complex posterior distributions, enabling the generation of new data samples that resemble the original dataset. We will see this in deep learning (I hope).
- **Speech Processing**: GMMs play a key role in systems like **Hidden Markov Models (HMMs)** for speech recognition, where they model the distributions of acoustic features.
- **Data Imputation**: GMMs can also be employed to handle missing data by generating samples from the estimated distribution.

11.4 Choosing the Value of K

As mentioned, $\boldsymbol{K}$ **is a hyperparameter** representing the number of clusters into which we want to divide our data. However, finding the right value (optimal) for K is often not so straightforward, so in this case we wonder how to determine it.

Several methods can guide this selection, described below. These include **Elbow Method**, **Silhouette Method**, and **Information Criteria (IC)**.

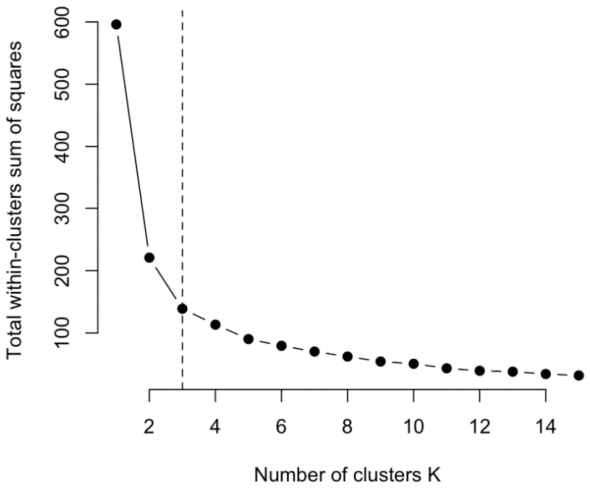
Elbow Method

The Elbow method is the simplest (and most popular) method to choose the value of K. It is a graphical inspection method in which the optimal number of cluster K is determined, based on the total **sum of squares** within the clusters (which measures the variance explained by the clustering model). The ideal number of clusters is selected by choosing the **elbow** of the curve, (hence the name of the method) the point where adding more clusters no longer significantly improves the model's performance.

We see this with K-means, but this is applicable with any other clustering technique.

In practice, the k-means is iterated for different values of K and each time the sum of the squared distances between each centroid and the points of its cluster is calculated.

By plotting the values of $\boldsymbol{K}$ (horizontal axis) and the values of the **sum of the squared distances** (vertical axis), a graph similar to the one in the Figure is obtained.



In Figure it can be seen that from the value $K = 1$ to $K = 3$ the curve rises more clearly, while from the value $K = 3$ onwards the curve flattens significantly. This means that the value $K = 3$ is our elbow.

→ The optimal number of clusters k is the one in which the elbow is positioned. In this case $K = 3$.

Interpretation: The **total sum of squares** is something that reflects the overall **variance** in the data: As the number of clusters K increases, more centroids are added, reducing the distances between data points and their closest centroid, which in turn lowers the variance within clusters. This improves how well the model fits the data, but beyond a certain point, adding more clusters leads to diminishing returns and the risk of **overfitting**.

In fact, the first clusters will add much information (**explain a lot of variation**), since the data actually consist of that many groups (so these clusters are necessary), but once the number of clusters exceeds the actual number of groups in the data, the added information contributes minimally to the explained variation, merely subdividing existing groups. This transition point, where the rate of improvement drops sharply, is exactly what the "**elbow**" represents (point where adding another cluster doesn't give much better modeling of the data) and so makes sense to choose it as k value.

This means that the Elbow method plots a graph of **explained variation versus clusters**: increasing rapidly up to optimal k (under-fitting region) and then increasing slowly after K (over-fitting region).

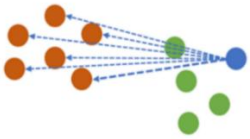
Silhouette Analysis

Silhouette analysis gives a measure of how close each point in one cluster is to points in the neighboring clusters helping to assess the suitability of the clustering structure. Specifically, the **Silhouette Coefficient** is a measure of how similar a data point is within-cluster (**cohesion**) compared to other clusters (**separation**).

For each individual data point $x^{(i)}$ we compute:

- $\boldsymbol{coh} = average\ distance\ of\ x^{(i)}\ to\ the\ samples\ in\ the\ same\ cluster$

- $\boldsymbol{sep} = \min\left(average\ distance\ of\ x^{(i)}\ to\ samples\ in\ another\ cluster\right)$



- we calculate the Silhoutte coefficient s_i through the equation:

$$s_i = 1 - \frac{coh}{sep} \quad if\ coh < sep$$

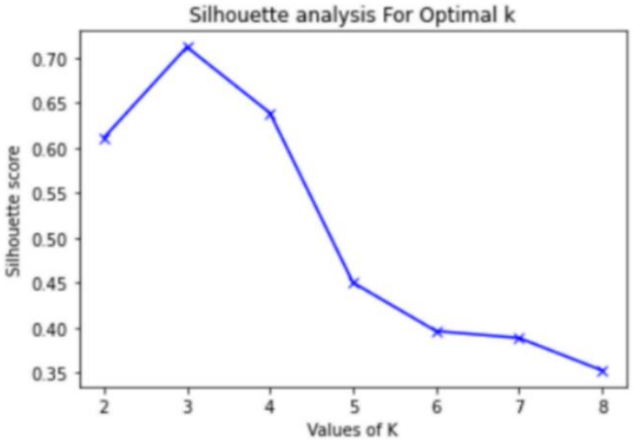
The value of the silhouette coefficient s_i is between $[-1, 1]$:

- A score near to 1 denotes that the data point $x^{(i)}$ is very compact within the cluster to which it belongs and far away from the neighboring clusters.
- A value of 0 indicates that the data point is on or very close to the decision boundary between two neighboring clusters
- A negative value indicates that the data point might have been assigned to the wrong cluster.
  → The closer to 1 the better.

To evaluate the overall clustering quality, the **Average Silhouette Score** (also simply known as Silhouette Score) is computed as the mean of all individual s_i values:

$$AverageSilhouette = \frac{1}{m}\sum_{i=1}^{m} s_i$$

By calculating the average silhouette score for different values number of clusters K, we can plot a graph of silhouette scores versus number of clusters. The optimal number of clusters corresponds to the K value that maximizes the average silhouette score.



E.g. we can see in the plot above the average silhouette score is maximized at $K = 3$. So, we will take 3 clusters.

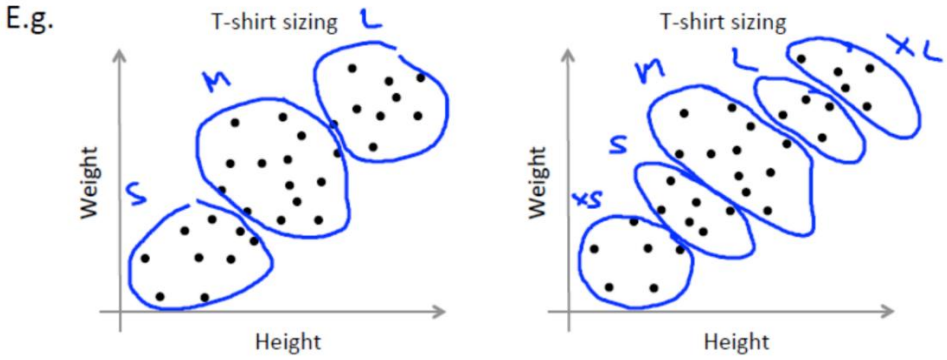
Note: Generally, the silhouette Method is used in combination with the Elbow Method for a more confident decision.

Choosing the Value of K: a bit more …

While the primary goal of clustering is often to find natural groupings in the data, there are situations where our ML system must fit some external constraints. In such cases, the number of clusters k is determined not solely by the structure of the data but by how well the clusters serve this external purpose or task.

Example: T-Shirt Sizing

Consider a scenario where you are tasked with designing T-shirt sizes. Suppose the requirement is to offer three sizes: Small (S), Medium (M), and Large (L). In this case, you would create exactly three clusters, regardless of whether the data naturally supports this division. Similarly, if the requirement changes to five sizes—Extra Small (XS), Small (S), Medium (M), Large (L), and Extra Large (XL)—you would form five clusters. Here, clustering is driven by the need to meet external market demands rather than purely optimizing the representation of the data. This is a common situation in market segmentation, where the goal is to create products or services that cater to predefined subpopulations.



When K is chosen based on external constraints, it becomes important to evaluate how well the resulting clusters perform in their intended task. We need better metrics that help us to answer: "How do we evaluate of good our clusters are?"

That's why to quantify the quality of clusters, additional metrics are often employed. One such family of metrics involves **Information Criteria (IC)**, which uses statistical measures all based or related to the concept of **KL Divergence**. Here, we will first introduce the concept of KL Divergence and then we will explore some of these IC.

Kullback–Leibler Divergence

In situations like this, it is often useful to quantify the difference between two probability distributions. This is typically framed as the problem of calculating the **statistical distance** between two distributions. While one approach is to measure the direct distance between them, interpreting such a distance can be challenging.

A more common and effective approach is to compute a **divergence** between the distributions, which provides a clearer measure of how one distribution differs from another. A widely used divergence measure from information theory is the Kullback-Leibler Divergence.

Kullback-Leibler (KL) Divergence measures how much a given probability distribution diverges from a reference or expected distribution (information loss). In other words, it quantifies how much one probability distribution

differs from another probability distribution, indicating how much information is lost when one distribution is used to approximate another.

The KL divergence between two distributions P (true distribution) and Q (approximating distribution) is often stated using the following notation $D_{KL}(P||Q)$ where the "||" operator indicates "divergence" of P from Q.

KL divergence can be calculated as:

$$D_{KL}(P||Q) = -\sum_i P(i) \log \frac{Q(i)}{P(i)} = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

Note: For continuous probability distributions, the summation is replaced by an integral.

KL Divergence values range from 0 to ∞, with 0 representing identical distributions. A score closer to 0 suggests the distributions behave similarly, whereas higher values indicate more substantial divergence.

Information Criterion

At this point we might ask: "What is the probability that our data are explained by our model?" and "what is the simplest possible model that explains our data?"

The first question concerns the likelihood function: How likely is it that our data x are explained by the model M (e.g. a certain clustering), which is parameterized by some parameters θ?". Once we identify the parameter values that maximize this probability, we denote them with a caret (^), resulting in the **maximum likelihood**:

$$\hat{L} = L(\hat{\theta}) = \max_{\theta} L(\theta; X; y; M) = p(y|X; \hat{\theta}, M)$$

Rember that: the likelihood function $L(\theta; X; y; M)$ represents the probability of observing the output data y (outcomes of the target variable that we are trying to model) given the input data X, the parameters θ and model M, and $\hat{L} = L(\hat{\theta})$ represents the maximum likelihood estimate of these parameters. The **maximum likelihood estimate** (MLE) is the value of θ\thetaθ that maximizes this likelihood function.

The challenge, however, is to balance the maximized likelihood with model complexity (which answers the second question). In clustering, this means we want to select the clustering (model) that fits with the fewest number of cluster k that still does a good job explaining the data.

A statistician, Hirotugu Akaike, took into consideration the relationship between KL Divergence and Maximum Likelihood and developed the concept of **Information Criterion (IC)** approach.

Several **Information Criteria** can be used to estimate how well a model fits the data while penalizing unnecessary complexity. In our case they will be used to selecting the best clustering, so the one that best fits data while having the minimum number of clusters K.

These criteria are equivalent or proportional to each other, although each was derived from a different framing or field of study:

- AIC: Akaike information criterion
- BIC: Bayesian information criterion
- DIC: Deviance information criterion
- BF: Bayes factors

- MDL: Minimum Description Length
- FIA: Fisher Information Approximation

We will briefly see just the first three.

Akaike Information Criterion (AIC)

When a statistical model is used to represent the process that generated the data, the representation will almost never be exact; so, some information will be lost by using the model to represent the process. The **Akaike Information Criterion (AIC)** provides a way to estimate the relative amount of information lost by a given model: the less information a model loses, the higher the quality of that model. Moreover, in estimating this amount of information lost by a model, AIC deals with the trade-off between the goodness of fit of the model and the simplicity of the model (as we will see soon).

The AIC is derived as an asymptotic approximation (meaning it is most reliable when the sample size is large) of the **Kullback-Leibler (KL) information distance** between the probability distribution described by the candidate model (the model used to estimate the true underline distribution) and the true data-generating process.

The AIC is expressed as:

$$AIC = 2K - 2\ln(\hat{L})$$

where K is the number of parameters (in clustering context they can be the number of clusters) and $\hat{L}$ is the maximum-likelihood of the model.

Given a set of candidate models for the data (e.g. models with varying k), then the model with the smallest AIC is preferred → The model that minimizes AIC is selected.

Interpretation: AIC rewards goodness of fit, as measured by the likelihood function $\hat{L}$, but it also includes a penalty that is an increasing function of the number of parameters ($2K$). The penalty discourages overfitting, which is desired because increasing the number of parameters typically improves the model's fit to the data but risks reducing its ability to generalize to new observations.

→ As the number of parameters K increases, both the likelihood $\hat{L}$ and the penalty term $2K$ increase. Consequently, the model with the smallest AIC will be the one with the best trade-off between maximizing fitness and minimizing complexity.

It is better to use an adjusted version of AIC known as **AICc** (**corrected AIC**), in which a correction related to the number of samples is injected:

$$AICc = AIC + \frac{2K(K+1)}{m-K-1}$$

AICc adds an additional penalty term specifically designed for small sample sizes. Thanks to this, AICc is more reliable when the sample size is small relative to the number of model parameters.

Bayesian information criterion (BIC)

The **Bayesian Information Criterion (BIC)** is a statistical criterion for model selection, similar to the Akaike Information Criterion (AIC). It evaluates models based on their fitness to the data and penalizes models with higher

complexity (i.e., more parameters). Like AIC, BIC estimates model parameters using the maximum likelihood method, but it incorporates a stronger penalty for complexity, especially as the dataset size increases.

The BIC is expressed as:

$$BIC = \ln(m)K - 2\ln(\hat{L})$$

where K is the number of parameters, $\hat{L}$ is the maximum-likelihood of the model and m is the number of data points in the training set.

→ The model that minimizes BIC is selected.

Interpretation: In BIC, the penalty term for model complexity is $\ln(m)K$, which increases with both the number of parameters (K) and the dataset size (m). This penalty grows faster than AIC's penalty ($2K$) for datasets where $m > 7$ (since $\ln(7) \approx 1.946$), making BIC more conservative in selecting complex models (so it prefers simpler ones).

Deviance Information Criterion (DIC)

The **Deviance Information Criterion (DIC)** is a model selection criterion similar to AIC and BIC, but it is specifically used in Bayesian model selection. Like AIC and BIC, DIC balances the trade-off between model fit and complexity, but instead of using directly the log-likelihood, DIC focuses on deviance to assess model fit. The **deviance** measures the **lack of fitness** of the model, with lower values indicating a better fit. More precisely, the deviance is defined as:

$$D(\theta) = -2\log L(\hat{\theta}) = -2\log\left(p(y|X;\hat{\theta},M)\right)$$

DIC is designed primarily for hierarchical Bayesian models, where parameters are estimated from their posterior distributions. Importantly, DIC is only valid when the posterior distribution of the parameters is approximately multivariate normal.

DIC is computed as:

$$DIC = \overline{D(\theta)} + p_D$$

Here:

- $\overline{D(\theta)} = \mathbb{E}[D(\theta)]$ is the **expected deviance** over the posterior distribution of the model parameters.
- p_D is the **effective number of parameters,** which quantifies model complexity and penalizes models with higher complexity. It is calculated as:

  $$p_D = \overline{D(\theta)} - D(\bar{\theta}) = \mathbb{E}_{\theta|y}\left[-2\log\left(p(y|X;\theta,M)\right) + 2\log\left(p(y|X;\bar{\theta},M)\right)\right]$$

  where $\bar{\theta} = \mathbb{E}[\theta|y]$ is the **posterior mean** of the parameters.

→ The model that minimizes DIC is selected.

Information criterions comparison:

- **AIC lacks the finite samples correction in its base form**: The base formula for AIC does not incorporate a correction for small sample sizes. As a result, when working with smaller datasets, AIC can overestimate the goodness of fitness and may lead to the selection of overly complex models. However, this limitation is addressed by using its adjusted version, the **AICc**.

- **AIC is asymptotically equivalent to k-folds cross-validation on linear regression models with a least squares optimization**: This means that as the sample size grows large, the AIC tends to provide the same model selection result as cross-validation done on linear regression models with a least squares optimization.
- **BIC is able to detect the true model (if present in the comparison)**: The BIC tends to be more conservative as it incorporates a stronger penalty for model complexity, which makes it more likely to select effectively select the true model (of course if it's part of the model candidate set).
- **BIC is only valid if m is much greater than K**: BIC is derived under the assumption that the sample size m is much larger than the number of parameters K in the model, so when m is not significantly greater than k, BIC may not perform well (e.g. we saw that m should at least 7, but higher is better).
- **DIC tends to suggest overfitted models**: DIC might favor overly complex models because it does not penalize complexity as strongly as other criteria, such as BIC, leading to a tendency to select models that are too complex relative to the data.

+ AIC/BIC for K-Means (extra)

We recall that the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC) are defined as:

$$AIC = 2K - 2\ln(\hat{L}) \qquad\qquad BIC = \ln(m)K - 2\ln(\hat{L})$$

where K is the number of parameters and $\hat{L}$ is the maximum likelihood of the model.

Since these criteria require a likelihood function, it is necessary to derive the likelihood for K-means clustering to apply them. Let's therefore see how we can derive it.

To derive the likelihood:

- We first demonstrate that K-means can be understood as a special case of the Expectation-Maximization (EM) algorithm applied to GMMs under specific assumption that we will see soon.
- Once this connection is established, we can delve into the computation of the likelihood for the K-means model and at the end use it to evaluate AIC and BIC.

Note: The derivation that I provided here simplifies the key concepts to enhance understanding. I wrote this derivation taking inspiration by:

- For the first part from portions of section 9.3 of the book *Pattern Recognition and Machine Learning* and from the following Stack Exchange discussion: K-means vs GMM
- For the second part from the appendix of this paper and from the following Stack Exchange discussion: AIC for K-means

For a deep understanding of the argument, I recommend you read Section 9.3 of *Pattern Recognition and Machine Learning* in its entirety (in my derivation a lot of stuff here present will be intentionally missed) and also look for this link which could help in mathematical steps derivation.

K-Means as a Special Case of the EM Algorithm applied to GMMs

K-means can be viewed as a limiting case of Expectation-Maximization (EM) applied to spherical Gaussian Mixture Models (GMMs) as the variances of the covariance matrices approach zero. Let's demonstrate why it is the case.

Rember that we saw that K-means can be seen as an application of EM algorithm.

K-means start by initializing values for the μ_k randomly and then iteratively alternate between the following two steps until convergence is reached:

- **E-step**: keeping the μ_k fixed, it chooses optimal r_{ik} by assigning $x^{(i)}$ to the nearest centroid μ_k:

$$\boldsymbol{r_{ik}} \rightarrow \begin{cases} \mathbf{1}, & \boldsymbol{if\ k = \underset{j}{\operatorname{argmin}} \left\| x^{(i)} - \mu_j \right\|^2} \\ \mathbf{0}, & \boldsymbol{otherwise} \end{cases}$$

- **M-step**: keeping r_{ik} fixed, it chooses optimal centroids μs by updating each μ_k to be the mean of the points assigned to each cluster:

$$\boldsymbol{\mu_k = \frac{\sum_{i=1}^{m} r_{ik} x^{(i)}}{\sum_{i=1}^{m} r_{ik}}}$$

  where $\sum_{i=1}^{m} \boldsymbol{r_{ik}} = |\boldsymbol{C_k}|$ (number of data points in the k-th cluster C_k).

Next, consider the standard EM steps for Gaussian Mixture Models (GMMs). After picking initial mean vectors μ_k, covariances Σ_k, and mixing coefficients π_k, EM alternates between the following two steps:

- **E-step**: Keeping fixed the parameters (all the μ_k, Σ_k, π_k) for each data point $x^{(i)}$, evaluate the "responsibility" r_{ik} of each cluster of index k for that data point as:

$$r_{ik} = \frac{\pi_k \, p_k\left(x^{(i)} | \theta_k\right)}{\sum_{l=1}^{K} \pi_l \, p_l(x^{(i)} | \theta_l)} = \frac{\pi_k \, \mathcal{N}\left(x^{(i)} | \mu_k, \Sigma_k\right)}{\sum_{l=1}^{K} \pi_l \, \mathcal{N}(x^{(i)} | \mu_l, \Sigma_l)}$$

- **M-step**: Keeping fixed r_{ik}, for each cluster of index k, re-estimate the parameters μ_k, Σ_k, π_k as:

$$\pi_k^{new} = \frac{1}{m} \sum_{i=1}^{m} r_{ik}$$

$$\mu_k^{new} = \frac{\sum_i r_{ik} x^{(i)}}{\sum_i r_{ik}}$$

$$\Sigma_k^{new} = \frac{\sum_i r_{ik} \left(\left(x^{(i)} - \mu_k^{new}\right) \left(x^{(i)} - \mu_k^{new}\right)^T \right)}{\sum_i r_{ik}}$$

If we compare these update equations to the update equations for K-means that we illustrated above, we see that, in both, r_{ik} serves as a probability distribution over clusters for each data point. The primary difference is that in K-means, the r_{ik} is a probability distribution that gives zero probability to all but one cluster, while EM for GMMs gives non-zero probability to every cluster.

Therefore, the key difference between the two lies in the responsibility matrix r_{ik}:

- In EM for K-means, r_{ik} is binary (**hard assignment**): each data point is assigned to a single cluster.
- In EM for GMMs, r_{ik} is continuous (**soft assignment**): each data point is assigned to all clusters with varying probabilities.

Now, consider a GMM in which the covariance matrices of the mixture components are **spherical**, so they are given by $\boldsymbol{\Sigma_j = \epsilon I}$, where ϵ is a variance parameter that is shared by all of the components, and I is the identity matrix. The generic probability density function (pdf) for a data point $x^{(i)}$ belonging to cluster of index j:

$$p\left(x^{(i)}; \mu_k, \Sigma_k\right) = \mathcal{N}\left(x^{(i)}|\mu_k, \Sigma_k\right) = \frac{1}{\sqrt{(2\pi)^n|\Sigma|}} e^{\left(-\frac{1}{2}(x-\mu_k)^T\Sigma^{-1}(x-\mu_k)\right)}$$

under this assumption simplifies to:

$$N\left(x^{(i)}|\mu_k, \epsilon I\right) = \frac{1}{\sqrt{(2\pi\epsilon)^n}} e^{-\frac{\left\|x^{(i)}-\mu_k\right\|^2}{2\epsilon}} = \frac{1}{(2\pi\epsilon)^{n/2}} e^{-\frac{\left\|x^{(i)}-\mu_k\right\|^2}{2\epsilon}}$$

Let's consider an EM algorithm for a mixture of K Gaussians of this form in which we treat ϵ as a fixed constant, instead of a parameter to be re-estimated. Using the simplified pdf $N\left(x^{(i)}|\mu_k, \epsilon I\right)$, the **E-step** now computes the responsibilities as:

$$r_{ik} = \frac{\pi_k \exp\left(-\frac{1}{2\epsilon}\left\|x^{(i)} - \mu_k\right\|^2\right)}{\sum_{l=1}^{K} \pi_l \exp\left(-\frac{1}{2\epsilon}\|x^{(i)} - \mu_l\|^2\right)}$$

If we consider **the limit $\boldsymbol{\epsilon \to 0}$**, we see that in the denominator the term l for which $\left\|x^{(i)} - \mu_l\right\|^2$ is smallest will go to zero most slowly, and hence the responsibilities r_{ik} for the data point $x^{(i)}$ all go to zero except for term k, for which the responsibility r_{ik} will go to unity. So, we got that $r_{ik} \to 0$ for all $l \neq k$ and $r_{ik} \to 1$ for $l = k$. Note that this holds independently of the values of the π_k as long none of the π_k is zero. Thus, in this limit, we obtain a **hard assignment** of data points to clusters, just as in the E-step of the *K*-means algorithm:

$$r_{ik} \to \begin{cases} 1, & if\ k = \underset{l}{\operatorname{argmin}}\left\|x^{(i)} - \mu_l\right\|^2 \\ 0, & otherwise \end{cases}$$

Moreover, since we are treating $\Sigma_k = \epsilon I$ as fixed, in the M-step there's no need to re-estimate the covariance Σ_k (it's simply ϵI). While the update for the mean μ_k is identical to that in K-means.

This demonstrates that K-means is a limiting case of EM for GMMs with spherical covariance matrices and vanishing variance.

Computing Likelihood of K-Means

We saw above that K-means is a limiting case of EM for GMMs under the assumption of spherical and equal covariance matrices ϵI where $\epsilon \to 0$. Moreover, we got that the pdf for a data point $x^{(i)}$ belonging to cluster k under this assumption is given by:

$$N\left(x^{(i)}|\mu_k, \epsilon I\right) = \frac{1}{(2\pi\epsilon)^{n/2}} e^{-\frac{\left\|x^{(i)}-\mu_k\right\|^2}{2\epsilon}}$$

We also established that r_{ik} results in a hard assignment of data points to clusters, independent of the values of the mixture weights π_k (provided none of them are zero). Therefore, to simplify further, we can assume that the mixture weights are uniform across all clusters, i.e., $\pi_k = \frac{1}{K}$ for all $k = 1, \ldots, K$.

With these assumptions, we can now compute the likelihood for k-means.

The likelihood for the entire dataset can be expressed as:

$$\hat{L} = \prod_{k=1}^{K} \prod_{i:x^{(i)} \in C_k} \pi_k \, N\left(x^{(i)} | \mu_k, \epsilon I\right)$$

Moreover, since we are more interested in computing $\ln(\hat{L})$ to apply AIC or BIC, we could take the natural logarithm:

$$\ln(\hat{L}) = \sum_{k=1}^{K} \sum_{i:x^{(i)} \in C_k} \ln\left(\pi_k \, N\left(x^{(i)} | \mu_k, \epsilon I\right)\right) =$$

$$= \sum_{k=1}^{K} \sum_{i:x^{(i)} \in C_k} \ln(\pi_k) + \ln\left(N\left(x^{(i)} | \mu_k, \epsilon I\right)\right) =$$

$$= \sum_{k=1}^{K} \sum_{i:x^{(i)} \in C_k} \ln\left(\frac{1}{K}\right) + \ln\left(\frac{1}{(2\pi\epsilon)^{n/2}} e^{-\frac{\left\|x^{(i)} - \mu_k\right\|^2}{2\epsilon}}\right) =$$

$$= \sum_{k=1}^{K} \sum_{i:x^{(i)} \in C_k} \ln\left(\frac{1}{K}\right) + \ln\left(\frac{1}{(2\pi\epsilon)^{\frac{n}{2}}}\right) - \frac{\left\|x^{(i)} - \mu_k\right\|^2}{2\epsilon}$$

Let's analyze the term $\sum_{k=1}^{K} \sum_{i:x^{(i)} \in C_k} \ln\left(\frac{1}{(2\pi\epsilon)^{\frac{n}{2}}}\right)$, this can be rewritten as:

$$\sum_{k=1}^{K} |C_k| \ln\left(\frac{1}{(2\pi\epsilon)^{\frac{n}{2}}}\right)$$

This is because the inner product $\ln\left(\frac{1}{(2\pi\epsilon)^{\frac{n}{2}}}\right)$ is constant for all points in a given cluster C_k and so, the summation over all data points in C_k effectively counts the number of points in C_k, which we denote by $|C_k|$.

Now, since the total number of data points is the sum of the points in all clusters:

$$\sum_{k=1}^{K} |C_k| = m$$

we have that:

$$\sum_{k=1}^{K} |C_k| \ln\left(\frac{1}{(2\pi\epsilon)^{\frac{n}{2}}}\right) = m \ln\left(\frac{1}{(2\pi\epsilon)^{\frac{n}{2}}}\right)$$

and expanding the logarithm we have:

$$\ln\left(\frac{1}{(2\pi\epsilon)^{\frac{n}{2}}}\right) = -\ln\left((2\pi\epsilon)^{\frac{n}{2}}\right) = -\frac{n}{2}\ln(2\pi\epsilon)$$

Therefore, the first term we have seen above is equivalent to:

$$\sum_{k=1}^{K} \sum_{i:x^{(i)} \in C_k} \ln\left(\frac{1}{(2\pi\epsilon)^{\frac{n}{2}}}\right) = -m\frac{n}{2}\ln(2\pi\epsilon)$$

The same holds for $\sum_{k=1}^{K} \sum_{i:x^{(i)} \in C_k} \ln\left(\frac{1}{K}\right)$ which becomes $m \ln\left(\frac{1}{K}\right)$.

Thus, the log-likelihood becomes:

$$\ln(\hat{L}) = m\ln\left(\frac{1}{K}\right) - m\frac{n}{2}\ln(2\pi\epsilon) - \sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\frac{\left\|x^{(i)} - \mu_k\right\|^2}{2\epsilon}$$

which we can further expand as:

$$\ln(\hat{L}) = m\ln\left(\frac{1}{K}\right) - \frac{mn}{2}\ln(2\pi\epsilon) - \frac{1}{2\epsilon}\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2 =$$

$$= m\ln\left(\frac{1}{K}\right) - \frac{mn}{2}[\ln(2\pi) + \log(\epsilon)] - \frac{1}{2\epsilon}\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2$$

At this point let's stop for a moment. From our earlier discussion, we demonstrated that:

- The centroid μ_k of each cluster of index k is updated same as in standard k-means, so it is computed as the mean of the points assigned to that cluster.
- ϵ, the shared variance parameter across all clusters, is fixed and not re-estimated. Since ϵ must be defined, a reasonable assumption is to set it as the average squared distance of points to their respective cluster centroids, i.e., the within-cluster variance:

$$\epsilon = \frac{\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2}{mn}$$

Substituting this definition of ϵ into the log-likelihood, we obtain:

$$\ln(\hat{L}) = m\ln\left(\frac{1}{K}\right) - \frac{mn}{2}[\ln(2\pi) + \ln(\epsilon)] - \frac{1}{2\epsilon}\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2 =$$

$$= m\ln\left(\frac{1}{K}\right) - \frac{mn}{2}\ln(2\pi) - \frac{mn}{2}\ln(\epsilon) - \frac{1}{2\epsilon}\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2 =$$

$$= m\ln\left(\frac{1}{K}\right) - \frac{mn}{2}\ln(2\pi) - \frac{mn}{2}\ln\left(\frac{\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2}{mn}\right)$$
$$- \frac{1}{2\frac{\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2}{mn}}\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2 =$$

$$= m\ln\left(\frac{1}{K}\right) - \frac{mn}{2}\ln(2\pi) - \frac{mn}{2}\ln\left(\frac{\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2}{mn}\right) - \frac{mn}{2}\frac{\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2}{\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2} =$$

$$= m\ln\left(\frac{1}{K}\right) - \frac{mn}{2}\ln(2\pi) - \frac{mn}{2}\ln\left(\frac{\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2}{mn}\right) - \frac{mn}{2} =$$

Now by simply collecting all the constant terms in C, we have that:

$$\ln(\hat{L}) = -\frac{mn}{2}\ln\left(\frac{\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)} - \mu_k\right\|^2}{mn}\right) + C$$

Since constant terms do not affect relative comparisons, we can write:

$$\ln(\hat{L}) \propto -\frac{mn}{2}\ln\left(\frac{\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)}-\mu_k\right\|^2}{mn}\right)$$

Compute AIC and BIC for K-Means

Using this simplified log-likelihood, we are now able to compute the AIC and BIC for k-means clustering:

$$AIC = 2K + mn\ln\left(\frac{\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)}-\mu_k\right\|^2}{mn}\right)$$

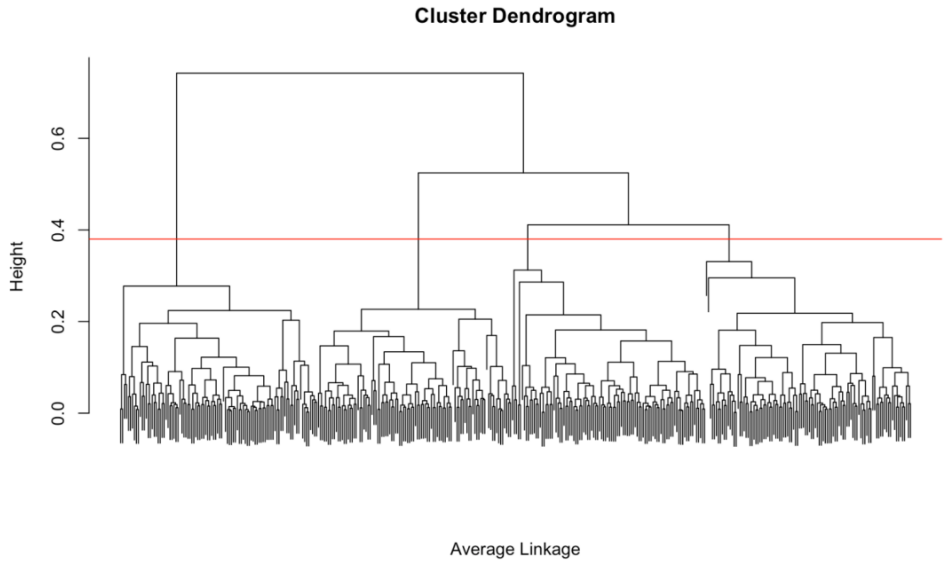
$$BIC = \ln(m)\,K + mn\ln\left(\frac{\sum_{k=1}^{K}\sum_{i:x^{(i)}\in C_k}\left\|x^{(i)}-\mu_k\right\|^2}{mn}\right)$$

11.5 Hierarchical Clustering

Hierarchical Clustering involves building a hierarchy of (potentially overlapping) clusters. This hierarchy can be of two types:

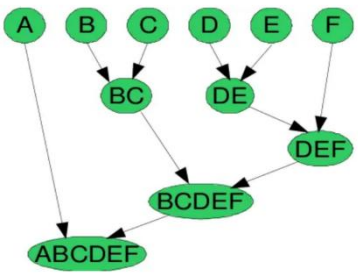
- Agglomerative ("bottom-up")
- Divisive ("top-down")

Once the hierarchy is built, the final number of clusters can be chosen by "**cutting**" the hierarchy at a specific height.



Note: Hierarchical clustering can be visualized as a **Dendrogram**, a tree-like diagram that records the sequences of merges or splits (Figure above).

Types of Hierarchies



- **Agglomerative ("bottom-up")**: In the agglomerative approach, each data point starts as its own individual cluster. Then, at each iteration, the two most similar clusters are merged together. This process continues until all data points are in one cluster.

  Algorithm pseudo-code:

  1. Let each data point be a cluster
  2. Repeat {

     Merge the two most similar clusters (based on a similarity metric)

     } until only a single cluster remains

- **Divisive ("top-down")**: The divisive approach works in reverse. Initially, all data points are grouped into a single cluster. At each iteration, the data points that are least similar to the rest of the cluster are separated and treated as individual clusters. This process continues until each data point is its own cluster. This approach is not much used in practice.

How do we calculate the similarity between clusters?

Calculating the similarity between clusters is essential to merge or divide the clusters.

Deciding how to merge or split clusters depends on two key factors:

1. **Similarity Measure/Metric**: Common similarity measures include Euclidean distance, Manhattan distance, cosine similarity, and others, depending on the nature of the data and the desired outcome.
2. **Linkage Criterion**: This determines how the similarity between clusters is quantified. It could be:
   - **Distance-Based**: This involves calculating the distance between clusters using methods like:
     - **Single Linkage**: The **minimum** distance between any two points in different clusters.
     - **Complete Linkage**: The **maximum** distance between points in different clusters.
     - **Average Linkage**: The **average** distance between all pairs of points in the two clusters.
   - **Probability-Based**: This measures the probability that two candidate clusters come from the same distribution.
   - **Graph-Based**: The similarity can be based on the product of the **in-and-out-degree** of a node in a graph (graph degree linkage).
   - **Instance-Based Constraints**: This involves constraints based on prior knowledge or external labels (e.g., see Wagstaff, 2002).

Single Linkage

The **Single Linkage** method is one of the most well-known hierarchical clustering algorithms.

The idea:

1. Start with every single point in its own cluster (m clusters)
2. Merge the two *closest points* to form a new cluster
3. Repeat: each time merging the two closest pairs of points

The configuration:

- The similarity metric is Euclidean distance
- The linkage criterion is the [pairwise] minimum distance

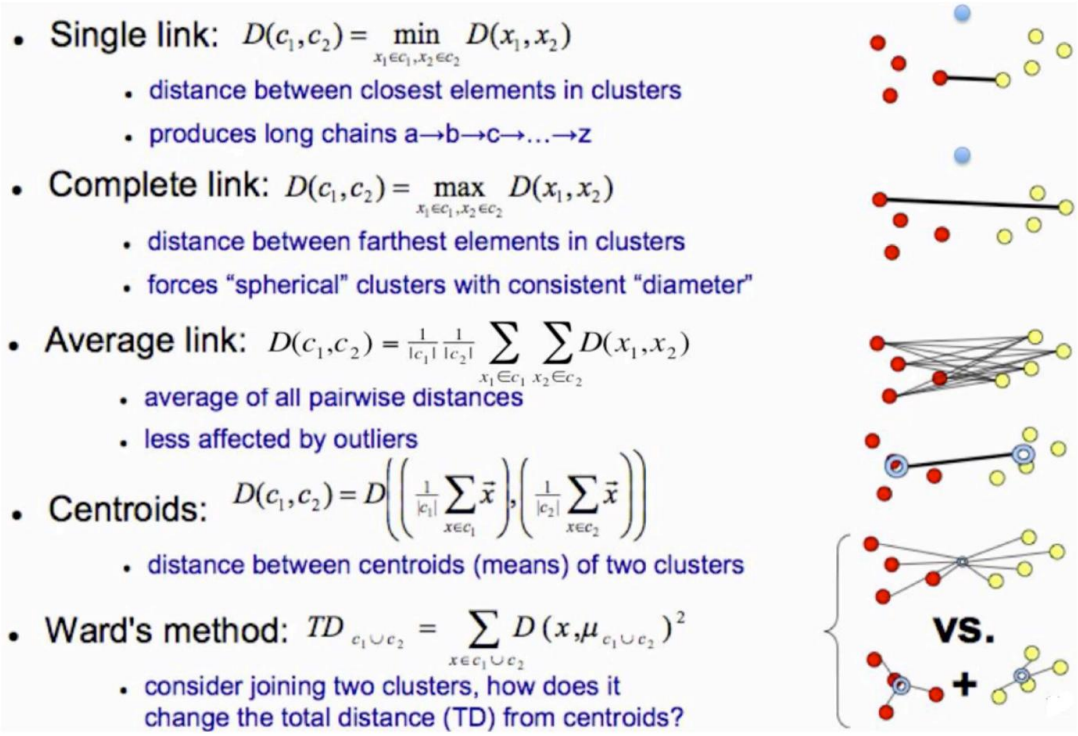
Ward's Criterion

Ward's criterion takes a different approach to clustering: rather than using a distance metric as measures of similarity it merges clusters based on a statistical measure that focuses on minimizing the total squared error.

This method assumes that the data points come from approximately elliptically-shaped distributions, and it seeks to minimize the variance within each cluster at each merge.

→ **"Ward's criterion proposes agglomeration where observations are merged in such a way that minimize the total squared error at each merge, which can be interpreted as maximizing the explained variance (R^2) of the given clustering"**

Cluster Distance Measures:



11.6 DBSCAN

DBSCAN, which stands for "**D**ensity-**B**ased **S**patial **C**lustering of **A**pplications with **N**oise," is a **density-based** algorithm. This means that it works by grouping data points based on regions of high density, separating these regions from areas of low density.

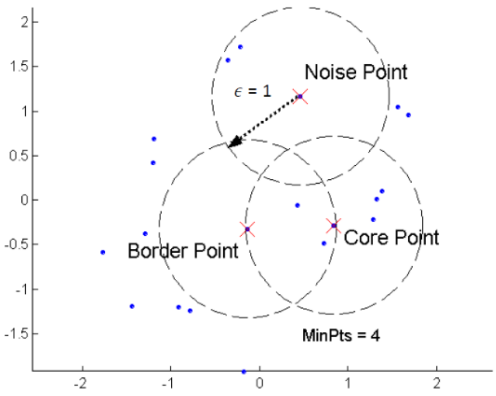
Note: It is NOT a Hierarchical Clustering algorithm.

In DBSCAN, **density** is defined as the number of data points within a specified radius, called ϵ (Eps).

Definition: The ϵ-Neighborhood of a point refers to all points within a radius of ϵ from that point.

The algorithm classifies points into three categories:

- **Core Point**: A point with at least $MinPts$ points (including itself) within its *ϵ-neighborhood*. These Core Points lie in the interior of clusters.
- **Border Point**: A point that is within the *ϵ-neighborhood* of a Core Point but does not have enough neighbors to be a Core Point itself (it has fewer points than $MinPts$ within Eps). These Border Points can still belong to a cluster.
- **Noise Point**: A point that is neither a Core Point nor a Border Point. These Noise Points are effectively considered outliers.



DBSCAN works in this way:

1. Any two core points that are close enough, within a distance ϵ of each other, are grouped in the same cluster.
2. Any border point that is "close enough" to a core point is also assigned to the same cluster as the core point.
3. Noise points are excluded from clusters entirely.

The algorithm depends on two critical hyperparameters, ϵ and $MinPts$.

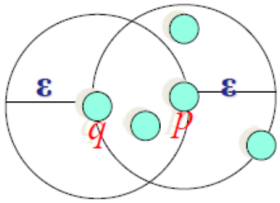
- ϵ specifies the radius of the neighborhood around a point, and it can be chosen using a *k-distance graph*.
- $MinPts$ represents the desired minimum cluster size (the minimum number of points required to form a dense region) and is often chosen using the empirical rule

$$minPts \geq (number\ of\ Dimensions) + 1$$

DBSCAN Connectivity

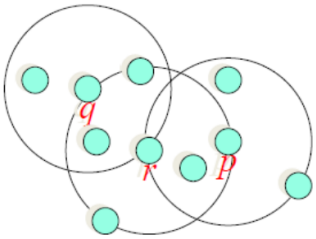
The theoretical foundation of DBSCAN relies on two concepts:

- **Directly density-reachable**: A point q is **directly density-reachable** from point p if q is within the ϵ-*neighborhood* of p and p is a core point.



q is directly density-reachable from p
p is not directly density-reachable from q

- **Density-connectivity**: A point p is **density-connected** to a point q if there is a third point r such that both p and q are *directly density-reachable* from r.



p and q are density-connected to each other by r
Density-connectivity is symmetric

These concepts enable DBSCAN to form clusters by linking density-connected points.

DBSCAN Algorithm

The algorithm proceeds as follows. Initially, the algorithm begins with empty collections for Core Points and Noise Points. It then iterates over each data point $x^{(i)}$ in the dataset X. For each point, the algorithm retrieves all other points that are density-reachable from $x^{(i)}$. Then if $x^{(i)}$ meets the criteria to be a Core Point (i.e., it has enough neighbors within its ϵ radius as defined by $MinPts$), a new cluster is formed. The algorithm adds $x^{(i)}$ to the collection of Core Points and continues to expand the cluster by finding all points connected to these Core Points through density-connectivity. This process identifies connected components in the dataset, and overlapping clusters are merged together to form larger clusters.

Once all Core Points have been processed, the algorithm checks the remaining points in the dataset. For each point not identified as a Core Point, it determines whether the point is within the ϵ radius of any cluster. If it is close enough to a cluster, the point is assigned to that cluster as a Border Point. If not, the point is labeled as noise and added to the collection of Noise Points.

Algorithm Pseudocode:

```
CorePoints =[]
NoisePoints = []
For x(i) in X:
        Retrieve all points density-reachable from x(i)
        if x(i) is a core point, a cluster is formed.
        CorePoints.push(x(i))
For x(i) in CorePoints:
        Retrieve all the connected components.
        Merge the clusters
For x(i) in X/CorePoints:
        if distance from the nearest cluster < Eps:
                Assign x(i) to that cluster
        else
                NoisePoints.push(x(i))
```

DBSCAN – PROS/CONS

Pros

- DBSCAN does not require one to specify the number of clusters K in the data a priori, as opposed to K-means.

- Unlike centroid-based algorithms like K-means, DBSCAN can identify clusters of arbitrary shape. It can even find a cluster completely surrounded by (but not connected to) a different cluster. Due to the $MinPts$ hyperparameter, the so-called single-link effect (different clusters being connected by a thin line of points) is reduced.
- DBSCAN has a notion of noise and is robust to outliers.
- DBSCAN requires just two hyperparameters and is mostly insensitive to the ordering of the points in the database.
- DBSCAN is designed for use with databases that can accelerate region queries, e.g. using an R* tree.
- The hyperparameters $minPts$ and ϵ can be set by a domain expert, if the data is well understood.

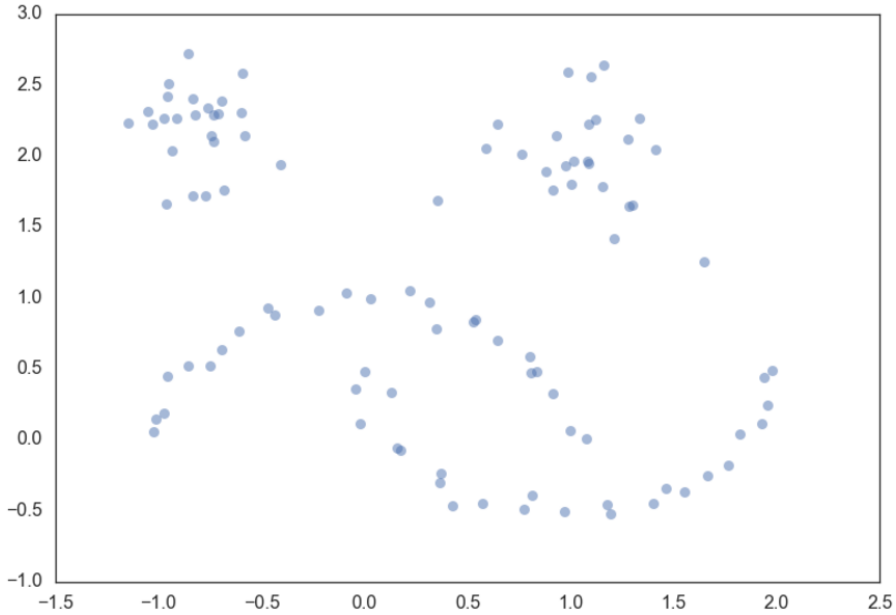
Cons

- DBSCAN is not entirely deterministic: border points that are reachable from more than one cluster can be part of either cluster, depending on the order the data is processed.
- The quality of DBSCAN depends on the distance measure. The most common distance metric used is Euclidean distance. Especially for high-dimensional data, this metric can be rendered almost useless due to the so-called "Curse of dimensionality".
- DBSCAN cannot cluster data sets well with large differences in densities, since the $MinPts$-ϵ combination cannot then be chosen appropriately for all clusters.
- If the data and scale are not well understood, choosing a meaningful distance threshold ϵ can be difficult.

11.7 HDBSCAN

Hierarchical DBSCAN (HDBSCAN) is a clustering algorithm that extends DBSCAN by converting it into a hierarchical clustering algorithm and then, using a technique that we will see in detail, it permits to extract a "flat" clustering based on the stability of clusters.

First, to make it easier to understand let's use an illustrative example. Consider having the following data points. The data considered here is fairly small so we can see what is going on. Moreover, we choose them in a way to have several clusters of different kinds.



Now, the best way to explain HDBSCAN is actually just use it and then go through the steps that occurred along the way teasing out what is happening at each step.

HDBSCAN can break it out into a series of steps:

1. Transform the space according to the density/sparsity.
2. Build the minimum spanning tree of the distance weighted graph.
3. Construct a cluster hierarchy of connected components.
4. Condense the cluster hierarchy based on minimum cluster size.
5. Extract the stable clusters from the condensed tree.

So now let's go through each of these steps.

1) Transform the Space

To find clusters we want to find the islands of higher density amid a sea of sparser noise – and the assumption of noise is important: real data is messy and has outliers, corrupt data, and noise. The core of the clustering algorithm is single linkage clustering, and it can be quite sensitive to noise: a single noise data point in the wrong place can act as a bridge between islands, gluing them together. Obviously, we want our algorithm to be robust against noise so we need to find a way to help 'lower the sea level' before running a single linkage algorithm.

How can we characterize 'sea' and 'land' without doing a clustering? As long as we can get an estimate of density we can consider lower density points as the 'sea'. The goal here is not to perfectly distinguish 'sea' from 'land' – this is an initial step in clustering, not the output – just to make our clustering core a little more robust to noise. So

given an identification of 'sea' we want to lower the sea level. For practical purposes that means making 'sea' points more distant from each other and from the 'land'.

That's just the intuition however. How does it work in practice?

First, we need a very inexpensive estimate of density, and the simplest approach is to use the distance to the k^{th} nearest neighbor. So, we define the core distance, $core_k(x)$ as the distance from a data point x to the k^{th} nearest neighbor. $d(a, b)$ is the distance between two different data points.

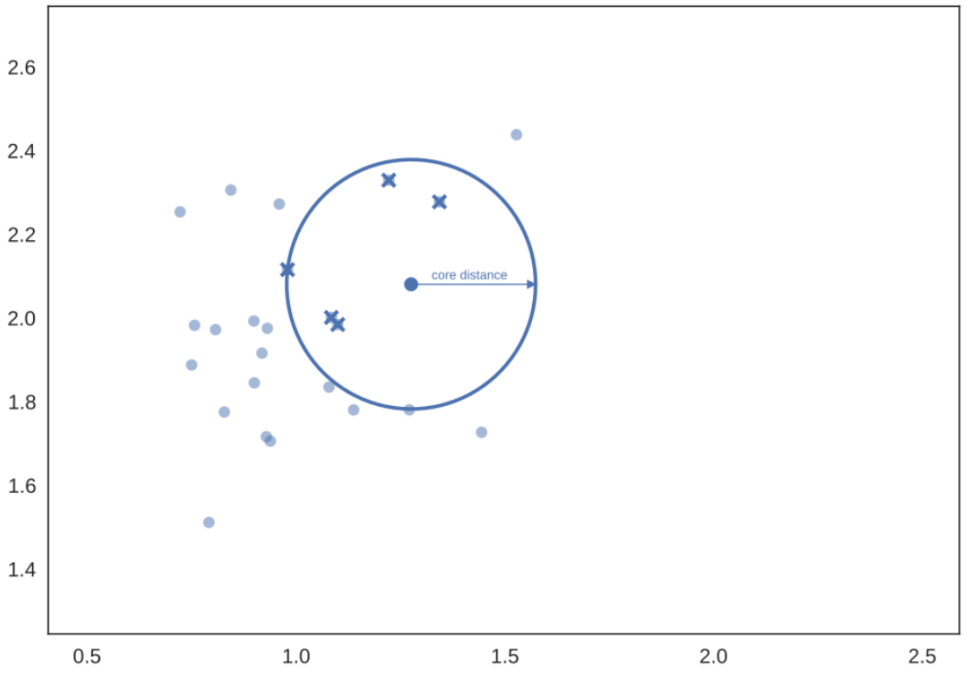
Now we need a way to spread apart points with low density (correspondingly high core distance). The simple way to do this is to define a new distance metric between points which we will call the *mutual reachability distance*. We define ***mutual reachability distance*** as follows:

$$d_{mreach-k}(a, b) = \max\{core_k(a), core_k(b), d(a, b)\}$$

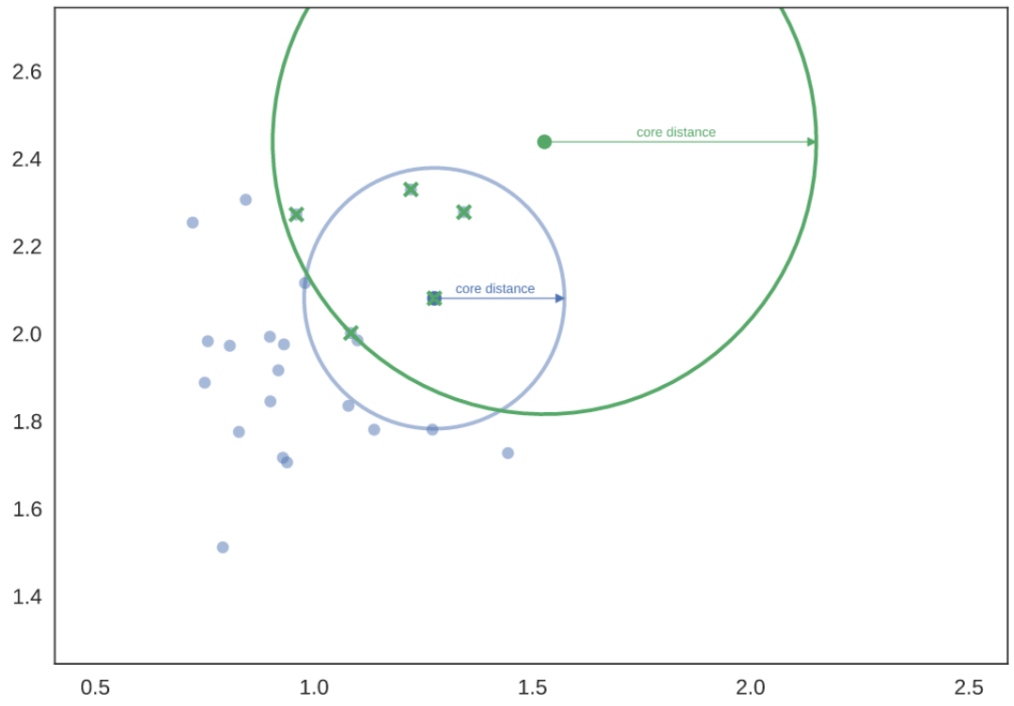
Under *mutual reachability distance* metric dense points (with low core distance) remain the same distance from each other but sparser points are pushed away to be at least their core distance away from any other point. This effectively 'lowers the sea level' spreading sparse 'sea' points out, while leaving 'land' untouched.

The problem here is that obviously this is dependent upon the choice of k; larger k values interpret more points as being in the 'sea'. **Note**: k is NOT the number of clusters K. Don't confuse it!

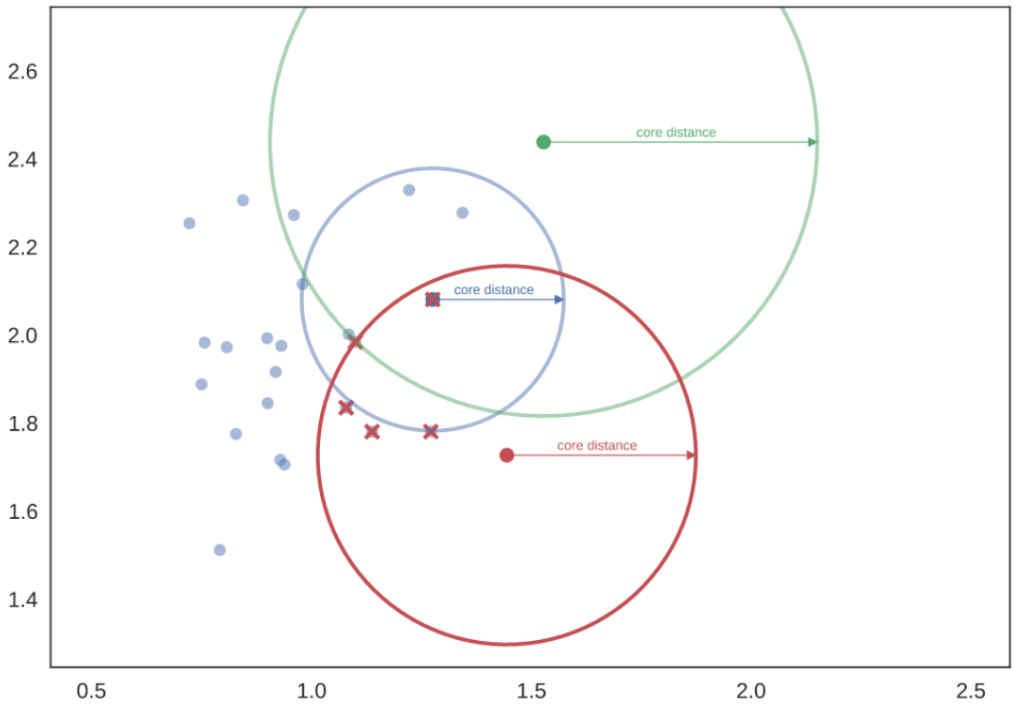
All of this is a little easier to understand with a picture. Let's use a k value of five ($k = 5$). Then for a given point we can draw the core distance as the circle that touches the 5^{th} nearest neighbor, like so:



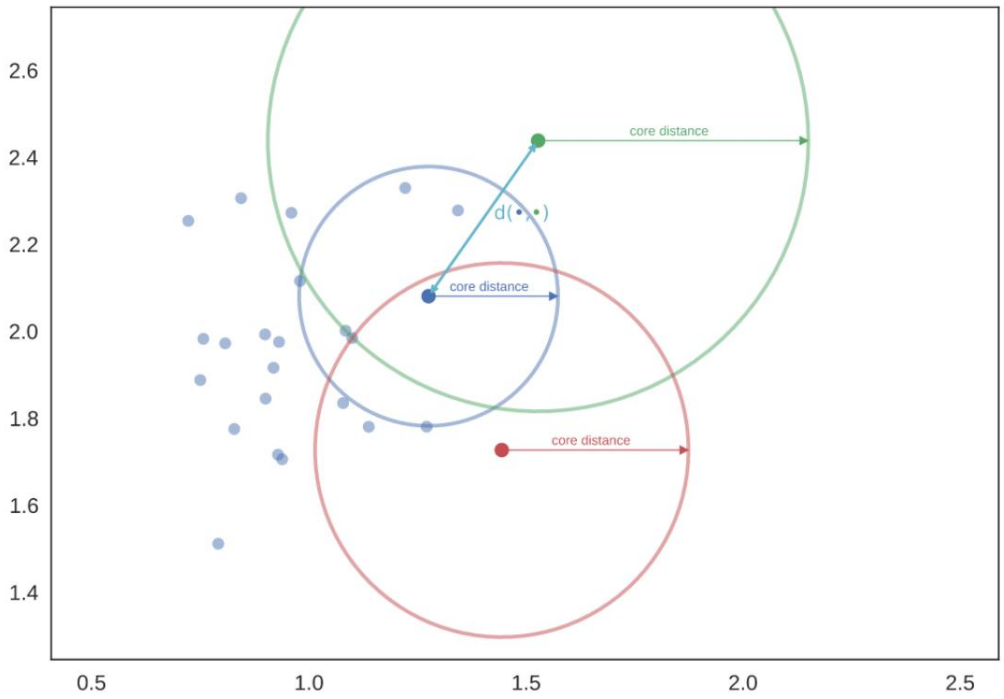
Pick another point and we can do the same thing, but this time will result in a different set of neighbors (one of them even being the first point we picked out): so again, we set the core distance as the distance between the point and 5^{th} nearest neighbor:



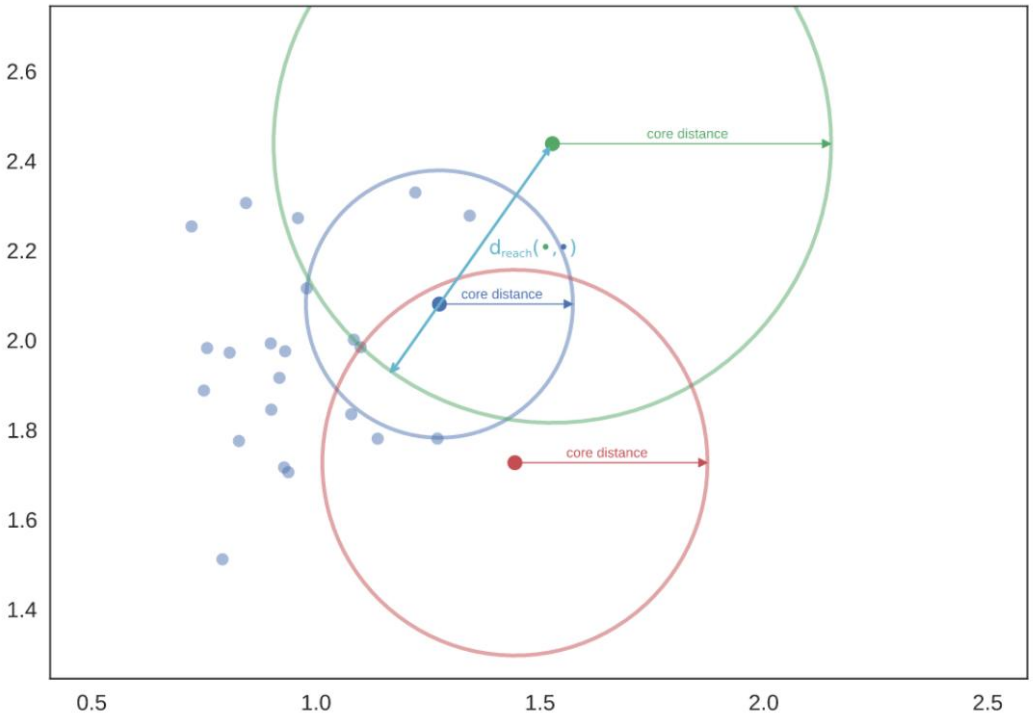
And we can do that a third time for good measure, with another set of five nearest neighbors and another circle with slightly different radius again (a different core distance).



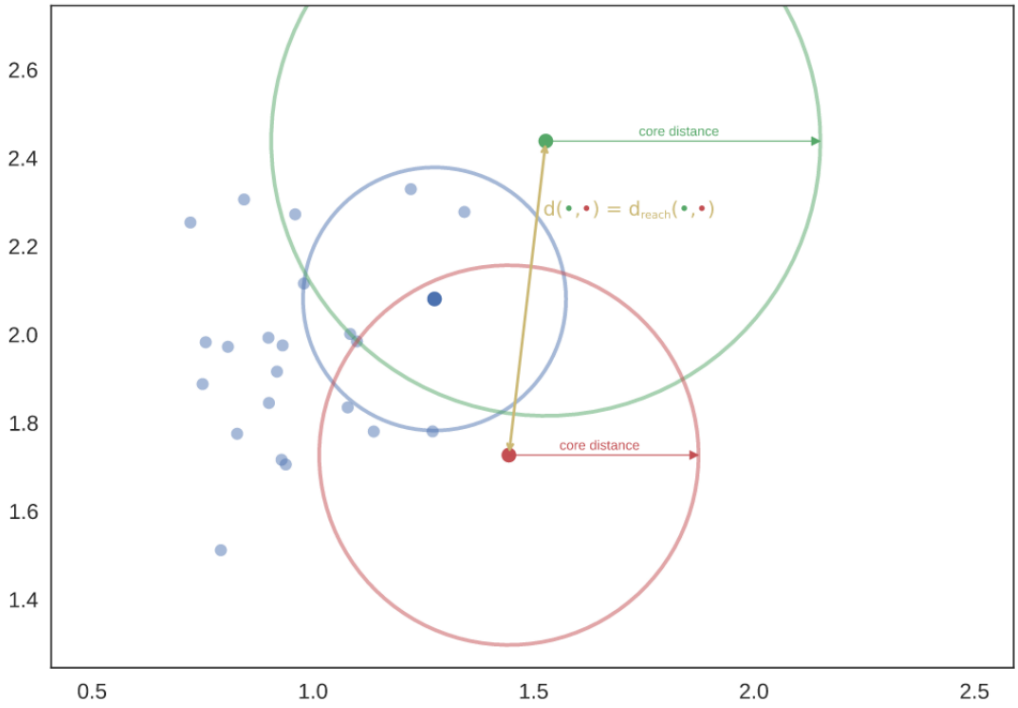
Now the distance $d(a, b)$ between the picked data point which formed the blue circle (a) and the one which formed for green circle(b) can be represented as a light blue arrow as shown below:



We notice that this passes through the blue circle, but not the green circle and the core distance for green ($core_k(b)$) is larger than the distance between blue and green $d(a,b)$. Thus, we need to mark the *mutual reachability distance* between blue and green as larger, specifically equal to the radius of the green circle ($d_{mreach-k}(a,b) = core_k(b)$).



On the other hand, the mutual reachability distance from red to green is simply distance from red to green since that distance is greater than either core distance (i.e. the distance arrow passes through both circles).



In general, there is underlying theory to demonstrate that mutual reachability distance as a transform works well in allowing single linkage clustering to more closely approximate the hierarchy of level sets of whatever true density distribution our points were sampled from.

2) Build the Minimum Spanning Tree

Now that we have a new mutual reachability distance metric on the data we want to start finding the islands on dense data. Of course, dense areas are relative, and different islands may have different densities. Conceptually what we will do is the following:

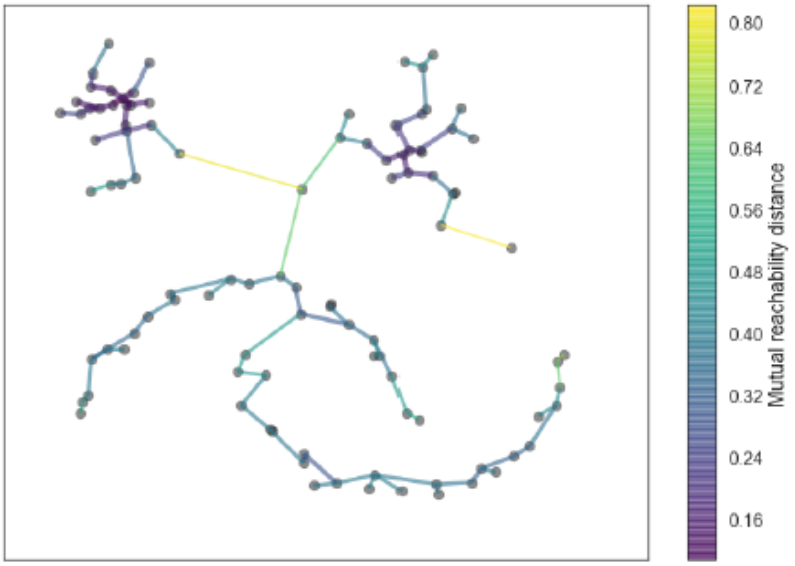
Consider the data as a weighted graph with the data points as vertices and an edge between any two points with weight equal to the mutual reachability distance of those points. Then given a threshold value, starting high and steadily being lowered, starting to drop any edges with weight above that threshold. As we drop edges we will start to disconnect the graph into connected components. Eventually we will end up with a hierarchy of connected components (from completely connected to completely disconnected) at varying threshold levels.

In practice this is very expensive: there are n^2 edges (since we compute mutual reachability distance of each data point to each other data point) and we don't want to run a connected components algorithm that many times. The right thing to do is to find a minimal set of edges such that dropping any edge from the set causes a disconnection of components. But we need more, we need this set to be such that there is no lower weight edge that could connect the components. Fortunately graph theory furnishes us with just such a thing: the minimum spanning tree of the graph.

We can build the **minimum spanning tree** very efficiently via Prim's algorithm: we build the tree one edge at a time, always adding the **lowest weight edge** that connects the current tree to a vertex not yet in the tree.

Other common algorithms to find the minimum spanning tree are Kruskal's algorithm and Borůvka's algorithm.

Running this procedure, you will end up with a minimum spanning tree constructed as in the Figure below.



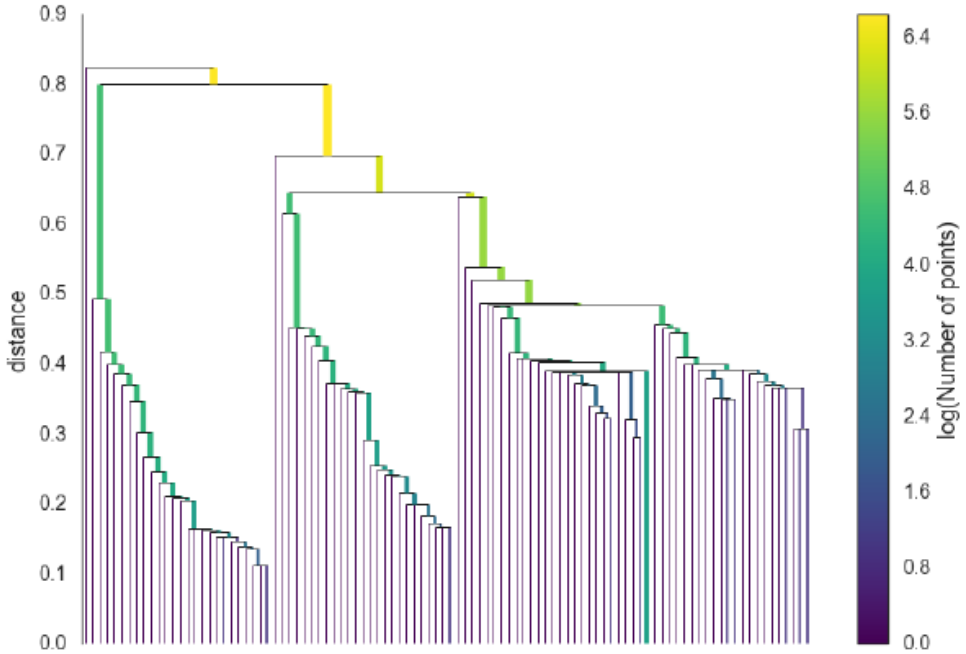
! Note that this is the minimum spanning tree build with mutual reachability distance where we settled $k = 5$, which is different from the pure distance in the graph! Moreover, note that the color of each edge depends on the value of its mutual reachability distance between points it connects (edges values represent the weights).

3) Build the Cluster Hierarchy

Given the minimum spanning tree, the next step is to convert that into the **hierarchy** of connected components.

To construct the hierarchy, what we can do is sort the edges of the tree by distance in ascending order (from smallest to largest) and adopt an **agglomerative approach** (bottom-up). This involves starting with clusters consisting of individual data points and iteratively merging the closest clusters based on the distance value considered, progressively forming larger clusters until we obtain a single cluster.

We can view the result as a dendrogram as we see below:



At this point obtained the hierarchy of clusters based on the distances, what we really want a set of flat clusters. We could do that by simply "**cutting**" the hierarchy at a specific distance and selecting the clusters that it cuts through. This is in practice what DBSCAN effectively does (declaring any singleton clusters at the cut level as noise).

Note: So HDBSCAN shown that DBSCAN corresponds *exactly* with cutting the Robust Single Linkage tree at height Eps.

The question is, how do we know where to cut? DBSCAN simply leaves that as a (very unintuitive) hyperparameter. Ideally, we want to be able to cut the tree at different places to select our clusters (basically, we want to know the **linkage stop condition**). This is where the next steps of HDBSCAN begin and create the difference from robust single linkage. As we will see, to do that HDBSCAN first tries to find a way to pass from a large hierarchy to a more condensed hierarchical tree and then from it tries to individuate the final flat clusters.

4) Condense the Cluster Tree

The first step in cluster extraction is condensing down the large and complicated cluster hierarchy into a smaller hierarchical tree with a little more data attached to each node.

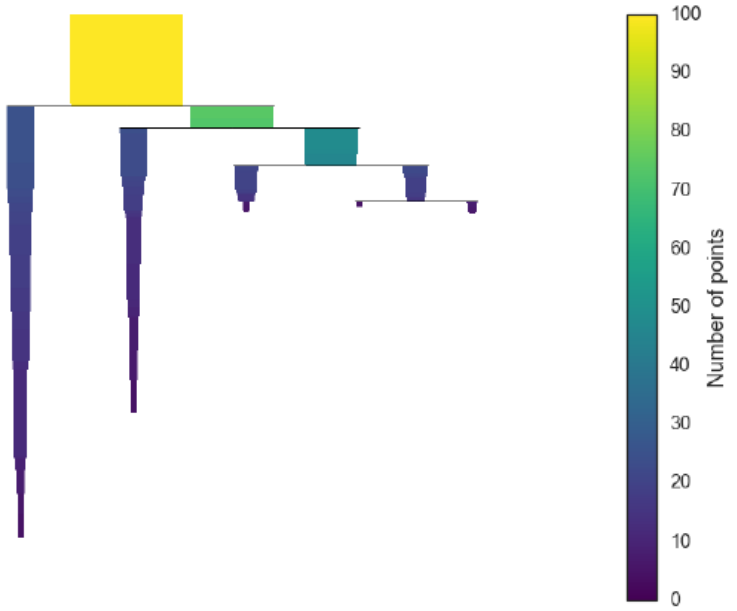
Look at the hierarchy above with a divisive prospective (top-down). You can see that a cluster split often consist just of one or two points splitting off from a cluster; and that is the key point – rather than seeing it as a cluster splitting into two new clusters we want to view it as a single persistent cluster that is 'losing points'. To make this concrete we need a notion of *minimum cluster size* which we take as a hyperparameter to HDBSCAN. The ***minimum cluster size*** ($\boldsymbol{minClusterSize}$) is a threshold above which a set of points is considered as a cluster.

Once we have a value for $minClusterSize$ we can now walk through the hierarchy and at each split ask if one of the new clusters created by the split has fewer points than the $minClusterSize$:

- If it is the case (that we have fewer points than the $minClusterSize$) we declare it to be 'points falling out of a cluster' and have the larger cluster retain the cluster identity of the parent, marking down which points 'fell out of the cluster' and at what distance value that happened.
- If on the other hand the split is into two clusters each at least as large as the $minClusterSize$ then we consider that a true cluster split and let that split persist in the tree.

After walking through the whole hierarchy and doing this we end up with a much smaller hierarchical tree with a small number of nodes, each of which with a higher number of data points and where the information on how the size of the cluster at that node decreases over varying distance is also maintained.

We can visualize this as a dendrogram similar to the one above, where here the width of the line represents the number of data points in the cluster. This width varies over the length of the line as points fall out of the cluster. For our data using a $minClusterSize = 5$ the result looks like the one shown in Figure on side.



This is much easier to look at and deal with, however, we still have a hierarchical clustering. In fact, the next and final step of HDBSCAN consists in extracting flat clusters from the ones shown above, in order to be apple to perform an effective flat clustering.

5) Flat the Clusters (Extract the Clusters)

Intuitively, we would like to choose (extract) those clusters that persist and have a longer lifetime which, looking at the previous plot corresponds to those that have the greatest area of ink in the plot. Graphically this seems pretty straightforward, but how actually do this in a numerical way? We need to add a further requirement that, if you select a cluster, then you cannot select any cluster that is a descendant of it. This intuitive notion of what should be done is exactly what HDBSCAN does at this step as we will see soon. Of course, we need to formalize things to make it a concrete algorithm. This is known as ***stability-based cluster validation method*** of HDBSCAN.

We define λ as a new quantity called **persistence**, equal to the inverse of the *mutual reachability distance* ($1/d_{mreach-k}$) and we use it to consider the persistence of the clusters.

For a given cluster we can then define values λ_{birth} and λ_{death}: we saw that when we explore the tree Top-Down at a certain point the tree is splitted and new clusters are created, that point of the tree is the λ_{birth} of the clusters. When the cluster is splitted again, or we reach the end, that point is called λ_{death}.

In turn, for a given cluster, for each data point p in that cluster we can define the value λ_p as the lambda value at which that data point 'fell out of the cluster' (because of the distance or because it leaves it when the cluster splits into two smaller clusters during condensation) which is a value somewhere between λ_{birth} and λ_{death}.

Said that we can first explore the tree Top-Down computing for each cluster the stability as:

$$stability = \sum_{p \in cluster} (\lambda_p - \lambda_{birth})$$

Then we declare all leaf nodes to be selected clusters, and starting from them we explore the tree Bottom-Up:

- If the sum of the stabilities of the child clusters is greater than the stability of the parent cluster (means that parent cluster is not stable), then we set the cluster stability to be the sum of the child stabilities.
- If, on the other hand, the cluster's stability is greater than the sum of its children (means that parent cluster is stable) then we declare that cluster to be a selected cluster and we unselect all its descendants (from the selected clusters).

Once we reach the root node we call the current set of selected clusters our **flat clusters** and return that.

This can be more clearly summarized with an example:

Suppose we have three clusters, one parent (C1) and two children (C2,C3).

Suppose all leaf nodes are clusters. Now we explore Bottom-Up:

If stability(C2)+ stability(C3) > stability(C1)

the cluster C1 is not stable

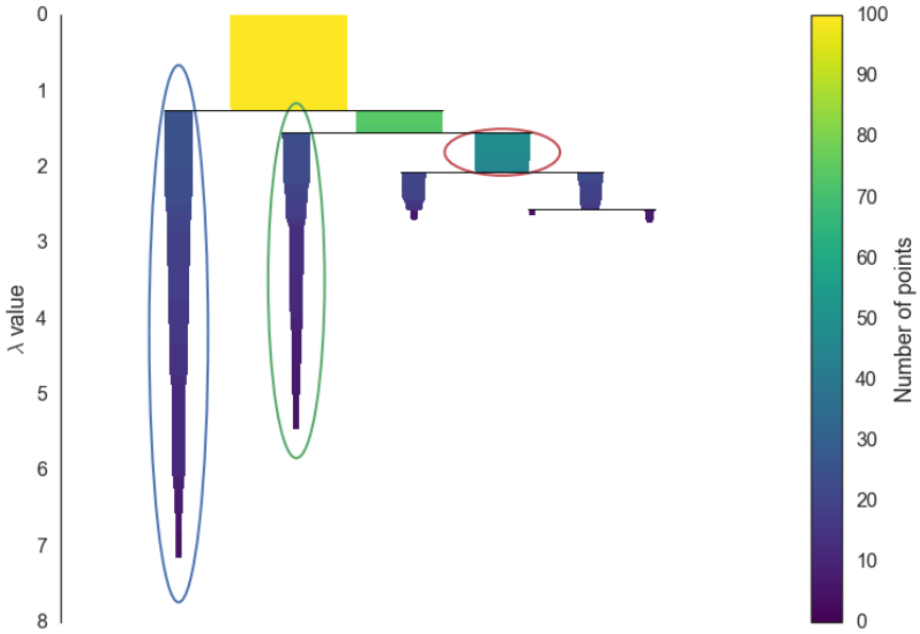
we set stability(C1) = stability(C2)+ stability(C3)

Else if stability(C1) >= stability(C2)+ stability(C3)

C1 is a stable cluster

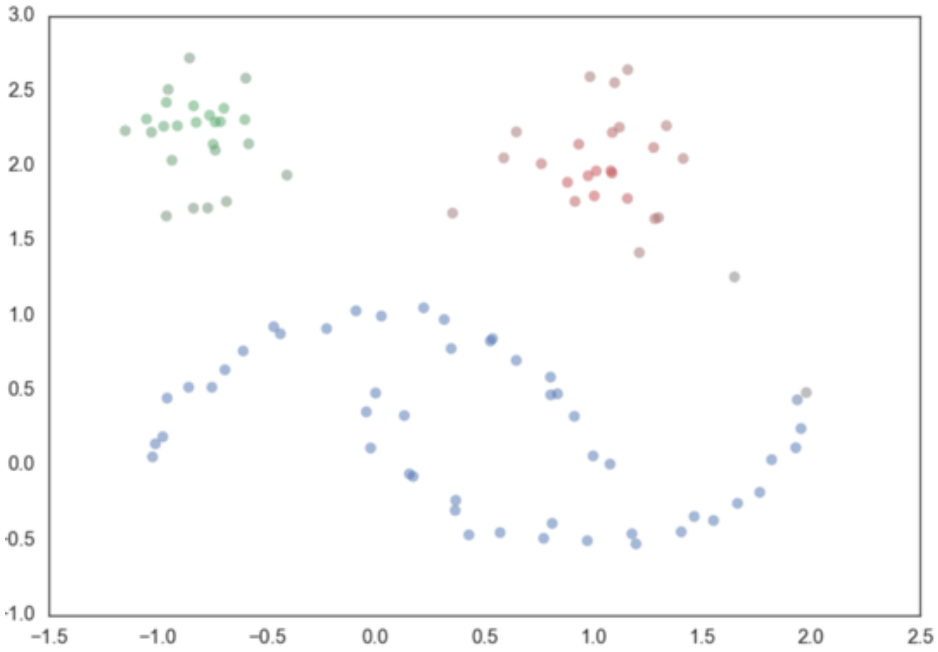
we put C1 in selected clusters and we unset all its descendants from the selected clusters

As result the goal ‘select the clusters in the plot with the largest total ink area’ is reached.



Now that we have the clusters it is a simple enough matter to turn that into cluster labelling. Any point not in a selected cluster is simply a noise point (and assigned the label -1).

With labels in hand we can make the standard plot, choose a color for points based on cluster label, and desaturate that color according to the strength of membership (and make unclustered points pure gray).



And that is how HDBSCAN works.

Recap:

Hierarchical DBSCAN (HDBSCAN) is a clustering algorithm that extends DBSCAN by converting it into a hierarchical clustering algorithm and then, using a stability-based cluster validation method, it permits to extract **"flat" clusters** based on the stability of clusters.

Steps of HDBSCAN:

1. **Transform the space according to the density/sparsity:**
    - **Objective:** Separate dense regions (clusters) from sparse regions (noise).
    - **Method:**

      Defined:

        - core distance, $core_k(x)$, as the distance from a data point x to the k^{th} nearest neighbor.
        - $d(a, b)$ as the original distance between two data points.
        - the ***mutual reachability distance*** as the maximum between the core distances and the original distance between two data points:

$$d_{mreach-k}(a, b) = \max\{core_k(a), core_k(b), d(a, b)\}$$

      Fixed a value for k, we calculate for each pair of data point their *core distances* and their original distance and we set their *mutual reachability distance*.

      Under *mutual reachability distance* metric dense points (with low core distance) remain the same distance from each other but sparser points (noise) are pushed away to be at least their core distance away from any other point. This transformation effectively "spreads out" sparse points (noise), making dense regions more distinct (note: dense regions are left untouched).

2. **Build the minimum spanning tree of the distance weighted graph:**
    - **Objective:** Consider the data as a weighted graph with the data points as vertices and an edge between any two points with weight equal to the mutual reachability distance of those points.
    - **Method:**

      Construct the **minimum spanning tree** using algorithms like Prim's, Kruskal's, or Borůvka's. The procedure builds the tree one edge at a time, adding the **lowest weighted edge** that connects the tree to a vertex not yet in the tree.

3. **Construct a cluster hierarchy of connected components:**
    - **Objective:** Convert the minimum spanning tree into the **hierarchy** of connected components.
    - **Method:**
        - Sort the minimum spanning tree edges by distance in ascending order (from smallest to largest) and adopt an **agglomerative approach** (bottom-up). This involves starting with clusters consisting of individual data points and iteratively merging the closest clusters based on the distance value considered, progressively forming larger clusters until we obtain a single cluster.

- **Result:** A hierarchical structure (dendrogram) representing clusters at different distance (mutual reachability distance) thresholds. However, we still need a **linkage stop condition** to be able to cut the tree at different places and so select (extract) our clusters.

4. **Condense the cluster hierarchy based on minimum cluster size.**
   - **Objective:** Simplify the hierarchy by condensing it into a **smaller** hierarchical tree (focus on meaningful clusters).
   - **Method:** To condense the hierarchy we:
     - set a $minClusterSize$, a threshold above which a set of data points is considered a cluster.
     - traverse the hierarchy top-down:
       - If a split results in two clusters smaller than the $minClusterSize$, we declare it to be 'points falling out of a cluster' and have the larger cluster retain the cluster identity of the parent, marking down which points 'fell out of the cluster' and at what distance value that happened.
       - If a split results in two clusters each at least as large as the $minClusterSize$ we consider that a true cluster split and let that split persist in the tree.
   - **Result:** A condensed hierarchical tree with only stable and meaningful cluster splits.
5. **Extract the stable clusters from the condensed tree.**
   - **Objective:** Select (extract) the most persistent and meaningful clusters from the condensed hierarchical tree.
   - **Method:**
     - Define ***persistence*** (λ) as the inverse of mutual reachability distance ($1/d_{mreach-k}$).
     - For each cluster we define:
       - λ_{birth} as the λ value where the cluster is created as consequence of the tree splitting.
       - λ_{death} as the λ value where the cluster ends (where the cluster is splitted again resulting in new clusters).
       - λ_p as the λ value at which that data point 'fell out of the cluster' (because of the distance or because it leaves it when the cluster splits into two smaller clusters during condensation) which is a value somewhere between λ_{birth} and λ_{death}.
     - Explore the tree Top-Down computing for each cluster the *stability*: the sum of persistence values for all points within a cluster:

$$stability = \sum_{p \in cluster} (\lambda_p - \lambda_{birth})$$

     - Then we declare all leaf nodes to be selected clusters, and starting from them we explore the tree Bottom-Up:
       - If the sum of the stabilities of the child clusters is greater than the stability of the parent cluster (means that parent cluster is not stable), then we set the cluster stability to be the sum of the child stabilities.

- If, on the other hand, the cluster's stability is greater than the sum of its children (means that parent cluster is stable) then we declare that cluster to be a selected cluster and we unselect all its descendants (from the selected clusters).

Once we reach the root node we call the current set of selected clusters our **flat clusters** and return that. Any point not in a selected cluster is simply a **noise point**.

- **Result:** A final set of "flat" clusters.

HDBSCAN – PROS/CONS

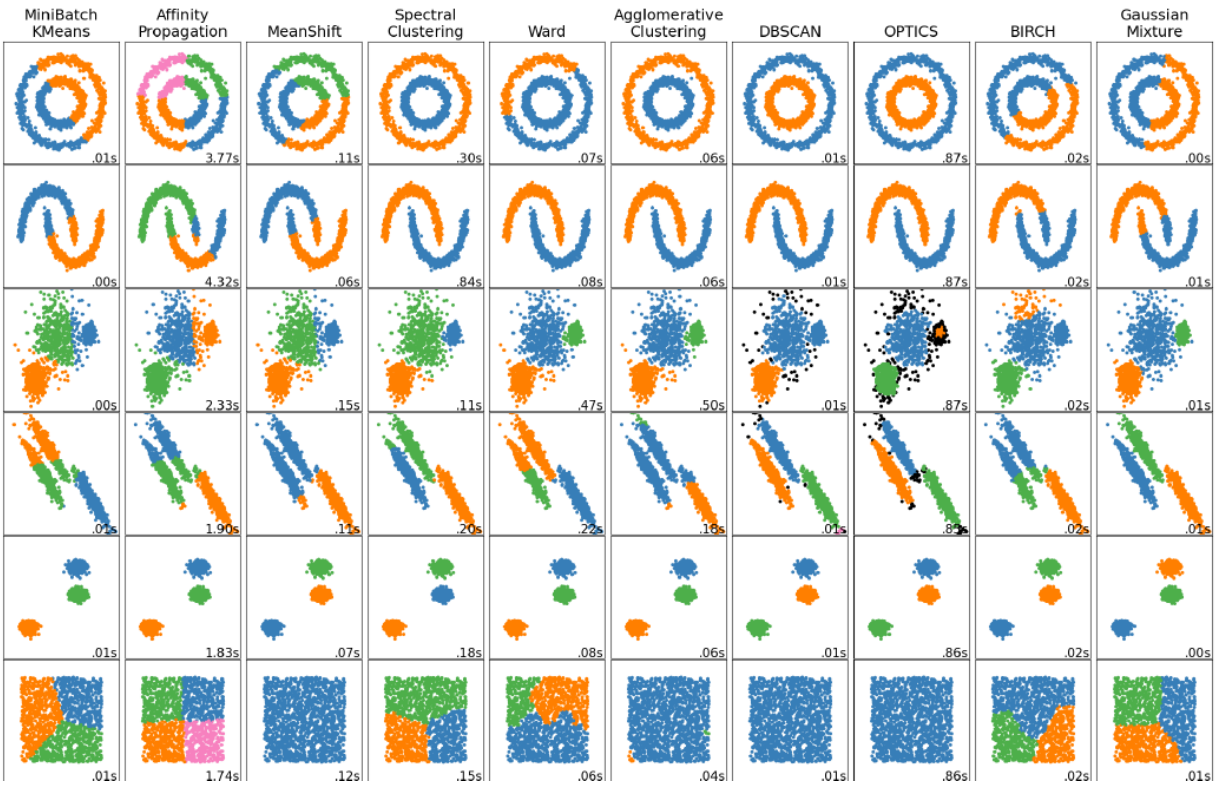
HDBSCAN **PROS:**

- Can handle varying densities
- The algorithm is stable over runs and hyperparameter choices
- Robust to outliers

HDBSCAN **CONS:**

- Some important hyperparameters must be set by hand $(k, minClusterSize)$

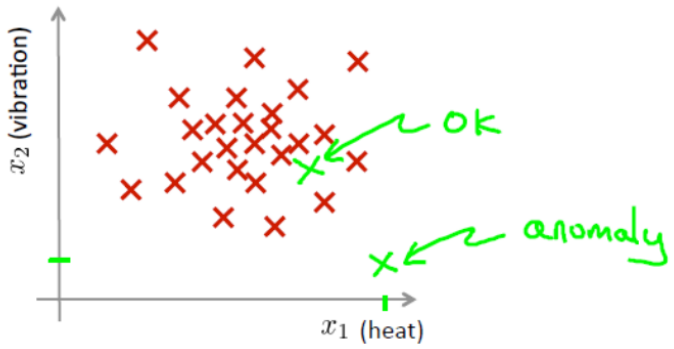
Clustering - A comparison (implementation in scikit-learn)



11.8 GMM Use Case: Anomaly Detection

Anomaly detection involves identifying unusual data points, referred to as **anomalies**, which deviate significantly from the majority of observations in a dataset.

The anomaly detection activity can be seen as a classification task affected by a non-trivial problem: there is very little data actually related to anomalies compared to the rest of the data not related to anomalies (**unbalanced dataset**).



Gaussian Mixture Model (GMM) can be used to detect anomalies in new samples by identifying data points in low density regions.

The idea is to train a GMM only on **non-anomalous data** so that it learns the underlying distribution of the non-anomalous data. Note: We fit the GMM via Expectation Maximization (EM) algorithm so that we find the Gaussian components parameters θ: **mean** and **variance** for each cluster (Gaussian component) and weight (**mixing coefficient**) of each cluster in the mixture.

Once trained, we can then calculate the probability of each new data point ($x_{test}^{(i)}$) to appertain to the mixture as the likelihood of that data point to belong to each of the K Gaussian components of the learned mixture:

$$p(x_{test}^{(i)}|\theta) = \sum_{j=1}^{K} \pi_j \, p_j(x_{test}^{(i)}|\theta_j)$$

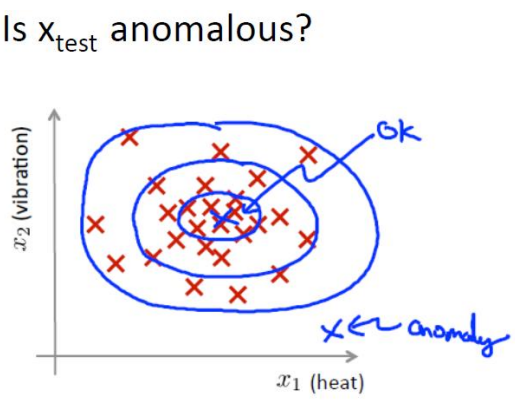
Points with **low probability densities** are likely to lie outside the learned normal data distribution, indicating potential anomalies.

Hower just like that we have probabilities for them, but at the end we aim to classify new samples as anomalies or not. To do that we set a threshold ϵ that it will be used as **anomaly flag**. Thus, if the probability value is below the fixed threshold ϵ, then the new data point is classified as an anomaly, otherwise it is classified as normal.

$$p(x_{test}|\theta) < \epsilon \rightarrow flag\ anomaly$$

$$p(x_{test}|\theta) \geq \epsilon \rightarrow OK$$

Obviously, the threshold ϵ is one of the hyperparameters of the model for which tuning will be required to choose the correct value.



Anomaly Detection Examples:

- **Fraud detection**: Suppose that a i^{th} user performs $x^{(i)}$ activities, a **fraud detection system** may be built by analyzing unusual behavior by focusing on **anomaly behavior** ($p(x) < \epsilon$) after the model $p(x)$ has been generated properly.
- **Manufacturing**.
- **System Monitoring**: Suppose a data center monitors the performance of its i^{th} computer through a set of features $x^{(i)}$, such as memory usage, disk accesses / sec, etc. An anomaly detection system can analyze these characteristics to identify computers exhibiting unusual behavior, which may signal a potential malfunction or irregular usage pattern.

Anomaly Detection Algorithm

1. Choose features that might be indicative of anomalous examples
2. Divide your data in Training set, Cross Validation set and Test Set, paying attention to fill up Training set with **only non-anomalous cases** (otherwise GMM will also learn the anomalies themselves within its distribution)
3. Fit parameters of GMM via EM algorithm
4. Use Cross Validation set to find the value of ϵ that maximizes the number of correctly classified cases
5. Evaluate the error on Test set
    - Given new example x, compute $p(x)$: if $p(x) < \epsilon$ then anomaly found

Anomaly Detection VS Supervised Learning

Anomaly detection:

- Very small number of positive examples ($y = 1$).
- High number of negative examples ($y = 0$).
- Many different "types" of anomalies. Hard for any algorithm to learn from positive examples what anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far.

Supervised learning:

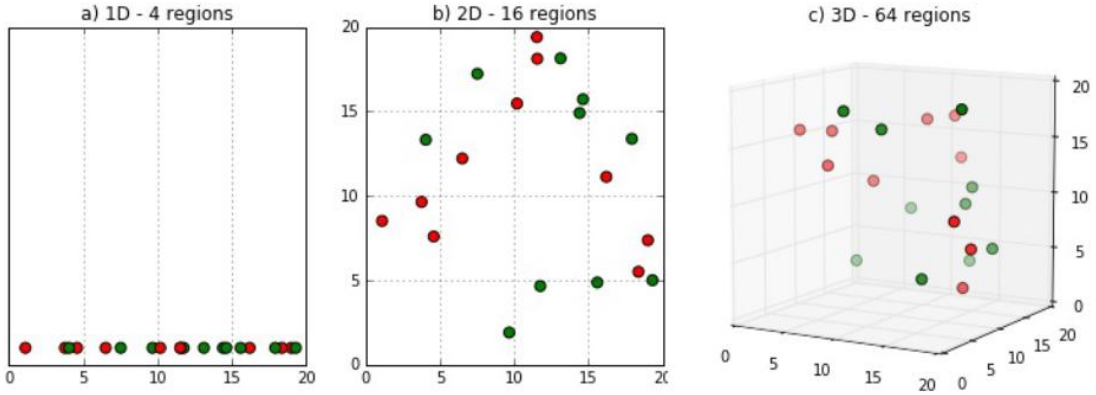
- Large number of positive and negative examples.
- Enough positive examples for the algorithm to get an idea of what the positive examples are like, future positive examples likely similar to the ones in the training set.

12 Dimensionality Reduction

12.1 The Curse of Dimensionality

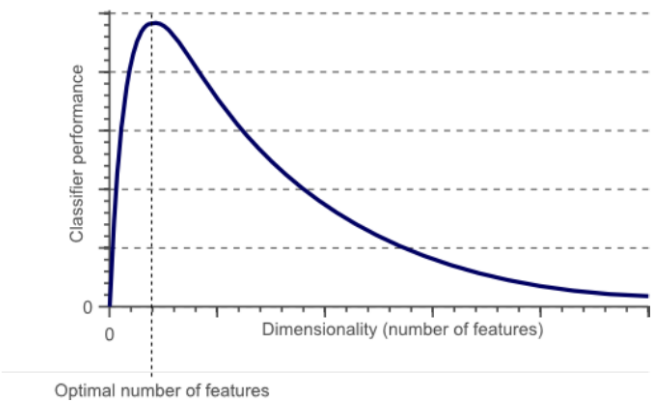
To introduce this concept, let's consider an example. Imagine you have a dataset and aim to classify its samples trying different features:

- **1d:** With just one feature, the classification is rudimentary and far from perfect.
- **2d:** Adding a second feature slightly improves classification, but the problem remains linearly inseparable.
- **3d:** Introducing a third feature in this case enables linearly separable classification



This progression might suggest a strategy: "Can we achieve perfect classification by adding more and more features?" The answer is a definitive **NO**!

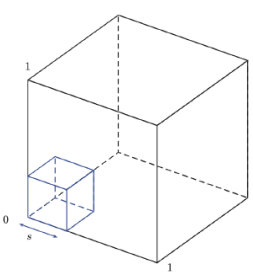
As dimensionality increases, in fact, the classifier's performance increases until a certain optimal number of features is reached. Beyond this point, further increasing the dimensionality without increasing the number of training samples results in a decrease in classifier performance. This decline stems from a phenomenon known as the "curse of dimensionality".

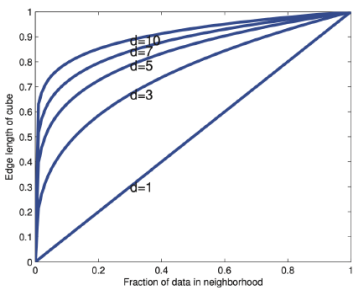


Spoiler: We are interested in that optimal number of features.

Why does this happen? One of the causes of this problem is related to the **density of the training samples**.

Suppose having some evenly spaced sample points in a d-dimensional unit cube. The expected edge length of the cube that contains a fraction f of the data is $f^{1/d}$. In fact, there is an **exponential** increase in volume associated with adding extra dimensions to a mathematical space.





→ Adding dimensions (features) exponentially increases the volume of the feature space, causing sample density to decrease drastically.

E.g. considering the example that we saw above we have that:

- **1d**: In 1d, the feature space is almost fully covered by the points. If we have 30 samples over 4 regions, the sample density is approximately 30/4.
- **2d**: In 2d, the feature space forms an area of 4×4=16 unit squares. The sample density decreases to 30/16, and the samples are more sparse.
- **3d**: In 3d, the feature space forms a volume of 4×4×4=64 unit cubes. The sample density drops further to 30/64, making the samples even more sparse.

As the number of dimensions grows, the data becomes increasingly sparse.

The **"curse" of dimensionality** introduces **sparseness** of training data. The more features we use, the sparser the data become, and consequently making harder to accurately estimate a classifier's parameters (i.e., its decision limits).

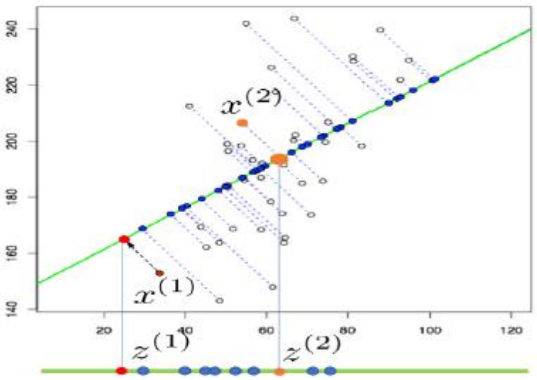
Moreover, if the amount of available training data is fixed, then **overfitting** occurs if we keep adding dimensions. In fact, in a high-dimensional feature space with each feature having a range of possible values, typically an enormous amount of training data is required to ensure that there are several samples with each combination of values.

To maintain adequate sample coverage and avoid overfitting in high-dimensional spaces, the amount of training data must grow **exponentially**—a requirement that is often impractical.

12.2 Introduction to Dimensionality Reduction

To address the challenges posed by the curse of dimensionality, **dimensionality reduction** techniques are employed. These methods focus on reducing the number of dimensions while retaining critical information. This helps create better models, even when starting with originally vary sparse data.

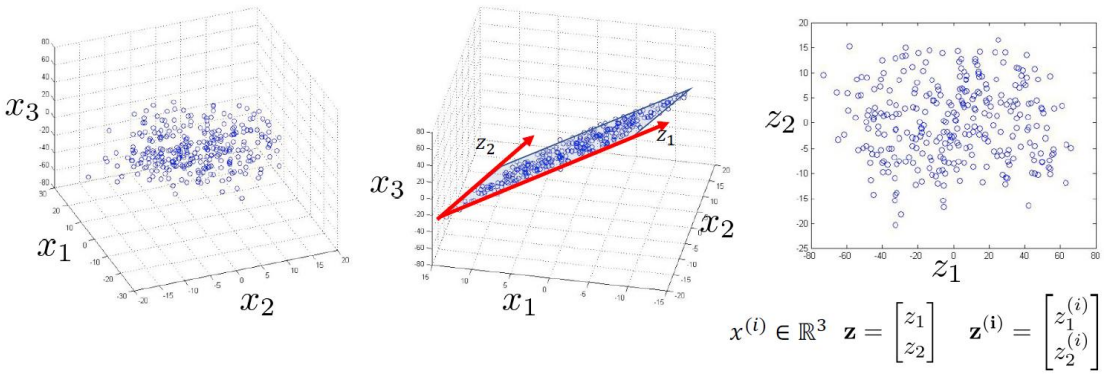
1. **Reduced Representation**: Often, not all features in a dataset are equally informative. Dimensionality reduction enables significant reduction of dimensions of features — for example, reducing from 2D to 1D (halved dimensionality) — while preserving most of the information. This process involves finding a direction of minimization, based on which it is possible to minimize the error in projecting data to a lower dimension; clearly, in the case of the transition from 2D to 1D we will obtain a straight line while, in the transition from 3D to 2D, we will obtain a hyperplane, as visible in the last illustration below.



Reduce data from 2D to 1D

$$x^{(1)} \in \mathbb{R}^2 \rightarrow z^{(1)} \in \mathbb{R}$$
$$x^{(2)} \in \mathbb{R}^2 \rightarrow z^{(2)} \in \mathbb{R}$$
$$\dots$$
$$x^{(m)} \in \mathbb{R}^2 \rightarrow z^{(m)} \in \mathbb{R}$$

Reduce data from 3D to 2D



$x^{(i)} \in \mathbb{R}^3 \quad \mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \quad \mathbf{z^{(i)}} = \begin{bmatrix} z_1^{(i)} \\ z_2^{(i)} \end{bmatrix}$

2. **Data visualization**: Trivially, we are unable to obtain a graphical visualization of data for dimensions higher than 3D, which is why, in the presence of dozens and dozens of features, dimensionality reduction allows us to represent data graphically in 2D or 3D, providing insight into patterns or clusters.
   For instance, imagine you have a large dataset containing 50 features that describe various characteristics of countries around the world. Analyzing such high-dimensional data is challenging—visualizing it is impossible because we cannot graphically represent 50-dimensional space.

	x_1	x_2	x_3	x_4	x_5	
Country	Region	Population	Area	Pop. Density	Coastline	...
Afghanistan	ASIA (EX. NEAR EAST)	31056997	647500	48,0	0,00	...
Albania	EASTERN EUROPE	3581655	28748	124,6	1,26	...
Algeria	NORTHERN AFRICA	32930091	2381740	13,8	0,04	...
American Samoa	OCEANIA	57794	199	290,4	58,29	...
Andorra	WESTERN EUROPE	71201	468	152,1	0,00	...
Angola	SUB-SAHARAN AFRICA	12127071	1246700	9,7	0,13	...
Anguilla	LATIN AMER. & CARIB	13477	102	132,1	59,80	...
Antigua & Barbuda	LATIN AMER. & CARIB	69108	443	156,0	34,54	...
Argentina	LATIN AMER. & CARIB	39921833	2766890	14,4	0,18	...
...	...	...	...	...	...	...

$x \in \mathbb{R}^{50}$

$x^{(i)} \in \mathbb{R}^{50}$

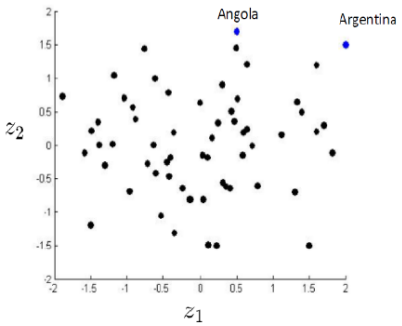
Through dimensionality reduction we can transform the data into a new representation with just two features (z_1, z_2) which summarize these 50 features (they capture the most important patterns or relationships in the data while discarding less significant information). With this transformation, we can then easily plot the countries on a simple 2D graph which allows us to visualize relationships, clusters, or trends among countries, making the data more understandable and offering insights into global patterns.

	z_1	z_2
Country	z1	z2
Afghanistan	1.6	1.2
Albania	1.7	0.3
Algeria	1.6	0.2
American Samoa	1.4	0.5
Andorra	0.5	1.7
Angola	2	1.5
Anguilla	1.4	1.5
Antigua & Barbuda	0.5	0.5
Argentina	2	1.7
...	...	...

$z \in \mathbb{R}^2$

$z^{(i)} \in \mathbb{R}^2$

We reduced data from 50D to 2D



$z \in \mathbb{R}^2$

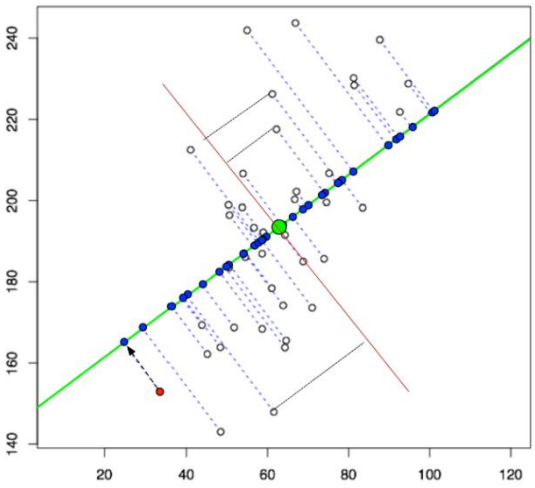
$z^{(i)} \in \mathbb{R}^2$

12.3 Principal component Analysis (PCA)

The most commonly used algorithm for dimensionality reduction is **Principal Component Analysis (PCA)**. PCA leverages the covariance matrix of the data in the dataset to identify directions (principal components) that maximize variance. By projecting the data onto these directions, PCA reduces its dimensionality while retaining as much variability as possible, as we will explore in detail shortly.

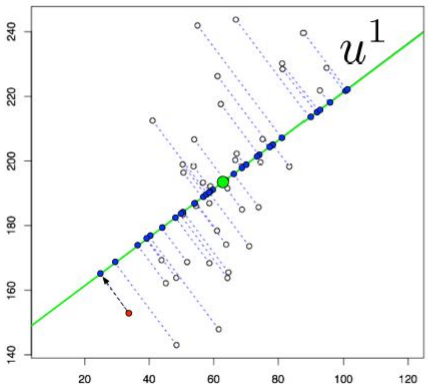
Intuition Behind PCA

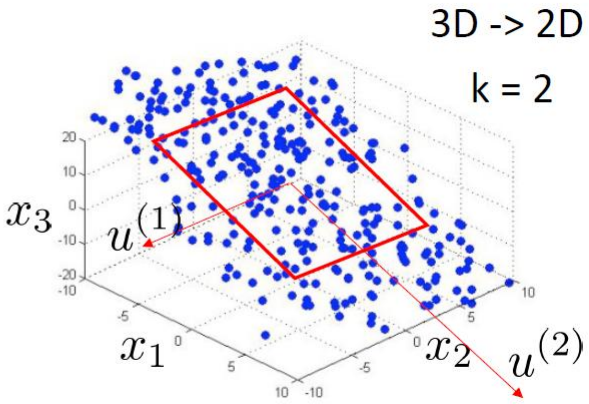
Imagine we have a 2D dataset and aim to reduce its dimensionality to 1D. This means we need to find a single line (in this case) onto which to project this data. How do we determine the optimal line? The projection of each data point onto the line should minimize the distance between the original data point and its projection (represented by the blue dotted lines in the Figure). This is also referred to as **projection error**. So, PCA aims to identify the lower-dimensional surface (a line in this case) which has the minimum projection error.



More specifically, to reduce from 2D to 1D PCA aims to find a direction, so a vector u_1 which to project the data as to minimize the projection error.

In case we want reduce from 3D to 2D, PCA aims to find a pair of vectors u_1 and u_2 onto which to project the data to minimize the projection error.

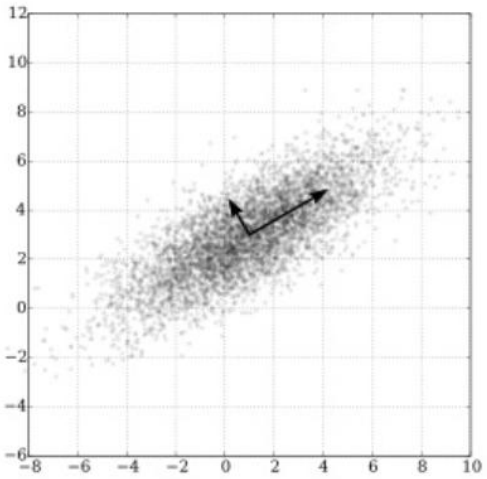




→ In the more general case, to reduce from nD to kD PCA aims to find k vectors $u_1, u_2, \dots, u_k$ (**latent variables**) onto which to project the data as to minimize the projection error.

Note: **Latent variables** are so named because they will probably not have an explicit semantic meaning at the end of the dimensionality reduction process.

PCA aims to find a **linear transformation** of the variables that projects the original ones in a **new coordinate system where** (**most of**) **the variation in the data** can be described with fewer dimensions than the original. In this new coordinate system, the axes are referred to as **Principal Components (PCs)**.



The principal components of a collection of data points in a real coordinate space are a sequence of k **unit vectors** where the j-th vector is the direction of a line that best fits the data while being **orthogonal** to the first $j-1$ vectors.

- Here, a best-fitting line is defined as one that minimizes the average squared projection error.
- These directions (i.e., principal components) constitute an orthonormal basis ensuring that individual dimensions in the transformed space are linearly uncorrelated (this will be clearer soon).

Thus, when performing PCA we will obtain k **orthonormal** PCs $u_1, u_2, \dots, u_k$ such that:

- the **first principal component** (u_1) is the derived variable formed as a linear combination of the original variables that **explains the most variance**
- the **second principal component** (u_2) explains the next highest variance in the data, accounting only for the variance not already captured by the first component.
- etc.

PCA selects the **first k principal components**, condensing most of the original variance into this reduced set of dimensions.

PCA - Problem Formulation

As previously mentioned, the PCA problem can be framed in terms of minimizing the projection error. Specifically, PCA can be defined as the linear transformation that minimizes the **average squared projection error**, also referred to as the **reconstruction error**, between the original data points and their projections onto a lower-dimensional space.

In simpler terms, PCA seeks the best-fitting lower-dimensional representation of the data by finding the directions that minimize the loss of information caused by this projection. This "best fit" is achieved by reducing the distance (error) between the data points and their projected counterparts.

Let:

- $u_j \in \mathbb{R}^n$ represent the j-th **principal direction** of the lower-dimensional space (for $j = 1, \dots, k$)
- The **projection** of a data point $x^{(i)}$ onto the j-th principal direction is given by:

$$z_j^{(i)} = u_j^T x^{(i)}$$

  where $z_j^{(i)}$ is the coordinate of the projected data along u_j. So, z_j is the projection of the data onto the j-th principal direction.

- $\tilde{x}^{(i)}$ represent the reconstructed version of the i-th data sample in the reduced k-dimensional space:

$$\tilde{x}^{(i)} = \sum_{j=1}^{k} z_j^{(i)} u_j$$

This is a linear combination of the original variables in terms of the principal directions.

If n is the original dimensionality, the **reconstruction error** for a reduced k-dimensional representation is given by:

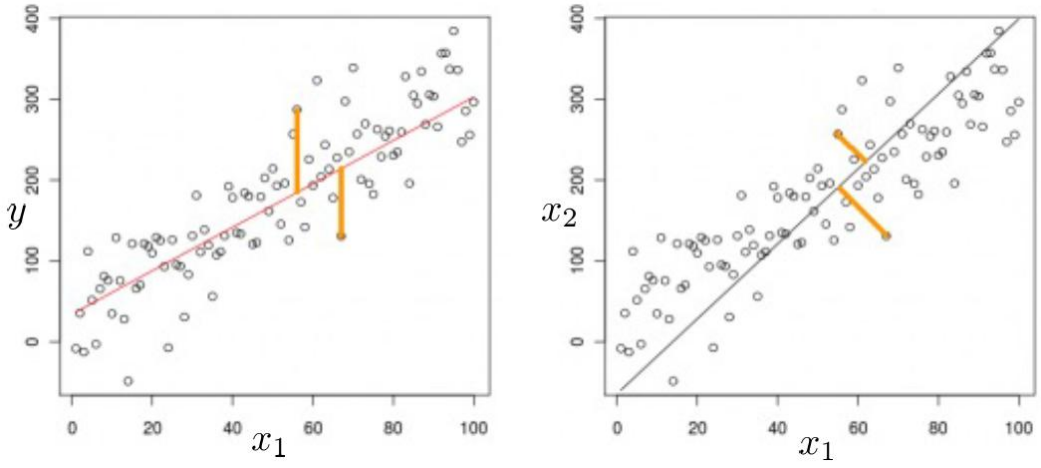
$$J(u_1, \dots, u_k) = \frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - \tilde{x}^{(i)}\right\|^2$$

where $\tilde{x}^{(i)} = \sum_{j=1}^{k} z_j^{(i)} u_j$ and the principal directions $u_1, u_2, \dots, u_k$ are constrained to be orthonormal.

It can be demonstrated that:

- *minimizing* the reconstruction error is equivalent to ***maximizing* the variance** of the projected data.
- minimizing the reconstruction error is equivalent to picking $u_1, u_2, \dots, u_k$ to be the **eigenvectors** of the **covariance matrix** with the largest **eigenvalues**.

N.B: It is important to point out that, unlike linear regression, in which the objective was to calculate the residuals as the distance between the data points and the model that attempted to approximate the latter, in the case of the PCA algorithm, the reconstruction error is not an approximation error but rather an error caused by the compression of the data themselves due to their projection onto a space of lower dimensionality.



Maximizing the Variance: Demonstration

I personally did this demonstration by taking in consideration both the books "*Machine Learning: A Probabilistic Perspective*" and "*Pattern Recognition and Machine Learning*".

Before delving into the demonstration, it's important to review the concepts of **eigenvalues** and **eigenvectors**. For a square matrix $A \in \mathbb{R}^{n\times n}$, a vector $x \in \mathbb{C}^n$ is called an **eigenvector of A** if:

$$Ax = \lambda x$$

where $A \in \mathbb{R}^{n\times n}$, $x \in \mathbb{C}^n$ and $\lambda \in \mathbb{C}$. Here, the $\lambda \in \mathbb{C}$ is called the **Eigenvalue** of A.

Given a dataset of m data points $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ where each $x^{(i)} \in \mathbb{R}^n$ (with n dimensions), our goal is to find the orthogonal projection of $x^{(i)}$ onto a lower dimensionality space of dimensions $k < n$, minimizing the reconstruction error.

Note: For the moment we will assume that the value of k is given. Later we will see how we can determine an appropriate value of k from the data.

Principal directions $u_1, u_2, \dots, u_k \in \mathbb{R}^n$ define the axes of this lower-dimensional subspace.

Since we are just interested in the directions, not in their magnitude, we can constrain each principal direction u_j to be a unit vector ($u_j^T u_j = 1$). Moreover, we also want each u_j to be orthogonal to each other.

→ Basically, we want these principal directions to be **orthonormal** (= orthogonal unit vectors). This means that our k principal directions $u_1, u_2, \dots, u_k$ must satisfy:

$$u_i^T u_j = \delta_{ij}$$

where δ_{ij} is the kronecker delta which is given by:

$$\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

If we denote with z_j the data projected onto the j-th principal direction in the lower-dimensional space, than we know for sure that we can reconstruct the data point $\tilde{x}^{(i)}$ in the original n-dimensional space as a linear combination of each of these projections along their respective projections:

$$\tilde{x}^{(i)} = \sum_{j=1}^{k} u_j z_j^{(i)}$$

The reconstruction error is given by:

$$J(u_1, \dots, u_k) = \frac{1}{m}\sum_{i=1}^{m} \left\| x^{(i)} - \tilde{x}^{(i)} \right\|^2 = \frac{1}{m}\sum_{i=1}^{m} \left\| x^{(i)} - \sum_{j=1}^{k} u_j z_j^{(i)} \right\|^2$$

and our goal is to minimize it.

Let us start by solving for the best single principal direction $u_1 \in \mathbb{R}^n$, corresponding to $k = 1$. We will find the remaining principal directions $u_2, \dots, u_k$ later. In this case the reconstruction error simplifies to:

$$J(u_1) = \frac{1}{m}\sum_{i=1}^{m} \left\| x^{(i)} - u_1 z_1^{(i)} \right\|^2 =$$

$$= \frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)} - u_1 z_1^{(i)}\right)^T \left(x^{(i)} - z_1^{(i)} u_1\right) =$$

$$= \frac{1}{m}\sum_{i=1}^{m} \left[\left(x^{(i)}\right)^T x^{(i)} - 2 z_1^{(i)} u_1^T x^{(i)} + \left(z_1^{(i)}\right)^2 u_1^T u_1\right] =$$

$$= \frac{1}{m}\sum_{i=1}^{m} \left[\left(x^{(i)}\right)^T x^{(i)} - 2 z_1^{(i)} u_1^T x^{(i)} + \left(z_1^{(i)}\right)^2\right]$$

since $u_1^T u_1 = 1$.

Taking derivates with respect to $z_1^{(i)}$ and equating to zero gives:

$$\frac{\partial}{\partial z_1^{(i)}} J(u_1) = \frac{1}{m}\left[-2u_1^T x^{(i)} + 2z_1^{(i)}\right] = 0 \Rightarrow \boldsymbol{z_1^{(i)} = u_1^T x^{(i)}}$$

So, we obtained that the projections $z_1^{(i)}$ are obtained by orthogonally projecting the original data ($x^{(i)}$) onto the first principal direction, u_1.

Substituting $z_1^{(i)} = u_1^T x^{(i)}$ back into the expression for the reconstruction error $J(u_1)$, we get:

$$J(u_1) = \frac{1}{m}\sum_{i=1}^{m}\left[\left(x^{(i)}\right)^T x^{(i)} - 2z_1^{(i)} z_1^{(i)} + \left(z_1^{(i)}\right)^2\right] =$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left[\left(x^{(i)}\right)^T x^{(i)} - 2\left(z_1^{(i)}\right)^2 + \left(z_1^{(i)}\right)^2\right]$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left[\left(x^{(i)}\right)^T x^{(i)} - \left(z_1^{(i)}\right)^2\right]$$

We aim to minimize this quantity with respect to u_1. Hower, since the term $\frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)}\right)^T x^{(i)}$ is constant with respect to u_1, we can drop it when minimizing $J(u_1)$.

$$= const - \frac{1}{m}\sum_{i=1}^{m}\left(z_1^{(i)}\right)^2$$

This leaves:

$$J(u_1) = -\frac{1}{m}\sum_{i=1}^{m}\left(z_1^{(i)}\right)^2$$

Let's pause for a moment to reflect on an important concept. The variance of the projected data (in this case, along the first principal direction) is defined as:

$$var[z_1] = \mathbb{E}[(z_1 - \mathbb{E}[z_1])^2]$$

where $\mathbb{E}[z_1]$ represents the expectation or mean of the projected data.

Expanding the expectation terms, we have:

$$= \frac{1}{m}\sum_{i=1}^{m}\left(z_1^{(i)} - \frac{1}{m}\sum_{i=1}^{m} z_1^{(i)}\right)^2$$

and substituting $z_j^{(i)} = u_j^T x^{(i)}$ the variance becomes:

$$= \frac{1}{m}\sum_{i=1}^{m}\left(u_1^T x^{(i)} - \frac{1}{m}\sum_{i=1}^{m} u_1^T x^{(i)}\right)^2$$

Now the term $\frac{1}{m}\sum_{i=1}^{m} x^{(i)}$ is simply the mean of the original data, which we can denote as μ. Substituting this, we get:

$$= \frac{1}{m}\sum_{i=1}^{m}\left(u_1^T x^{(i)} - u_1^T \mu\right)^2$$

Expanding further:

$$= \frac{1}{m}\sum_{i=1}^{m}\left(u_1^T x^{(i)} - u_1^T \mu\right)\left(u_1^T x^{(i)} - u_1^T \mu\right)^T =$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(u_1^T x^{(i)} - u_1^T \mu\right)\left(\left(x^{(i)}\right)^T u_1 - (\mu)^T u_1\right) =$$

$$= \frac{1}{m}\sum_{i=1}^{m} u_1^T\left(x^{(i)} - \mu\right)\left(x^{(i)} - \mu\right)^T u_1$$

Now, if we assume that the data is mean-centered ($\mu = 0$), this simplifies to:

$$= \frac{1}{m}\sum_{i=1}^{m} u_1^T x^{(i)}\left(x^{(i)}\right)^T u_1 = \frac{1}{m}\sum_{i=1}^{m} u_1^T x^{(i)}\left(u_1^T x^{(i)}\right)^T$$

Substituting back $z_j^{(i)} = u_j^T x^{(i)}$, we find:

$$= \frac{1}{m}\sum_{i=1}^{m} z_1^{(i)}\left(z_1^{(i)}\right)^T = \boldsymbol{\frac{1}{m}\sum_{i=1}^{m}\left(z_1^{(i)}\right)^2}$$

Returning to our previous point, we can see that the quantity we were analyzing, $\frac{1}{m}\sum_{i=1}^{m}\left(z_1^{(i)}\right)^2$, is nothing more than the variance of the projected data when the original data is centered such that its mean is zero.

Therefore, we can write:

$$J(u_1) = \frac{1}{m}\sum_{i=1}^{m}\left[\left(x^{(i)}\right)^T x^{(i)} - \left(z_1^{(i)}\right)^2\right] = -\frac{1}{m}\sum_{i=1}^{m}\left(z_1^{(i)}\right)^2$$

From this, we conclude that *minimizing* the reconstruction error is equivalent to *maximizing* the variance of the projected data:

$$\min_{u_1} J(u_1) = \max_{u_1} var[z_1]$$

This equivalence arises because minimizing $-\frac{1}{m}\sum_{i=1}^{m}\left(z_1^{(i)}\right)^2$ is the same as maximizing $\frac{1}{m}\sum_{i=1}^{m}\left(z_1^{(i)}\right)^2$. This is why it is often said that PCA finds the directions of maximal variance.

If we further develop the expression for $var[z_1]$, we have:

$$var(z_1) = \frac{1}{m}\sum_{i=1}^{m}\left(z_1^{(i)}\right)^2 = \frac{1}{m}\sum_{i=1}^{m} u_1^T x^{(i)}\left(x^{(i)}\right)^T u_1 = u_1^T\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\left(x^{(i)}\right)^T\right)u_1 = \boldsymbol{u_1^T \hat{\Sigma} u_1}$$

where $\hat{\Sigma} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}\left(x^{(i)}\right)^T$ is the **empirical covariance matrix** (covariance with data having zero mean).

Note: The original formulation of the covariance matrix is given by:

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu\right)\left(x^{(i)} - \mu\right)^T$$

where μ is the mean of the data.

So now equivalently we aim to maximize the variance of the projected data, which means maximize $\boldsymbol{u_1^T \hat{\Sigma} u_1}$ with respect to u_1 subject to the constraint $u_1^T u_1 = 1$ that we set earlier. Here we have a maximization problem with one equality constraint, so we can think to introduce a Lagrange multiplier λ_1 such that:

$$u_1^T \hat{\Sigma} u_1 - \lambda_1 (u_1^T u_1 - 1)$$

In this way we convert the constrained optimization problem into an **unconstrained** one which incorporates the constraint directly into the function, so that when solving for u_1, the constraint is automatically handled as part of the optimization process.

Since we're optimizing with respect to u_1, we can set the derivative of it with respect to u_1 equal to zero:

$$\frac{\partial \left(u_1^T \hat{\Sigma} u_1 - \lambda_1 (u_1^T u_1 - 1) \right)}{\partial u_1} = 2\hat{\Sigma} u_1 - 2\lambda_1 u_1 = 0$$

$$\rightarrow \hat{\Sigma} u_1 = \lambda_1 u_1$$

This equation shows that u_1 must be an eigenvector of $\hat{\Sigma}$ and λ_1 its corresponding eigenvalue.

If we left-multiply both sides by u_1^T and make use of $u_1^T u_1 = 1$, we see that the maximum variance of the projected data is equal to the eigenvalue λ_1:

$$u_1^T \hat{\Sigma} u_1 = u_1^T \lambda_1 u_1$$

$$u_1^T \hat{\Sigma} u_1 = \lambda_1 u_1^T u_1 \quad (since\ \lambda_1\ is\ a\ scalar)$$

$$u_1^T \hat{\Sigma} u_1 = \lambda_1 \quad (since\ u_1^T u_1 = 1)$$

➔ So, we obtained that the variance is a maximum when u_1 is equal to the eigenvector of the covariance matrix $\hat{\Sigma}$ corresponding the largest eigenvalue λ_1. This eigenvector is known as the **first principal component**.

Since we want to maximize the variance, we have to pick the eigenvector of $\hat{\Sigma}$ which corresponds to the largest eigenvalue λ_1.

We can identify additional principal components by choosing each new direction to be that which maximizes the projected variance amongst all possible directions **orthogonal** to those already considered.

For instance, let us find the second principal direction u_2, to further minimize the reconstruction error. As always it must be subject to the orthonormality assumption, in this case $u_1^T u_2 = 0$ and $u_2^T u_2 = 1$. The reconstruction error is:

$$J(u_1, u_2) = \frac{1}{m} \sum_{i=1}^{m} \left\| x^{(i)} - z_1^{(i)} u_1 - z_2^{(i)} u_2 \right\|^2$$

It can be shown that expanding this and then computing $\frac{\partial J}{u_2} = 0$ will yield to:

$$\frac{\partial J}{u_2} = 0 \rightarrow z_2^{(i)} = u_2^T x^{(i)}$$

This shows that the second principal encoding is obtained by projecting $x^{(i)}$ onto the second principal direction u_2.

Substituting $z_2^{(i)} = u_2^T x^{(i)}$ into the error expression yields:

$$J(u_1, u_2) = \frac{1}{m}\sum_{i=1}^{m}\left[\left(x^{(i)}\right)^T x^{(i)} - \left(z_1^{(i)}\right)^2 - \left(z_2^{(i)}\right)^2\right] = \frac{1}{m}\sum_{i=1}^{m}\left[\left(x^{(i)}\right)^T x^{(i)} - u_1^T x^{(i)}\left(x^{(i)}\right)^T u_1 - u_2^T x^{(i)}\left(x^{(i)}\right)^T u_2\right]$$

Since we aim to minimize this quantity with respect to u_1 we consider:

$$J(u_2) = const - \frac{1}{m}\sum_{i=1}^{m} u_2^T x^{(i)}\left(x^{(i)}\right)^T u_2 = const - \boldsymbol{u_2^T \hat{\Sigma} u_2}$$

That means all the other terms becomes constants, so we can drop them.

To ensure u_2 satisfies orthonormality:

- $u_1^T u_2 = 0$
- $u_2^T u_2 = 1$

we introduce Lagrange multipliers λ_{12} and λ_2, so that the optimization problem becomes:

$$u_2^T \hat{\Sigma} u_2 - \lambda_2(u_2^T u_2 - 1) - \lambda_{12}(u_1^T u_2 - 0)$$

Taking the derivative of it with respect to u_2 and setting it equal to zero will give us:

$$\hat{\Sigma} u_2 = \lambda_2 u_2$$

From this we conclude that the solution is given by the eigenvector of $\hat{\Sigma}$ with the second largest eigenvalue (λ_2). This eigenvector is the **second principal component**.

The proof continues in this way. Formally one can use induction to prove it (**prove by induction**).

For the general case of projecting data into a lower k-dimensional space ($k < n$), the optimal projection that maximizes the variance of the projected data is obtained as follows:

1. Subtract the mean of each feature μ_j to center the data around the origin, ensuring each feature has a mean of zero:

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$x_j^{(i)} \leftarrow x_j^{(i)} - \mu_j \qquad for\ i = 1, \dots, m\ \ \&\ \ j = 1, \dots, n$$

2. Compute $\hat{\Sigma}$ the covariance matrix of the centered data as:

$$\hat{\Sigma} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}\left(x^{(i)}\right)^T$$

3. **Select the k eigenvectors** (principal directions) $u_1, u_2, \dots, u_k$ **of the covariance matrix $\hat{\Sigma}$ corresponding to the k largest eigenvalues** ($\lambda_1, \lambda_2, \dots, \lambda_k$). This means we should solve the eigenvalue equation:

$$\hat{\Sigma} u_j = \lambda_j u_j$$

for every principal component u_j with $j = 1, \dots, k$.

A better approach to reach the same result is through eigenvector decomposition methods such as *Eigen Value Decomposition* or *Singular Value Decomposition*.

Eigen Value Decomposition

Eigen Value Decomposition (EVD) states that:

> *"If A be a **square** $n \times n$ matrix with n linearly independent eigenvectors q_j (where $j = 1, \dots, n$), then A can be factored as:*
>
> $$A = Q\Lambda Q^{-1}$$
>
> *where Q is the square $n \times n$ matrix whose j-th column is the eigenvector q_j of A, and Λ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, $\Lambda_{jj} = \lambda_j$. This decomposition applies to any diagonalizable matrix".*

In our case, our covariance matrix $\hat{\Sigma}$ is not simply diagonalizable but it is **symmetric**, which allows for further simplifications:

> *"Every real symmetric matrix A can be factored as:*
>
> $$A = Q\Lambda Q^{T}$$
>
> *where Q is an orthogonal matrix whose columns are the eigenvectors of A, and Λ is a diagonal matrix whose diagonal elements are the corresponding eigenvalues, $\Lambda_{jj} = \lambda_j$". In fact, for a generic matrix Q which is orthogonal, holds that $Q^{-1} = Q^{T}$.*

Applying this to our empirical covariance matrix $\hat{\Sigma}$, we have:

$$\boldsymbol{\hat{\Sigma} = U\Lambda U^{T}}$$

At this point we must **reorder** Λ (and as consequence U) in order to have eigenvalues in descending order (from the higher to the lower). After doing that we will end up with:

$$\hat{\Sigma} = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_n \\ | & | & & | \end{bmatrix}_{n\times n} \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \lambda_n \end{bmatrix}_{n\times n} \begin{bmatrix} - & u_1 & - \\ - & u_2 & - \\ & \vdots & \\ - & u_n & - \end{bmatrix}_{n\times n}$$

You should notice that this is the full eigenvalue decomposition, where U contains all n eigenvectors of $\hat{\Sigma}$, and Λ is the diagonal matrix of all n eigenvalues.

Since in PCA we are interested just on the k principal components we can truncate U by keeping only the first k columns (eigenvectors) which after the reordering corresponds to the largest k eigenvalues. We call this matrix U_{reduce}:

$$U_{reduce} = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_k \\ | & | & & | \end{bmatrix}_{n\times k}$$

Given it, the projection of a data point onto the reduced space can be easily computed as:

$$\boldsymbol{z = U_{reduce}^{T} x^{(i)}}$$

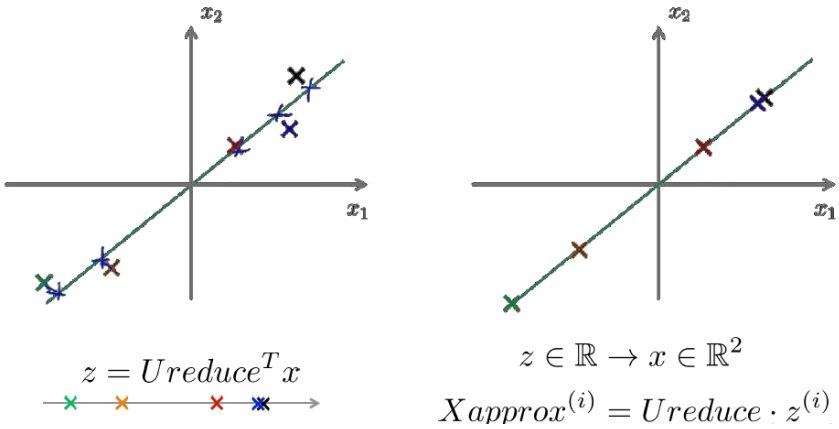
$$\begin{bmatrix} z_1^{(i)} \\ z_2^{(i)} \\ \vdots \\ z_k^{(i)} \end{bmatrix}_{k\times 1} = \begin{bmatrix} - & u_1 & - \\ - & u_2 & - \\ & \vdots & \\ - & u_k & - \end{bmatrix}_{k\times n} \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix}_{n\times 1}$$

Once you map your data into the new lower-dimensional space, you get a result similar to the one shown in the following Figure on the left. Then, wanting to reconstruct the starting data point starting from the compressed one, we can:

$$\tilde{\boldsymbol{x}}^{(i)} = \boldsymbol{U}_{reduce}\boldsymbol{z}_j^{(i)}$$

$$\begin{bmatrix} \tilde{x}_1^{(i)} \\ \tilde{x}_2^{(i)} \\ \vdots \\ \tilde{x}_n^{(i)} \end{bmatrix}_{n\times 1} = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_k \\ | & | & & | \end{bmatrix}_{n\times k} \begin{bmatrix} z_1^{(i)} \\ z_2^{(i)} \\ \vdots \\ z_k^{(i)} \end{bmatrix}_{k\times 1}$$

obtaining an **approximation** of the original data points (see Figure on the right), obviously due to the minimal information loss suffered during the compression process.



Singular Value Decomposition

While PCA fundamentally revolves around identifying the eigenvectors and eigenvalues of the covariance matrix, in practice, this process is commonly executed through **Singular Value Decomposition (SVD)** for better computational stability and efficiency, especially with large datasets. The use of SVD to perform PCA is motivated by the computational challenges and numerical instability that can arise when directly calculating the eigen-decomposition of the covariance matrix, particularly for high-dimensional data.

We will now show that performing SVD on data matrix X, which has been centered (zero mean for each feature) is equivalent to compute EVD on the empirical covariance matrix $\hat{\Sigma}$.

Formally:

> *"Any (real) $n \times m$ matrix A can be decomposed as follows:*
>
> $$A = USV^T$$
>
> *where U is an $n \times n$ matrix whose columns are orthornormal ($U^T U = I_n$), V is an $m \times m$ matrix whose rows and columns are orthonormal (so $V^T V = VV^T = I_m$) and S is a $n \times m$ (rectangular) diagonal matrix containing non-negative **singular values** ($s_j > 0$) on the main diagonal, with 0s filling the rest of the matrix. The columns of U are called the **left singular vectors**, and the columns of V are called the **right singular vectors**".*

Applying in our case we will have:

$$\boldsymbol{X = USV^T}$$

Now, to understand the relationship between SVD and PCA, we can analyze how these matrices relate to the empirical covariance matrix of the dataset X which in vectorial form is expressed as $\hat{\Sigma} = \frac{1}{m} XX^T$. If we substitute the SVD of X into this equation, we get:

$$\hat{\Sigma} = \frac{XX^T}{m} = \frac{(USV^T)(USV^T)^T}{m} =$$

$$= \frac{USV^T VS^T U^T}{m} =$$

Since $V^T V = I_m$ we simplify to:

$$= \frac{USS^T U^T}{m} =$$

$$= U\frac{S^2}{m}U^T = UDU^T$$

where we denoted $D = \frac{S^2}{m}$ to make the notation more compact.

Finally, we can multiply both side by U which will give us the eigenvalue equation for $\hat{\Sigma}$:

$$\hat{\Sigma}U = UDU^T U \rightarrow \boldsymbol{\hat{\Sigma}U = UD}$$

Since the eigenvectors are unaffected by linear scaling of a matrix we see that the right singular vectors (columns) of U are the **eigenvectors** of $\hat{\Sigma}$ and that the matrix D contains the singular values squared s_j^2 simply scaled by $1/m$, corresponding to the eigenvalues λ_j of the covariance matrix $\hat{\Sigma}$ ($\lambda_j = s_j^2/m$).

In the PCA context, the left singular vectors of U represent the principal components, and the singular values in $\hat{\Sigma}$ are related to the variance captured by each principal component. Specifically, the magnitude of each singular value indicates the importance of its corresponding principal component; larger singular values correspond to more significant principal components.

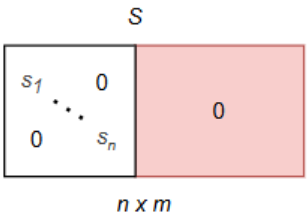
! A natural consequence of performing SVD on a matrix is that the singular values on S matrix are already arranged in decreasing order, that means **we don't have to perform any reordering** of S (and consequently of U). In contrast in EVD we saw that we needed to perform this reordering explicitly.

Therefore, to find the k principal components of X, one can apply SVD on the data matrix X and then take directly the first k columns of U.

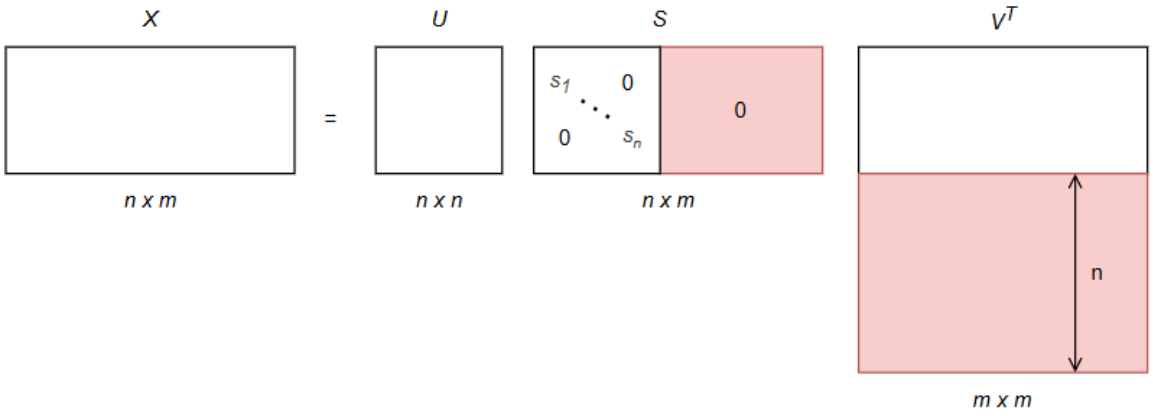
More importantly, you should notice that this approach avoids direct computation of the covariance matrix and its eigen value decomposition, leveraging the numerical stability and efficiency of SVD.

Economy-sized SVD

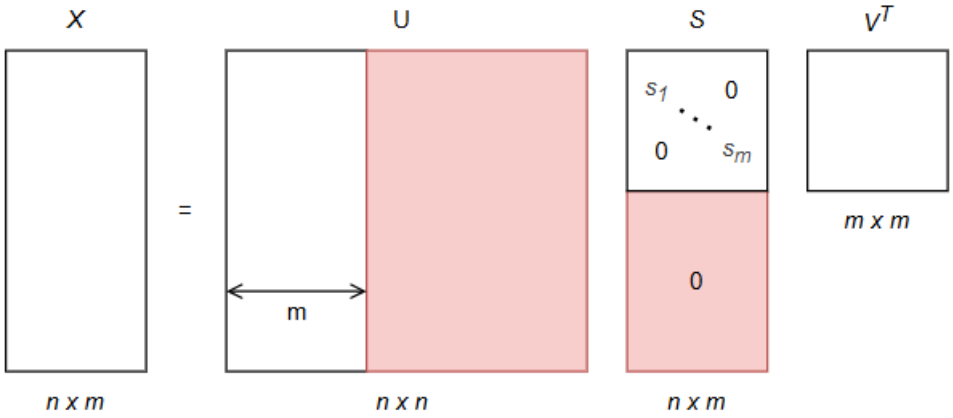
Let us revisit the definition of SVD. We said that SVD factorize a matrix X as $X = USV^T$ where "*S is a $n \times m$ (rectangular) diagonal matrix containing non-negative **singular values** ($s_j > 0$) on the main diagonal, with 0s filling the rest of the matrix*". Explicitly, this means that S will look like:



This of course is the case when $m > n$ (the number of columns instances in our dataset m is higher than the rows features n). In this case we can notice that since we will have at most n singular values, the last $n - m$ columns of V^T are irrelevant because they will be multiplied by the 0s columns of the right part of S:



The same hold in case of $n > m$ (the number of row features in our dataset is higher than the columns instances) in which case we will have at most m singular values. In this case the last $n - m$ columns of U^T are irrelevant, since they will be multiplied by the 0s columns of the right part of S:

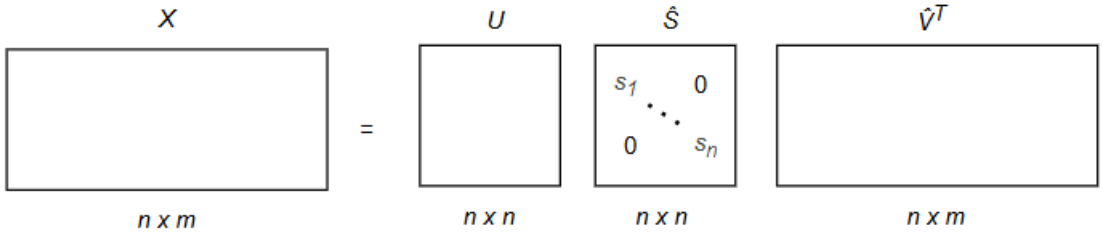


To avoid computing these unnecessary elements, we use the **economy-sized SVD** (or **thin SVD**), which omits the redundant rows or columns. Specifically, only the non-zero singular values of S ($s_j > 0$) are included and the matrices U and V^T are reduced in size accordingly:

- Case $m > n$ (truncate S and V^T):

$$X = U\hat{S}\hat{V}^T$$

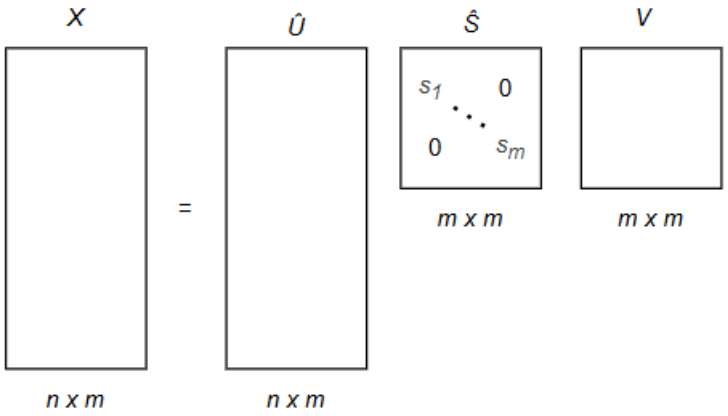
Where U remains $n \times n$, $\hat{S}$ is truncated to $n \times n$ and $\hat{V}^T$ is truncated to $n \times m$.



- Case $n > m$ (truncate S and U):

$$X = \hat{U}\hat{S}V^T$$

where $\hat{U}$ is truncated to $n \times m$, $\hat{S}$ is truncated to $m \times m$ and V^T remains $m \times m$.



So, the economy-sized SVD thus reduces the computational complexity and memory requirements by discarding unnecessary elements.

Finally, we can project the dataset onto the new k-dimensional subspace (defined by the top k principal components) as:

$$Z = U_{reduce}^T X$$

While wanting to reconstruct the starting dataset starting from the compressed one, we can:

$$\tilde{X} = U_{reduce} Z$$

! **Note**: If you prefer work with datasets that are structured as $m \times n$, so where the rows correspond to the dataset instances and where the columns correspond to the features, then to maintain you aligned with this notation I suggest you transpose the data matrix, i.e. X^T.

Recap: Performing PCA Using SVD

1) **Data Preprocessing**

Consider to have a dataset $X = \sum_{i=1}^{m} x^{(i)}$ of dimensions $n \times m$.

To prepare the data for PCA, we ensure the data is properly centered and, if necessary, normalized:

- **Centering the Data**:

  For PCA, we subtract the mean of each feature to center the data around the origin, ensuring each feature has a mean of zero:

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$x_j^{(i)} \leftarrow x_j^{(i)} - \mu_j \qquad for\ i = 1, \dots, m\ \ \&\ \ j = 1, \dots, n$$

where μ_j is the mean of the j-th feature. Centering is crucial for PCA as it aligns the data with the coordinate axes, making it easier to identify the principal components.

- **Normalization (Optional)**:

  If the features have different scales, it is useful to normalize them. A common approach is **z-score normalization**, where each feature is scaled to have zero mean and unit variance:

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j} \qquad for\ i = 1, \dots, m\ \ \&\ \ j = 1, \dots, n$$

  Here, σ_j is the standard deviation of the j-th feature. Normalization prevents features with larger scales from dominating the PCA process.

2) Performing SVD

Once the data is preprocessed (centered and optionally normalized), we perform SVD on the data matrix X:

$$X = USV^T$$

From U, we construct U_{reduce}, by selecting the first k columns of U which represent the k principal components.

3) Transforming the Data

Finally, we project the data onto the new k-dimensional subspace defined by the top k principal components as:

$$Z = U_{reduce}^T X$$

4) Optionally we can reconstruct the data starting from the compressed one as:

$$\tilde{X} = U_{reduce} Z$$

Choice of the Number of Principal Components

How do you choose the optimal number k of principal components?

One approach is to measure the proportion of variance retained after projecting the data into the reduced k-dimensional space.

Algorithm:

Perform:

$$\hat{\Sigma} = U\Lambda U^T$$

Iteratively choose a value for $k = 1,2,3, \dots$ $(k < n)$ and based on it:

- Build U_{reduce}. Then compute:

- $z^{(1)}, z^{(2)}, \dots, z^{(m)}$
- $\tilde{x}^{(1)}, \tilde{x}^{(2)}, \dots, \tilde{x}^{(m)}$

- Check if the normalized reconstruction error is under a given **threshold**, e.g. 0.01if we want at least 99% of the variance preserved.

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - \tilde{x}^{(i)}\right\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2} \leq 0.01$$

Here:

- $\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - \tilde{x}^{(i)}\right\|^2$ represents the average squared projection error.
- $\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)}\right\|^2$ represents the total variance of the data.

We repeat this until the condition above is satisfied (changing k).

Alternative: An equivalent approach to determine k involves using the singular values s_j of S obtained from the SVD of X.

Alghorithm:

Perform:

$$X = USV^T$$

(or better use the economy-sized version).

Iteratively choose a value for $k = 1,2,3, \dots (k < n)$ and check if:

$$1 - \frac{\sum_{j=1}^{k} s_j}{\sum_{j=1}^{n} s_j} \leq 0.01 \Rightarrow \frac{\sum_{j=1}^{k} s_j}{\sum_{j=1}^{n} s_j} \geq 0.99$$

This will represent the **cumulative explained variance** and gives us a measure of how much of the total variance is retained as we include more principal components.

We repeat this until the condition above is satisfied (changing k).

Note: In this case we don't need to compute the reduced representations and the reconstructed approximations of our data, but we simply leverage the diagonal entries s_j of S.

PCA – PROS/CONS

Pros

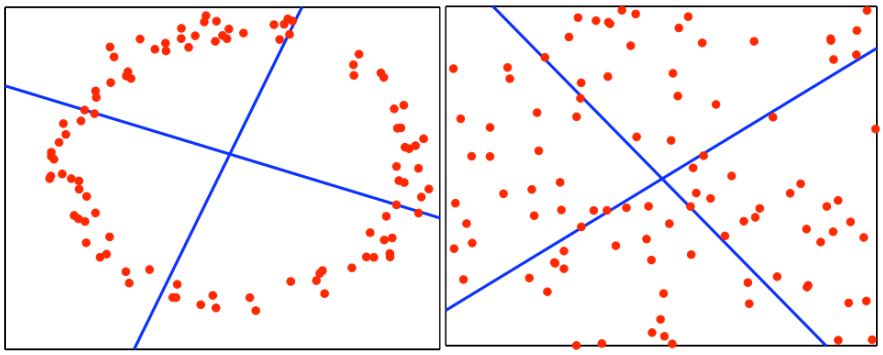
- It can handle varying densities.
- The algorithm is stable over runs and parameter choices.
- Robust to outliers.

Cons

- Some important parameters must be set by hand (like k).
- Cannot distinguish non-linear structure from no structure.

12.4 Kernel PCA

A limitation of traditional PCA is its inability to distinguish between non-linear structures and random noise in the data. For instance, consider two examples shown in Figure: on the left, the data points form a circular structure, while on the right, the data points are randomly distributed.



Despite the clear structure in the left case, PCA will make no difference between these two examples, because it is able only to identify linear relationships.

E.g. for the circular structure, the true "non-linear" dimension of the data is 1 using polar coordinates; however, PCA cannot exploit this non-linear relationship.

Problem: **PCA cannot distinguish non-linear structure from no structure**. While it is effective for identifying linear manifolds in a dataset, many real-world data types contain non-linear structures that PCA fails to capture.

Are there ways to find non-linear, low-dimensional manifolds? An idea could be to map the data into a higher-dimensional space where the non-linear structure becomes linearly separable, making PCA effective in this new space.

We saw during the treatment of Kernel SVM how the technique of kernel substitution allows us to take an algorithm expressed in terms of scalar products of the form $\Phi(x^{(i)}) \cdot \Phi(x^{(j)})$ and generalize that algorithm by replacing the scalar products with a nonlinear **kernel** $k(x^{(i)}, x^{(j)})$. Here basically we aim to apply this technique of kernel substitution to principal component analysis, thereby obtaining a nonlinear generalization called ***Kernel PCA***, which will help to solve the non-linearity problem.

Formulation

Suppose we wish to transform a training sample $x^{(i)} \in \mathbb{R}^n$ from its original feature space into a higher-dimensional feature space using a mapping function $\Phi(x^{(i)}) \in \mathbb{R}^d$. The goal is to perform PCA in this new Φ-space.

Note: I followed the derivation done in the book "*Pattern Recognition and Machine Learning*".

Assuming the projected vectors in the higher-dimensional space are **zero-mean** (a condition we will address later), the covariance matrix in the Φ-space is given by:

$$\hat{\Sigma} = \frac{1}{m}\sum_{i=1}^{m} \Phi(x^{(i)})\Phi(x^{(i)})^T$$

and its eigenvector expansion is defined as:

$$\hat{\Sigma} u_j = \lambda_j u_j \qquad with\ j = 1, \dots, n$$

If we want to apply PCA in the ϕ-space, we need to eigen-decompose $\hat{\Sigma}$. That is, we want to find u_j such that $\hat{\Sigma}u_j = \lambda_j u_j$. However, explicitly computing $\hat{\Sigma}$ involves directly working with $\Phi(x^{(i)})\Phi(x^{(i)})^T$, which can be computationally prohibitive for high-dimensional data. Instead, we aim to solve this eigenvalue problem using **kernel functions** without explicitly computing the Φ-space representation.

We can re-write the eigenvalue equation as:

$$\frac{1}{m}\sum_{i=1}^{m}\Phi(x^{(i)})\Phi(x^{(i)})^T u_j = \lambda_j u_j \qquad with\ j = 1, \dots, n$$

Now, A crucial result (the "**representer theorem**") states that any eigenvector u_j can be expressed as a **linear combination** of the transformed data points $\Phi(x^{(i)})$. That is:

$$\boldsymbol{u_j = \sum_{i=1}^{m} \alpha_j^{(i)} \Phi(x^{(i)})}$$

We can take note of this by looking at the fact that from the expression:

$$\frac{1}{m}\sum_{i=1}^{m}\Phi(x^{(i)})\Phi(x^{(i)})^T u_j = \lambda_j u_j$$

we can get:

$$\rightarrow u_j = \sum_{i=1}^{m} \underbrace{\frac{\Phi(x^{(i)})^T u_j}{\lambda_j m}}_{\alpha_j^{(i)}} \Phi(x^{(i)})$$

This also means that *finding the eigenvectors u_j is equivalent to finding the alpha coefficients $\alpha_j^{(i)}$*, provided $\lambda_j \neq 0$). Thus, in our case finding the eigenvectors reduces finding these alpha coefficients.

Now, plugging this representation of u_j back into the eigenvector equation, we obtain:

$$\frac{1}{m}\sum_{i=1}^{m}\Phi(x^{(i)})\Phi(x^{(i)})^T\left(\sum_{l=1}^{m}\alpha_j^{(l)}\Phi(x^{(l)})\right) = \lambda_j \sum_{l=1}^{m}\alpha_j^{(l)}\Phi(x^{(l)})$$

$$\rightarrow \frac{1}{m}\sum_{i=1}^{m}\Phi(x^{(i)})\left(\sum_{l=1}^{m}\alpha_j^{(l)}\Phi(x^{(i)})^T\Phi(x^{(l)})\right) = \lambda_j \sum_{l=1}^{m}\alpha_j^{(l)}\Phi(x^{(l)})$$

The key step is now to express this in terms of the kernel function $k(x^{(i)}, x^{(l)}) = \Phi(x^{(i)})^T\Phi(x^{(l)})$. We can do this by applying a small trick: multiply both sides by $\Phi(x^{(k)})^T$ to the left:

$$\frac{1}{m}\sum_{i=1}^{m}\Phi(x^{(k)})^T\Phi(x^{(i)})\left(\sum_{l=1}^{m}\alpha_j^{(l)}\Phi(x^{(i)})^T\Phi(x^{(l)})\right) = \lambda_j \sum_{l=1}^{m}\alpha_j^{(l)}\Phi(x^{(k)})^T\Phi(x^{(l)})$$

$$\rightarrow \frac{1}{m}\sum_{i=1}^{m}k(x^{(k)}, x^{(i)})\left(\sum_{l=1}^{m}\alpha_j^{(l)}k(x^{(i)}, x^{(l)})\right) = \lambda_j \sum_{l=1}^{m}\alpha_j^{(l)}k(x^{(k)}, x^{(l)})$$

This can be rewritten in matrix notation as:

$$K^2\alpha_j = \lambda_j m K \alpha_j$$

We can remove a factor of K from both sides of the matrix obtaining:

$$\boldsymbol{K\alpha_j = \lambda_j m \alpha_j}$$

Note: Here K is the **kernel matrix** in which the ij-th element is $k\left(x^{(i)}, x^{(j)}\right)$. This is also known as **Gram matrix**, a matrix whose entries are given by the inner product $\langle \Phi\left(x^{(i)}\right)^T, \Phi\left(x^{(j)}\right) \rangle$.

Thus, α_j is actually an eigenvector of K with eigenvalue λ_j (scaled by m). Similar to the original PCA, we sort the eigenvalues in descending order so that we can select the first k principal components. And given an α, the k-dimensional eigenvector in Φ -space is (as we saw above):

$$u_k = \sum_{i=1}^{m} \alpha_k^{(i)} \Phi\left(x^{(i)}\right)$$

! Note that here u_k indicated the eigenvector consisting of the first k components (i.e. it is equivalent to u_{reduce}).

Having solved the eigenvector problem, we can now represent data in the new space. For a data point x, its projection onto the k principal components is given by:

$$\boldsymbol{z_k = u_k^T \Phi(x) = \sum_{i=1}^{m} \alpha_k^{(i)} \Phi\left(x^{(i)}\right)^T \Phi(x) = \sum_{i=1}^{m} \alpha_k^{(i)} K\left(x^{(i)}, x\right)}$$

Normalized Kernel Matrix

So far, we have assumed that the projected data set given by $\Phi\left(x^{(i)}\right)$ has zero mean, which in general will not be the case. We cannot simply compute and then subtract off the mean, since we wish to avoid working directly in feature space. So again, let's formulate the algorithm purely in terms of the kernel function.

The projected data points after centralizing, denoted $\bar{\Phi}\left(x^{(i)}\right)$, are given by:

$$\bar{\Phi}\left(x^{(i)}\right) = \Phi\left(x^{(i)}\right) - \frac{1}{m}\sum_{k=1}^{m} \Phi\left(x^{(k)}\right)$$

and the corresponding elements of the kernel matrix (Gram matrix) are given by:

$$\bar{K}_{ij} = \bar{k}\left(x^{(i)}, x^{(j)}\right) = \bar{\Phi}\left(x^{(i)}\right)^T \bar{\Phi}\left(x^{(j)}\right) =$$

$$= \Phi\left(x^{(i)}\right)^T \Phi\left(x^{(j)}\right) - \frac{1}{m}\sum_{l=1}^{m} \Phi\left(x^{(i)}\right)^T \Phi\left(x^{(l)}\right) - \frac{1}{m}\sum_{l=1}^{m} \Phi\left(x^{(l)}\right)^T \Phi\left(x^{(j)}\right) + \frac{1}{m^2}\sum_{r=1}^{m}\sum_{l=1}^{m} \Phi\left(x^{(r)}\right)^T \Phi\left(x^{(l)}\right) =$$

$$= k\left(x^{(i)}, x^{(j)}\right) - \frac{1}{m}\sum_{l=1}^{m} k\left(x^{(i)}, x^{(l)}\right) - \frac{1}{m}\sum_{l=1}^{m} k\left(x^{(l)}, x^{(j)}\right) + \frac{1}{m^2}\sum_{r=1}^{m}\sum_{l=1}^{m} k\left(x^{(r)}, x^{(l)}\right)$$

This can be expressed in matrix notation as:

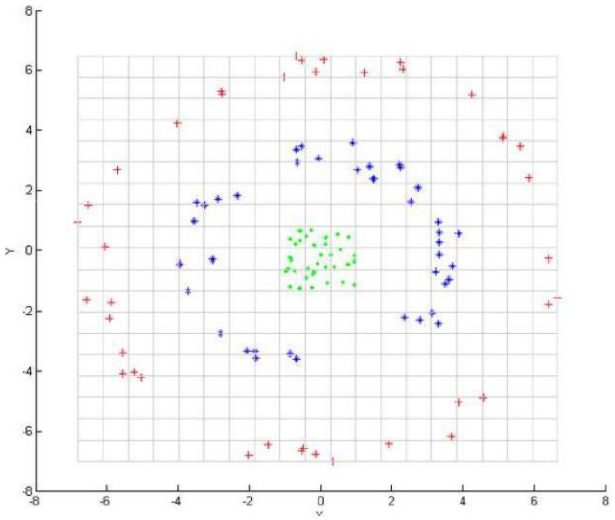
$$\bar{K} = K - 1_m K - K 1_m + 1_m K 1_m$$

where 1_m denotes an $m \times m$ matrix in which every element takes the value $1/m$.

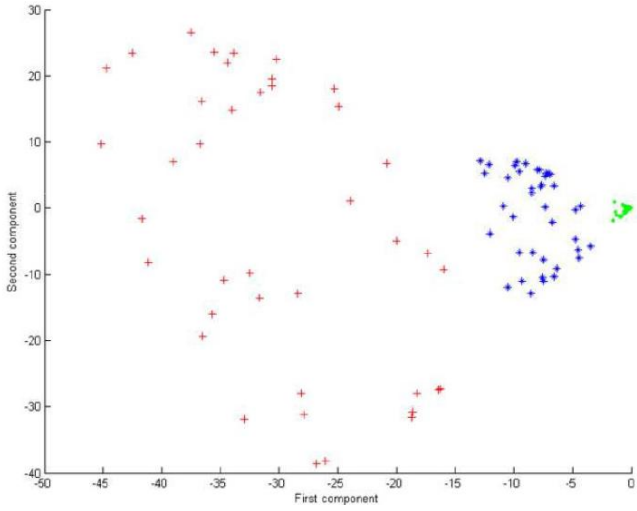
Thus, we can evaluate $\bar{K}$ using only the kernel function and then use $\bar{K}$ to determine the eigenvalues and eigenvectors.

Below are some examples of the use of the Kernel PCA algorithm:

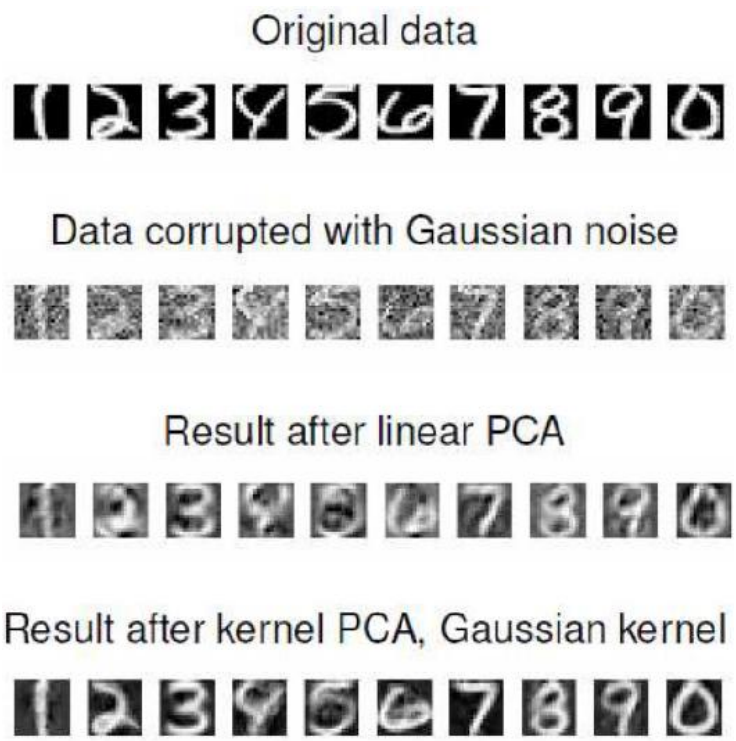
Kernel PCA-Example 1:



Kernel PCA-Example 2:



Kernel PCA – Denoising images:



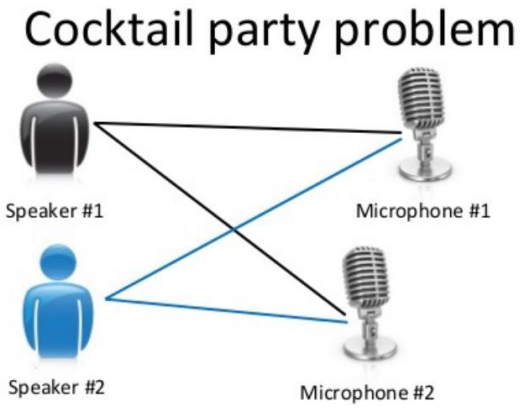
12.5 Independent Components Analysis (ICA)

Independent Component Analysis (ICA) is a technique used to decompose mixed signals into their **independent** source signals.

Note: I followed Andre Ng notes, to which I added my considerations.

Note: ICA is NOT a dimensionality reduction technique, even if it also aims to decompose the data.

As a motivating example, consider the "**Cocktail party**" problem. In this scenario, k speakers are talking simultaneously in a room at the party. In the room there are k microphones, each placed at a different position, so each captures a unique mixture of the k speakers' voices due to varying distances from the speakers. The challenge that the Cocktail Party Problem poses is: "Using these microphone recordings, can we separate out the original k speakers' speech signals?"



To formulate this mathematically:

- Let $s^{(i)}(t)$ represent the independent speakers' speech signals at time t.
- Let $x^{(i)}(t)$ represent the recorded mixed signals captured by the microphones at time t.

For simplicity, assume the number of speakers equals the number of microphones (k) to avoid analytical complications.

Each observed microphone signal $x^{(i)}(t)$ is assumed to be a linear combination of the speakers' speech signals $s^{(i)}(t)$:

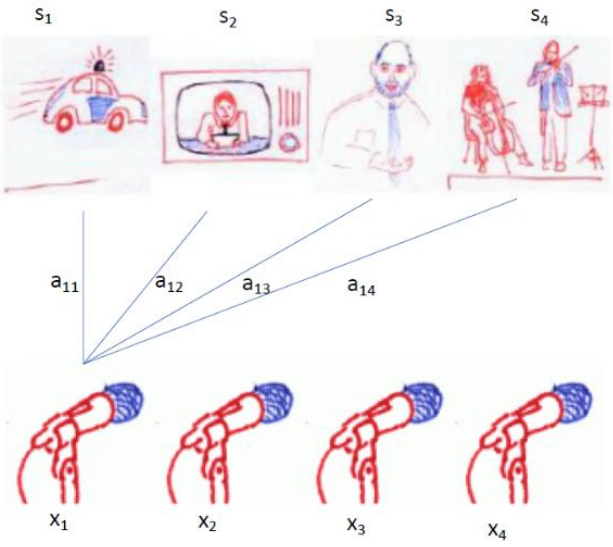
$$x^{(i)}(t) = a_{i1}s^{(1)}(t) + a_{i2}s^{(2)}(t) + \cdots + a_{ik}s^{(k)}(t)$$

where $a_{i1}, a_{i2}, \ldots, a_{ik}$ are coefficients determined by the distances between the microphones and the speakers.

Considering the matrix/vector notation and ignoring the index t, this is:

$$\boldsymbol{x = As}$$

where A is a square matrix ($k \times k$) called ***mixing matrix*** whose element a_{ij} represents the contribution of **the *j*-th source** to the ***i*-th mixed signal**.



Example: Consider a party where we have two speakers and two microphones ($k = 2$): The microphones give us two signals recorded over time (one for each microphone), which we indicate by:

$$x = \left(x^{(1)}(t), x^{(2)}(t)\right)$$

Here $x^{(1)}$ and $x^{(2)}$ are amplitudes and t is the time index. Instead, we indicate the independent signals with:

$$s = \left(s^{(1)}(t), s^{(2)}(t)\right)$$

In this case $x^{(1)}$ and $x^{(2)}$ where be given by:

$$x^{(1)}(t) = a_{11}s^{(1)}(t) + a_{12}s^{(2)}(t)$$

$$x^{(2)}(t) = a_{21}s^{(1)}(t) + a_{22}s^{(2)}(t)$$

and so, our 2×2 mixing matrix will be given by:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Okay, it said that now to accurately estimate the original speech signals $s^{(i)}(t)$ using only the recorded signals $x^{(i)}(t)$, we just need to determine the elements of the mixing matrix A. This matrix represents the linear relationship between the independent source signals $s^{(i)}(t)$ and the observed mixtures $x^{(i)}(t)$, encoding how the sources combine into the recorded signals. However, to achieve this separation, we must assume that the source signals $s^{(i)}(t)$ are statistically independent at each time instant t.

Statistical Independence: In probability theory, saying that two events are **independent** means that the occurrence of one event makes it neither more nor less probable that the other occurs (does not affect the probability of the other). Examples of such independent random variables are the value of a dice thrown and of a coin tossed.

Similarly, in our context, the source signals $s^{(1)}(t)$ and $s^{(2)}(t)$ are statistically independent if the amplitude of one signal at time t provides no information about the amplitude of the other signal. In the cocktail party problem this is actually verified, so we can say that we have Statistical Independence.

Let $W = A^{-1}$ be the **unmixing matrix**. Our goal is to find W, so that given our microphone recordings $x^{(i)}$, we can recover the independent sources by computing $\boldsymbol{s^{(i)} = Wx^{(i)}}$.

More specifically, let:

- $s_j^{(i)}$ the sound that speaker j was uttering at time i.
- $x^{(i)}$ represent the vector of mixed microphones signals at time i.
- w_j^T denote the j-th row of W, so that

$$W = \begin{bmatrix} - & w_1^T & - \\ & \vdots & \\ - & w_k^T & - \end{bmatrix}$$

Then, the j-th source can be recovered as:

$$\boldsymbol{s_j^{(i)} = w_j^T x^{(i)}}$$

In order to make ICA work some **assumptions** have to be made:

1. The independent components are assumed **statistically *independent*** (we already talk about it above).
2. The independent components must have ***non-Gaussian*** distributions.
3. For simplicity, we assume that the unknown **mixing matrix is *squared***, this means that the number of independent components is equal to the number of observed mixtures (we also talk about this above).

ICA Ambiguities

It should be noted that, in the context of use of the ICA algorithm, some ambiguities are always present. In fact, if we have no prior knowledge about the sources and the mixing matrix, it is easy to see that there are some inherent ambiguities in A that are impossible to recover, given only the $x^{(i)}$'s.

The **first ambiguity** is linked to the fact that the algorithm itself is not able to understand which source corresponds to which specific original signal, but it is just able to identify the presence of the different sources. For example, in the case of the cocktail party problem, ICA is able to know that there are two speakers but not to understand which of the two is referred to. This problem is due to the presence of **ambiguous permutations** in the mixing and unmixing processes.

To understand this ambiguity, consider a 2×2 **permutation matrix** P:

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

If z is a vector, then multiplying it by P results in a permutation of z's coordinates:

$$P \cdot z = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} z_2 \\ z_1 \end{bmatrix}$$

Now considering our unmixing equation in ICA (where the sources s are recovered as $s = Wx$) if a permutation matrix P is applied to W, we will obtain PWx and this permuted version will still produce valid independent components, but their order is switched.

➔ This means that, **given only the $x^{(i)}$'s, there is no way to distinguish between W and $P \cdot W$.**

Example: Let us illustrate this with numerical values. Consider:

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad W = \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} \qquad x = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

If we compute $W \cdot x$ and then $P \cdot W \cdot x$ we get:

$$W \cdot x = \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 19 \\ 24 \end{bmatrix}$$

$$P \cdot W \cdot x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 & 6 \\ 3 & 5 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 24 \\ 19 \end{bmatrix}$$

Notice that the results are permuted versions of the original. This demonstrates that the sources can be identified as distinct, but their order remains ambiguous.

For this reason, ICA can effectively separate independent sources but provides no inherent labeling of which source corresponds to which specific signal.

The **second ambiguity** concerns the **scaling of the sources and the unmixing matrix**. This arises from the nature of the decomposition equation:

$$x^{(i)} = A \cdot s^{(i)}$$

For instance, consider the following transformation:

$$x^{(i)} = 2A \cdot \left(0.5 \cdot s^{(i)}\right)$$

Here A is replaced by $2A$ and $s^{(i)}$ is replaced by $0.5 \cdot s^{(i)}$. In this case the product $x^{(i)}$ remains exactly the same as before making it impossible to recover the true scaling of the elements in A and $s^{(i)}$ based solely on $x^{(i)}$.

➔ **There is no way to determine the precise scaling of the elements in the mixing matrix A or the sources $s^{(i)}$ based solely on the observed data $x^{(i)}$.**

More broadly, if any single column of A is scaled by a factor α, and the corresponding independent source $s_j^{(i)}$ is scaled by $1/\alpha$, the observed data $x^{(i)}$ remains unchanged, so the result is indistinguishable from the original data. Thus, there is no unique way to assign a "correct" scaling to the sources or the mixing matrix when working with ICA.

While the scaling ambiguity prevents ICA from determining the absolute amplitude of the sources, for the applications that we are concerned with - including the cocktail party problem - this ambiguity also does not matter. Specifically, scaling a speaker's speech signal $s_j^{(i)}$ by some positive factor α affects only the volume of that speaker's speech. Also, sign changes do not matter, and $s_j^{(i)}$ and $-s_j^{(i)}$ sound identical when played on a speaker. Thus, if the $w^{(i)}$ found by an algorithm is scaled by any non-zero real number, the corresponding recovered source $s_j = w_j^T x$ will be scaled by the same factor; but this usually does not matter.

The **final ambiguity** in ICA concerns the **distributions of the independent components (sources)**. We stated in the hypotheses at the beginning, that the independent sources must have non-Gaussian distributions. But why is this assumption necessary? Let's explore this in detail.

Why Must Sources Be Non-Gaussian? To understand this requirement, consider the following scenario.

Suppose our sources s are drawn from a multivariate Gaussian distribution with zero mean and identity covariance:

$$s \sim N(0, I)$$

We observe so the corresponding x (which we said is given by $x = As$). Then we can easily see that this resulting x will also follow a Gaussian distribution (since linear transformations of Gaussian variables remain Gaussian):

$$x \sim N(0, AA^T)$$

In fact, the distribution of x will also be Gaussian with zero mean and covariance:

$$\mathbb{E}[xx^T] = \mathbb{E}[Ass^T A^T] = AA^T$$

Now, let R be an orthogonal matrix such that $RR^T = R^T R = I$, and consider a new mixing matrix $A' = AR$. Then if the data had been mixed according to A' instead of A, we would have instead observed $x' = A's$.

If we compute the covariance of this x' we get:

$$\mathbb{E}[x'(x')^T] = \mathbb{E}[A'ss^T(A')^T] = \mathbb{E}[ARss^T(AR)^T] = \mathbb{E}[ARR^T A^T] = AA^T$$

Thus, x' will also follow a Gaussian distribution with zero mean and variance AA^T ($x' \sim N(0, AA^T)$).

Hence, whether the mixing matrix is A or A', we would observe data from a $N(0, AA^T)$ distribution. Thus, there is no way to tell if the sources were mixed using A or A'.

➔ **If sources are Gaussian there is no way to know if they were mixed with A or A'.**

This ambiguity arises because the Gaussian distribution exhibits **rotational symmetry**, meaning that any orthogonal transformation (such as a rotation) of a multivariate Gaussian distribution results in another Gaussian distribution with the same statistical properties. Non-Gaussianity breaks this symmetry because non-Gaussian distributions lack this property.

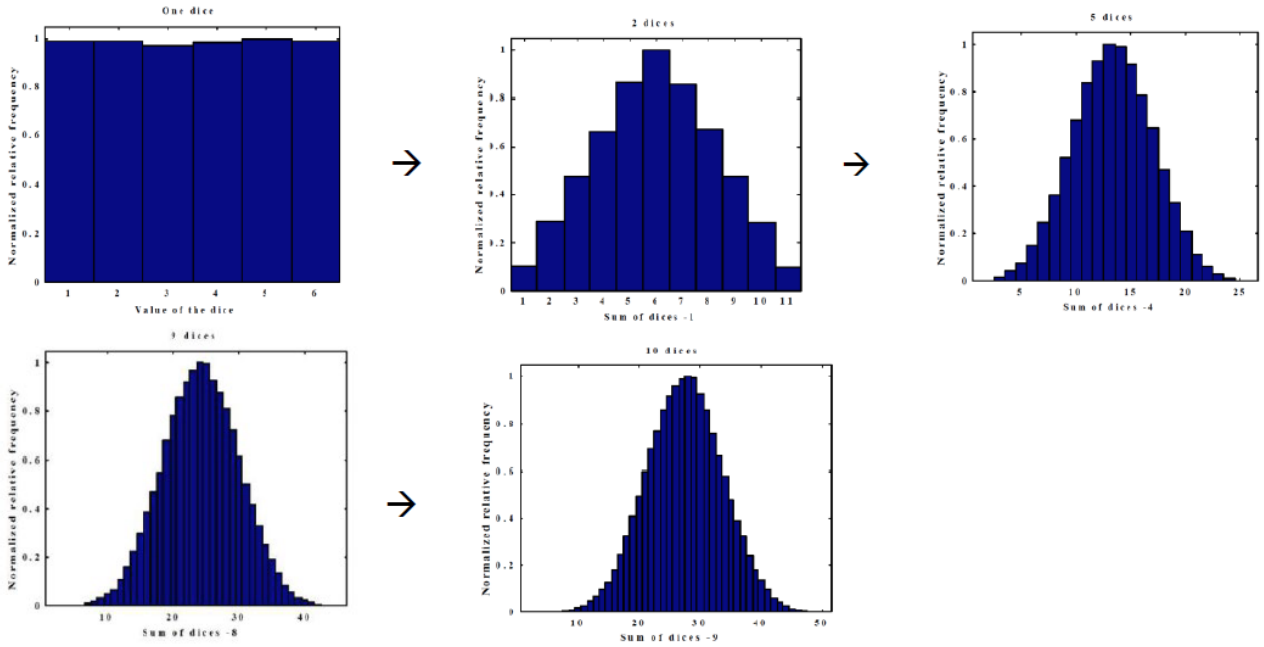
Moreover, the necessity for Non-Gaussianity in ICA arises directly from the Central Limit Theorem.

The **Central Limit theorem** states that the **linear combination** (sum) of a set of random variables, regardless of their original distributions, has a distribution that tends to become increasingly Gaussian as the number of variables in the linear combination increases.

This also means that any linear combination of given random variables will have a distribution that is more Gaussian as compared to the original variables themselves.

For example, in the repeated throwing of non-biased (impartial) dice, with the result given by the sum of the dice faces we have that:

- For one dice the distribution is uniform (one random variable);
- For two dices the sum is piecewise linearly distributed (two random variables in the sum);
- For a set of N dices, the distribution is given by a N-order polynomial distribution (N random variables in the sum);
- With the increase of N ($N \to \infty$), the distribution of the sum tends towards a more Gaussian one, as can be seen from the following Figures.



In the context of ICA, this principle implies that if the independent components were Gaussian, their linear mixtures would also be Gaussian. As consequence, due to the rotational symmetry property of Gaussian distributions, there would be no way to distinguish between different sources of the mixed data.

The Non-Gaussianity is the most important assumption that ICA requires to permit to make estimation.

Densities and Linear Transformations

Before we dive into deriving the ICA algorithm, it's important to first understand how linear transformations affect probability densities.

Suppose we have a random variable s drawn according to some density function $p_s(s)$. We are interested in finding the probability density of the transformed variable x, where $s = Wx$. Basically, we ask: What is p_x?

At first glance, it might seem tempting to directly compute $s = Wx$ and then evaluate p_s at that point to conclude that "$p_x(x) = p_s(Wx)$". However, this is incorrect. The correct procedure for deriving the probability density function for x involves a different relationship, which we will clarify through an example.

Consider the case where s follows a uniform distribution, $s \sim Uniform[0,1]$, so that the density is $p_s(s) = 1$ for $0 \leq s \leq 1$. Now, let's suppose $A = 2$, which gives $x = 2s$. This means that s is scaled by a factor of 2, and consequently, x is uniformly distributed over the interval $[0,2]$. Therefore, the density of x, denoted p_x, is given by:

$$p_x(x) = \frac{1}{2} \quad for\ 0 \leq x \leq 2$$

Notice that this is **not** the same as $p_s(Wx)$. Instead, the correct formula is $p_x(x) = p_s(Wx) \cdot |W|$.

More generally, if s is a vector-valued distribution with density p_s, and $s = Ax$ for a square, invertible matrix A, then the density of x is given by

$$\boldsymbol{p_x(x) = p_s(Wx) \cdot |W|}$$

where $W = A^{-1}$ and $|W|$ represents the determinant of the matrix W.

Thus, the probability density of the observed signal x is **not** equal to the source signal's density, but it is related to the source signal's density multiplied by the scaling factor $|W|$, the determinant of the unmixing matrix W.

ICA Algorithm

We are now ready to derive the Independent Component Analysis (ICA) algorithm.

Since we assumed that the k sources $s^{(1)}, s^{(2)}, \ldots, s^{(k)}$ are **independent** random variables we can express the **joint distribution** of the sources as a product of their individual densities:

$$p(s) = \prod_{j=1}^{k} p_s\left(s^{(i)}\right)$$

Using the formulas that we derived above, the probability density of the observed data x can be expressed as:

$$p(x) = \prod_{j=1}^{k} p_s\left(w_j^T x\right) \cdot |W|$$

Now we aim to maximize this $p(x)$. This means that we want to find the values of the unmixing matrix W (our parameters which are unknow) that maximize the probability of obtaining the observed data x (so $p(x)$). But this is the definition of likelihood, hence we can compute the log likelihood of $p(x)$:

$$l(W) = \sum_{i=1}^{m} \left(\sum_{j=1}^{k} \log p_s\left(w_j^T x\right) + \log|W| \right)$$

and maximize it w.r.t to W.

Remember that logarithm simplifies the product of densities into a sum.

Now, all that remains is to specify the form of the individual source density p_s.

Recall that, in general given a real-valued random variable z, its cumulative distribution function (cdf) F is defined by $F(z_0) = P(z \leq z_0) = \int_{-\infty}^{z_0} p_z(z)\, dz$ and the probability density function is the derivative of the cdf: $p_z(z) = F'(z)$.

Thus, to specify a density for a $s^{(i)}$, all we need to do is to specify a suitable cdf for it. The cdf must be a monotonic function that increases from 0 to 1. Following our previous discussion, we cannot choose the Gaussian cdf, as ICA doesn't work on Gaussian data. However, we know that $sigmoid(s) \in [0,1]$ and is monotonic.

➔ **Idea**: We can choose as a reasonable "default" cdf the sigmoid function $P(s) = sigmoid(s) = g(s)$. Hence, the corresponding density is given by the derivative of the sigmoid $p_s(s) = g'(s)$.

Note: If you have prior knowledge that the sources' densities take a certain form, then it is a good idea to substitute that in here. But in the absence of such knowledge, the sigmoid function can be thought of as a reasonable default that seems to work well for many problems. Also, the presentation here assumes that either the data $x^{(i)}$ has been preprocessed to have zero mean, or that it can naturally be expected to have zero mean (such as acoustic signals). This is necessary because our assumption that $p_s(s) = g'(s)$ implies $\mathbb{E}[s] = 0$ (the derivative of the logistic function is a symmetric function, and hence gives a density corresponding to a random variable with zero mean), which implies $\mathbb{E}[x] = \mathbb{E}[As] = 0$.

Substituting $p_s(s) = g'(s)$ into the log-likelihood function, we obtain:

$$l(W) = \sum_{i=1}^{m}\left(\sum_{j=1}^{k} \log g'\left(w_j^T x\right) + \log|W|\right)$$

To maximize this function with respect to W, we can think of using an iterative approach using gradient ascent. However, to do it we first need to compute its gradient.

From linear algebra, we know that the gradient of the determinant $|W|$ (in case W is invertible) is equal to:

$$\nabla_W = |W|(W^{-1})^T$$

Using this result and by taking derivatives w.r.t. to W we can now derive the **<u>stochastic</u> gradient ascent update rule**.

Let's differentiate each term in the loss function separately:

(a) First Term: $\sum_{i=1}^{m}\sum_{j=1}^{k} \log g'\left(w_j^T x\right)$

Let's first remember that the derivative of sigmoid is:

$$g'(z) = g(z)\left(1 - g(z)\right) \rightarrow \log g'(z) = \log g(z) + \log\left(1 - g(z)\right)$$

Focus on a single term, $\log g'\left(w_j^T x^{(i)}\right)$, our derivative with respect to w_j is:

$$\frac{\partial}{\partial w_j} \log g'\left(w_j^T x^{(i)}\right) = \frac{\partial}{\partial w_j}\left(\log g\left(w_j^T x^{(i)}\right) + \log\left(1 - g\left(w_j^T x^{(i)}\right)\right)\right)$$

Using the chain rule:

$$\frac{\partial}{\partial w_j} \log g\left(w_j^T x^{(i)}\right) = \frac{1}{g\left(w_j^T x^{(i)}\right)} \cdot \frac{\partial}{\partial w_j} g\left(w_j^T x^{(i)}\right) = \frac{1}{g\left(w_j^T x^{(i)}\right)} g\left(w_j^T x^{(i)}\right)\left(1 - g\left(w_j^T x^{(i)}\right)\right) \cdot \frac{\partial w_j^T x^{(i)}}{\partial w_j} =$$
$$= \left(1 - g\left(w_j^T x^{(i)}\right)\right) x^{(i)}$$

$$\frac{\partial}{\partial w_j} \log\left(1 - g\left(w_j^T x^{(i)}\right)\right) = \frac{1}{1 - g\left(w_j^T x^{(i)}\right)} \cdot \frac{\partial}{\partial w_j}\left(1 - g\left(w_j^T x^{(i)}\right)\right) = \frac{1}{1 - g\left(w_j^T x^{(i)}\right)}\left(-\frac{\partial}{\partial w_j} g\left(w_j^T x^{(i)}\right)\right) =$$

$$= \frac{1}{1 - g\left(w_j^T x^{(i)}\right)}\left(-g\left(w_j^T x^{(i)}\right)\left(1 - g\left(w_j^T x^{(i)}\right)\right) \cdot \frac{\partial w_j^T x^{(i)}}{\partial w_j}\right) =$$

$$= -g\left(w_j^T x^{(i)}\right) x^{(i)}$$

So, combining the two we get:

$$\frac{\partial}{\partial w_j} \log g'\left(w_j^T x^{(i)}\right) = \left(1 - g\left(w_j^T x^{(i)}\right)\right) x^{(i)} - g\left(w_j^T x^{(i)}\right) x^{(i)} = \left(1 - g\left(w_j^T x^{(i)}\right) - g\left(w_j^T x^{(i)}\right)\right) x^{(i)}$$
$$= \left(1 - 2g\left(w_j^T x^{(i)}\right)\right) x^{(i)}$$

Thus, for the j-th row w_j, the derivative is:

$$\rightarrow \boldsymbol{\frac{\partial}{\partial w_j} \log g'^{\left(w_j^T x^{(i)}\right)} = \left(1 - 2g\left(w_j^T x^{(i)}\right)\right) x^{(i)}}$$

(b) Second Term: $\sum_{i=1}^{m} \sum_{j=1}^{k} \log|W|$

Above we said that in case W is invertible the gradient of the determinant $|W|$ is equal to:

$$\nabla_W = |W|(W^{-1})^T$$

There in our case:

$$\frac{\partial}{\partial W} \log|W| = \frac{1}{|W|} |W|(W^{-1})^T = (W^{-1})^T$$

Since we supposed W is invertible is also true that:

$$(W^{-1})^T = (W^T)^{-1}$$

Therefore, we get that:

$$\boldsymbol{\frac{\partial}{\partial W} \log|W| = (W^T)^{-1}}$$

For a given training example $x^{(i)}$, the update rule is:

$$W := W + \alpha \left(\begin{bmatrix} 1 - 2g\left(w_1^T x^{(i)}\right) \\ 1 - 2g\left(w_2^T x^{(i)}\right) \\ \vdots \\ 1 - 2g\left(w_k^T x^{(i)}\right) \end{bmatrix} x^{(i)} + (W^T)^{-1} \right)$$

where α is the learning rate. Note: Here since we are maximizing, we have a + instead of – in the update rule.

Once the algorithm converges, we can recover the original independent sources $s^{(i)}$ by applying the learned unmixing matrix W to the observed data $x^{(i)}$:

$$s^{(i)} = W x^{(i)}$$

This completes the ICA algorithm.

+ Non-Gaussianity for ICA Estimation

So, we found out that an approach to solve the ICA problem is via a *Maximum Likelihood Estimation (MLE)*, where the log-likelihood of the unmixing matrix W is maximized to identify the independent components. However, other ICA algorithms use a different approach leveraging a profound implication of the Central Limit Theorem (CLT),

which has practical significance for estimating the independent components. The discussion that will follow is beyond the scope of this course, so here I will give you just a simplified explanation to provide some intuition.

We saw above that the CLT tells us that the linear combination of independent random variables tends to have a distribution closer to Gaussian than the distributions of the original variables. Now, suppose we take a linear combination of the observed data $\tilde{s} = Wx$, according to CLT this will exhibit a distribution that is closer to Gaussian than the original independent components s. Now, if we can identify a linear that is ***least Gaussian***, it strongly suggests that $\tilde{s}$ corresponds to one of the original independent components. Why? Because the independent components are, by assumption, non-Gaussian, and the least Gaussian linear combination of the observed data is most likely to isolate one of them. Once we've identified such a $\tilde{s}$, we can extract the corresponding independent component, exclude it from further consideration, and repeat the process with the remaining data. This iterative procedure enables us to recover the independent components one by one.

This principle—**maximizing the non-Gaussianity of Wx to recover the independent components**—forms the foundation of several ICA algorithms. For example, the popular FastICA algorithm is based on precisely this idea, efficiently identifying directions that maximize non-Gaussianity and leveraging this property to separate the sources. If you are interested in a deeper understanding of this topic, you can refer to this paper.

12.6 Embeddings

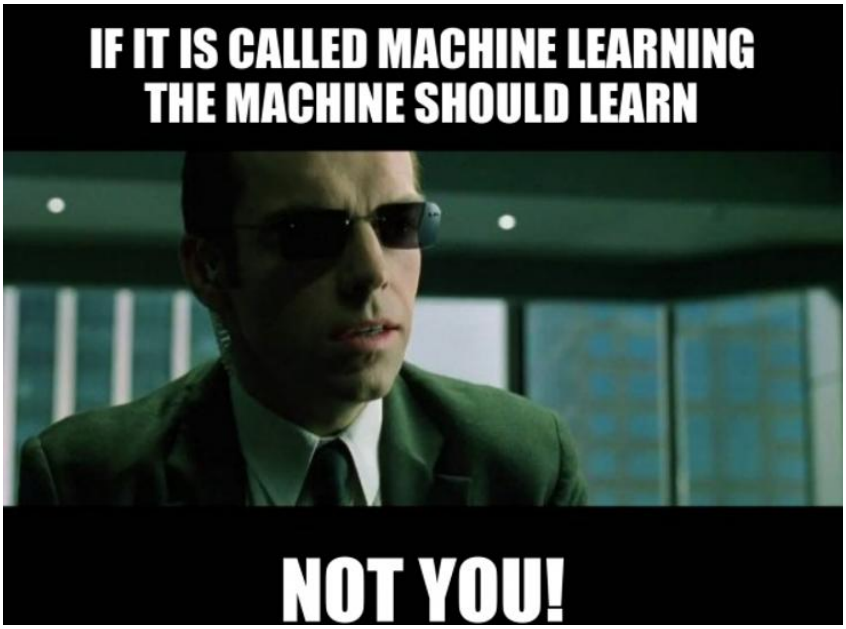
Vector Space Models

In the field of *Natural Language Processing* (NLP), a fundamental task is to transition from a textual representation of words or documents to a structured, mathematical format that can be processed by machine learning algorithms. This is achieved through the use of **vector space models**, which represent text data as vectors in a mathematical space. Each word or document is positioned as a point in this multi-dimensional space, with the dimensions corresponding to different features, such as word occurrences or linguistic patterns.

How is it possible to move from textual representation to vector representation?

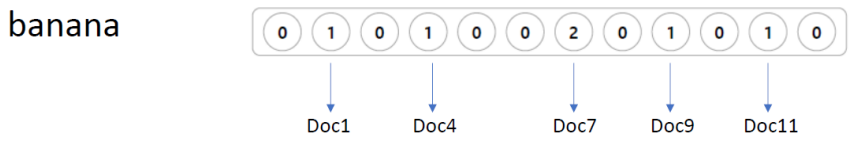
The core idea is to represent a word or any textual element as a vector of numbers.

Note: We have to translate words into numbers because, surprise surprise, machine learning algorithms don't speak human—they only understand numbers.

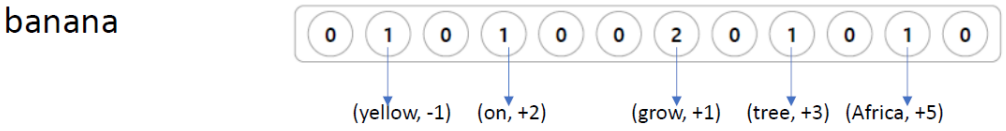


These numbers inside the vectors are designed to encode meaningful information about the word, and there are several ways to define them. Below are three possible representations:

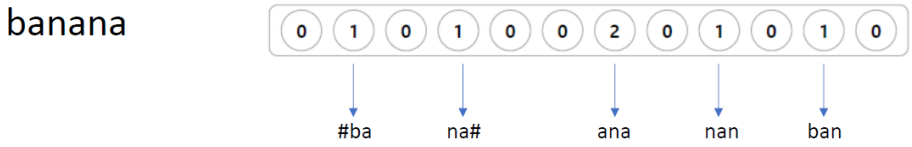
1) The vector can correspond to documents in which the word occurs.



2) The vector can correspond to neighboring word context with respect to the following two documents:
   a. "Bananas grow best in soil that is rich and fertile"
   b. "yellow bananas grow on trees in Africa"



3) The vector can correspond to character trigrams in the word. Trigrams are useful, for example, if you want to create a spelling checker.



Once words are represented as vectors, it becomes possible to compare them to measure their similarity. One common technique is **cosine similarity**, which calculates the cosine of the angle between two vectors. The smaller the angle, the higher the similarity. However, the interpretation of this similarity depends on what vector representation you have chosen for the words.

To illustrate this, consider the following example. We have four documents:

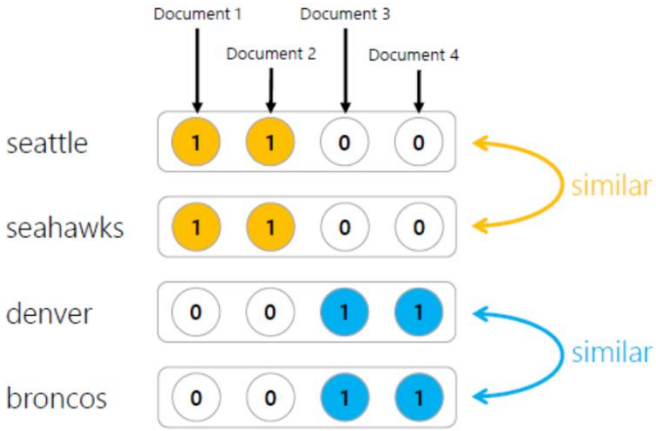
Document 1: "seattle seahawks jerseys"

Document 2: "Seattle Seahawks Highlights"

Document 3: "denver broncos jerseys"

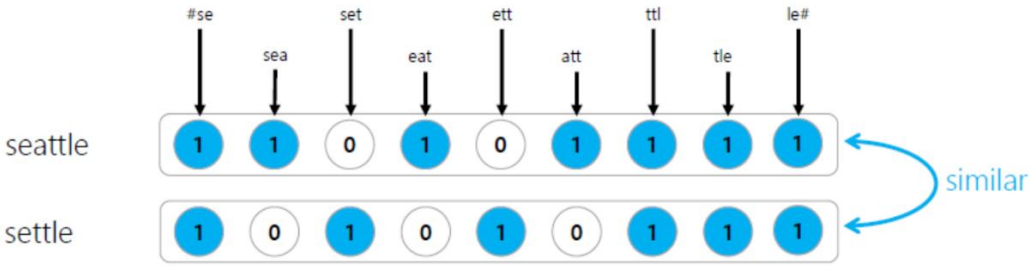
Document 4: "denver broncos highlights"

- If we use the first representation, similarity is typically about how often words appear together in same documents. E.g. in this case we could say that "seattle" and "seahawks" are considered similar since both appear in document 1 and document 2. Same happens for "denver" and "broncos" which are considered similar since they both appear in document 3 and document 4. We refer to this notion of similarity as **Topical similarity**.



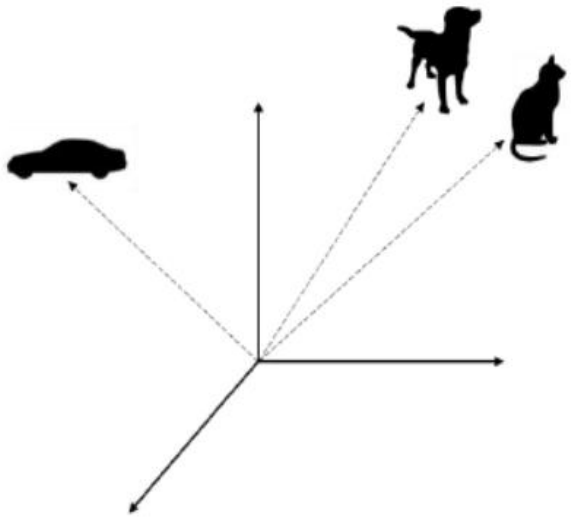
- If we use the second representation, similarity is based on how words appear in similar contexts across documents. E.g. in the case words like "seattle" and "denver" are considered similar because they frequently co-occur in the same context, just as "seahawks" and "broncos" do. We refer to this notion of similarity as **Typical (by-type) similarity**.

- If we use the third representation, similarity depends on the trigrams from which the words are constructed. For example, "seattle" and "settle" would be considered similar due to overlapping trigrams. This notion of similarity is similar to **string edit-distance**, which measures how similar two words are based on their character-level structure.



Embeddings

The vectors we have discussed so far are typically high-dimensional (ranging from thousands to even millions of dimensions) and sparse, meaning that most of their components are zeros. To overcome these limitations, techniques have been developed to generate lower-dimensional, dense vector representations for words while preserving their meaningful relationships. These dense vectors are referred to as **embeddings**.



The process of generating word-embeddings generally involves word vectorization combined with dimensionality reduction.

Embeddings serve as a foundational tool for numerous NLP tasks, allowing words to be represented in a way that captures their meanings, relationships, and contextual similarities in a computationally efficient form.

There are different methods for learning these embeddings:

1. **Matrix Factorization**

   This approach involves factorizing a word-context matrix to derive lower-dimensional dense representations.

   - For instance:
     - **LDA (Latent Dirichlet Allocation):** Learns word-document embeddings.
     - **GloVe (Global Vectors for Word Representation):** Captures relationships between words and their neighboring words by analyzing co-occurrence statistics in a corpus.

2. **Neural Networks**

   These models learn embedding by mapping words to their contexts through an intermediate "bottleneck" layer, which compresses the information into dense vectors.

   - For instance:
     - **Word2Vec:** Learns word embeddings by predicting neighboring words (Skip-Gram) given the target word or using neighboring words to predict the target word (CBOW – Continuous Bag of Words).

Matrix Factorization

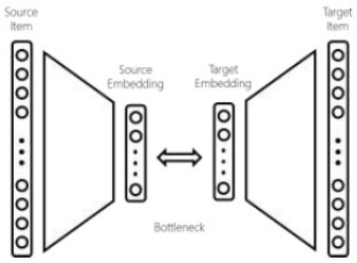
Factorize word-context matrix.

	$Context_1$	$Context_1$	....	Context $_k$
$Word_1$				
$Word_2$				
⋮				
$Word_n$				

E.g., LDA (Word-Document),
GloVe (Word-NeighboringWord)

Neural Networks

A neural network with a bottleneck, word and context as input and output respectively.



E.g., Word2vec (Word-NeighboringWord)

Language Modeling Taks

To train models to learn vector representations of words, we typically rely on solving auxiliary tasks, often referred to as "**language modeling**" tasks. These tasks are not the ultimate goal but are designed as a means to enable the model to capture meaningful patterns in the data. The idea is that by solving these tasks, the model indirectly learns the relationships and properties of words, which are then encoded in the resulting word embeddings.

The motivation for using such tasks lies in their simplicity and effectiveness. Instead of manually defining rules or structures for word relationships, these tasks let the model discover these patterns autonomously by leveraging large amounts of text data. The embeddings learned in the process capture semantic and syntactic relationships between words, which can then be used for various downstream applications.

Some common language modeling tasks used to train word embeddings include:

1. **Next-word prediction**: This involves predicting the next word in a sequence, such as predicting "sunset" in the phrase "Watching the beautiful ___". E.g. this task is used to train models like GPT or BERT.

2. **Fill-in-the-blank tasks**: These tasks, like cloze tests, require the model to fill in missing words or phrases in a sentence, helping it learn contextual word representations.
3. **Predicting a word given its context**: For example, in a sentence like “The cat sits on the ___,” the model predicts the missing word "mat" based on the surrounding words.
4. **Predicting the surrounding words given a center word**: Here, the task is reversed, and the model predicts words like "The" and "sits" when given the center word "cat".

While the ultimate aim is not to perfectly solve these tasks, the process of attempting to do so forces the model to develop a structured understanding of word meanings and relationships. The embeddings learned as a byproduct of solving these tasks encode these insights in a dense, continuous vector space.

For example, embeddings trained using these proxy tasks excel in the **Word Analogy Task**, which involves solving analogies such as:

- "Man is to woman as king is to ____?"
- "Good is to best as smart is to ____?"
- "China is to Beijing as Russia is to ____?"

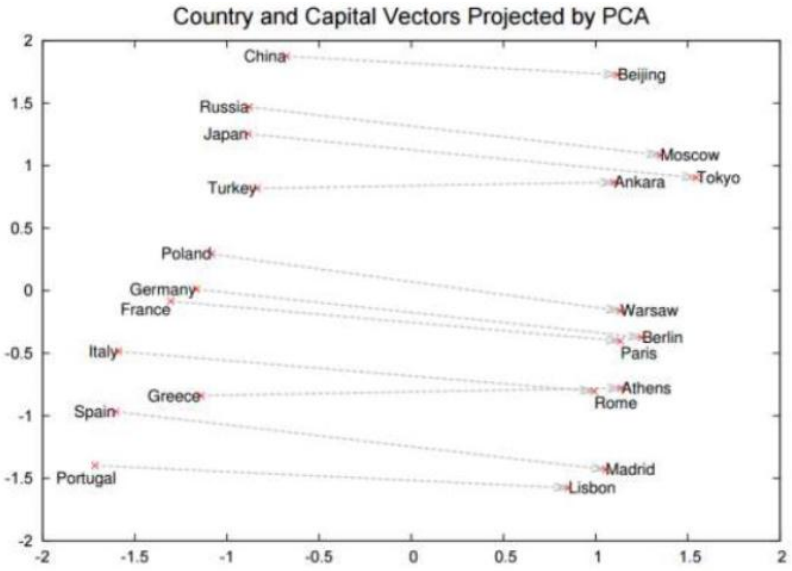
For example, the relationship between "man" and "woman" can be extended to "king" and "queen." Mathematically, if we compute the vector difference between "king" and "man" and then add the vector for "woman," the result is a vector that is closest to the vector representation of the word “queen”.

$$[king] - [man] + [woman] \approx [queen]$$

For example, the representation of dense vectors (embeddings) of these concepts can result as:

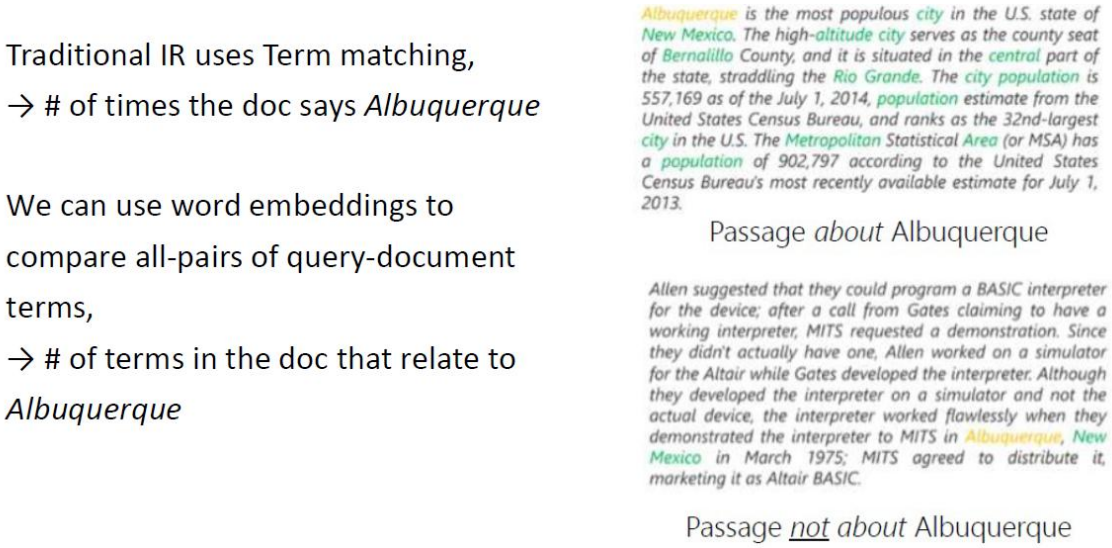
$$\underset{king}{\begin{bmatrix}0.8\\0\\0\\0.5\\-1\end{bmatrix}} - \underset{man}{\begin{bmatrix}0.2\\0\\0\\0.3\\-1\end{bmatrix}} + \underset{woman}{\begin{bmatrix}0.2\\0\\0\\0.1\\1\end{bmatrix}} = \underset{pred.}{\begin{bmatrix}0.8\\0\\0\\0.3\\1\end{bmatrix}} \approx \underset{queen}{\begin{bmatrix}0.8\\0\\0\\0.4\\1\end{bmatrix}}$$

Visually, in a word embedding space, relationships like the one between "China" and "Beijing" are almost parallel to those between "Russia" and "Moscow." These parallels reflect how embeddings encode not just individual word meanings but also relationships between them.



Word embeddings for Document Ranking: Word embeddings are not just limited to word-to-word relationships; they also significantly enhance document ranking in Information Retrieval (IR) systems. Traditional IR techniques often rely on exact term matching, such as counting the number of times a specific term like "Albuquerque"

appears in a document. By contrast, embeddings allow us to compare all pairs of terms between a query and a document, capturing semantic similarities even when the exact terms don't match. For example, embeddings can identify terms in a document that are related to "Albuquerque," enabling more nuanced and effective ranking of relevant documents.



Word2Vec

Word2Vec is a set of models developed by Google AI for **generating word embeddings**.

It is able to take a large corpus of text in input and map it in a vector space of a given dimensionality (typically of several hundred dimensions), with each unique word in the corpus being assigned a corresponding vector in the space.

Word2Vec leverages the idea of learning embeddings such that words appearing in similar contexts have similar vector representations. So, the core principle behind Word2Vec is simple:

Word similarity = vector similarity

and for reach it uses the **context** of a word—the words surrounding it—to provide its meaning.

For example, in the sentences:

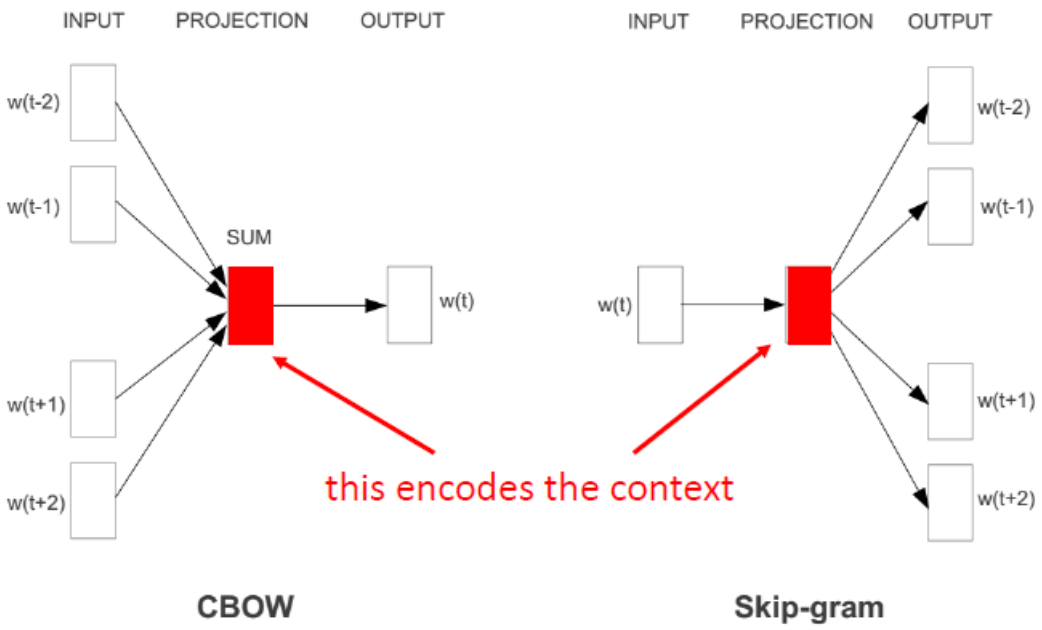
«*Rome is the capital of the country Italy*»
«*London is the capital of the country UK*»

After we are trained Word2Vec we will obtain that:

Rome is similar to *London* as they share the same context *capital*
Italy is similar to *UK* as they share the same context *country*

To generate these embeddings from the words Word2vec provides two basic Neural Network models:

- **Continuous Bag of Word (CBOW)**: predict the middle (target) word from its surrounding words.
- **Skip-gram (SG)**: predict the surrounding words from the target word (it does the reverse).



One fantastic fact you should notice from the Figure above is that Word2vec models consist of just a simple two-layer neural network, which makes Word2Vec closer in complexity to traditional machine learning rather than deep learning. However, a lot of DL applications involving text have shown improvements after using Word2vec embeddings as features.

To really understand Word2Vec you should understand how its two algorithms work. In this chapter we will just see in detail how CBOW works.

Continuous Bag of Words (CBOW)

The **Continuous Bag of Words (CBOW)** model tries to predict the target middle word based on its surrounding words, known as context words. By leveraging patterns in the relationship between context and target words across a large text corpus, CBOW learns embeddings that represent semantic relationships between words.

Without wasting more time let's see how it works.

To begin, we must choose a corpus of text—this could be a collection of sentences or paragraphs—from which we intend to learn. Given it we split the text into individual words (and additionally convert them to lowercase for uniformity). This operation is also known as ***tokenization***, though it represents one of the simplest forms of tokenization.

Note: In general, **Tokenization** refers to the process of breaking a text into smaller units, called **tokens**, which can be words, subwords, or even characters. Tokens serve as the fundamental building blocks for natural language processing (NLP). More advanced tokenization techniques exist to handle the intricacies of natural language, including methods such as N-grams, regular expressions, Byte Pair Encoding (BPE), and Penn Treebank tokenization. Additionally, popular NLP libraries like SpaCy and Gensim offer robust tokenization functionalities.

Once the text is tokenized, we extract the unique tokens from the corpus to form a vocabulary. The **vocabulary** is essentially a collection of all distinct words in the corpus. Generally, we map each word of the vocabulary to a unique index, which is helpful to create one-hot vectors. Specifically, if the vocabulary size is V, each token will be represented as a **one-hot vector** of length V (so, length equal to the size of the vocabulary) with a single 1 at the index corresponding to that token in the vocabulary, and 0s elsewhere.

For example, suppose that our corpus is made of the following sentences:

“The cat sat on the floor”

“I always sat on my favorite chair”

“Just sat on the sofa”

After tokenization (splitting into words) and converting words to lowercase, the tokens obtained are:

“the”, “cat”, “sat”, “on”, “the”, “floor”;

“I”, “always”, “sat”, “on”, “my”, “favorite”, “chair”;

“just”, “sat”, “on”, “the”, “sofa”;

From this, the vocabulary is constructed by taking the distinct tokens (set of tokens):

V: {‘the’, ‘cat’, ‘sat’, ‘on’, ‘floor’, ‘I’, ‘always’, ‘my’, ‘favorite’, ‘chair’, ‘just’, ‘sofa’}

and then a word-to-index mapping assign is done:

{ ‘the’: 0, ‘cat’: 1, ‘sat’: 2, ‘on’: 3, ‘floor’: 4, ‘I’: 5, ‘always’: 6, ‘my’: 7, ‘favorite’: 8, ‘chair’: 9, ‘just’: 10, ‘sofa’: 11}

We can use this mapping to represent each token as a one-hot encoded vector. For instance, the token "cat", which has an index of 1 in the vocabulary mapping, is represented by the one-hot vector:

$$\begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}$$

Similarly, the one-hot vectors for the rest of the tokens in the vocabulary would look like this:

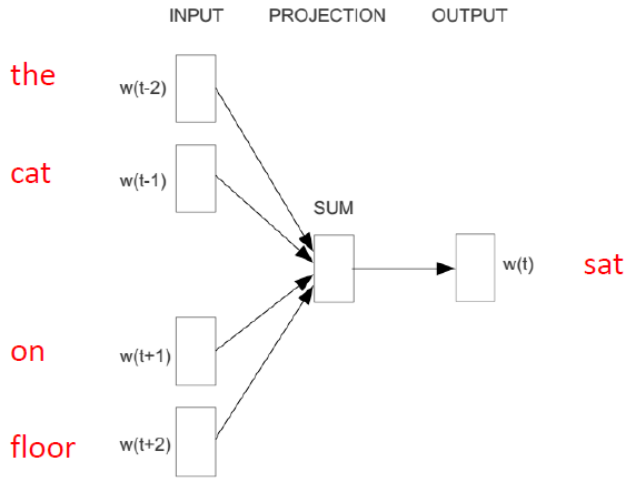
the	cat	sat	on	floor	I	always	my	favorite	chair	just	sofa
$\begin{bmatrix}1\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\1\\0\\0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\0\\0\\0\\1\\0\\0\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\\1\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\\0\\1\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0\\1\end{bmatrix}$

For simplicity, in subsequent discussions, the terms "token" and "word" will be used interchangeably.

At this stage, we need to define a **context window size (n)**, which specifies the number of context words to consider on either side of a target word. For instance, if $n = 2$, we take two words to the left and two words to the right of the target word as its context. For example, given the sentence:

"The cat sat on the floor"

If the target word is **"sat"**, the context words for $n = 2$ would be: **["the", "cat", "on", "floor"]**.

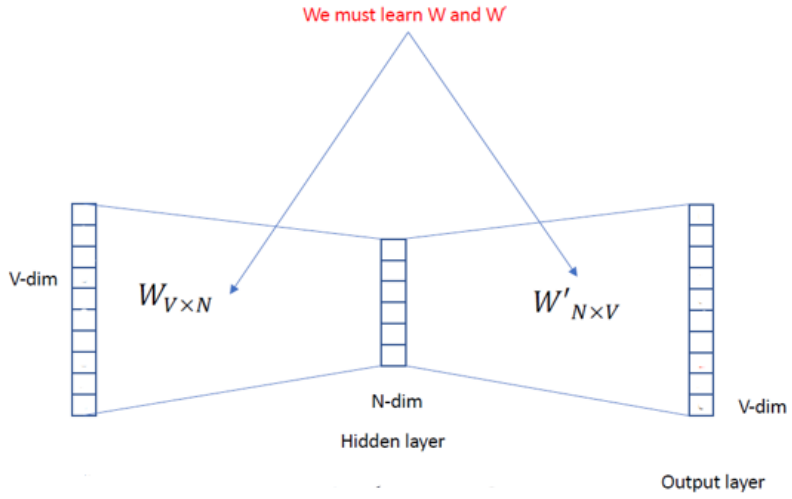


Given it we can iterate through the entire corpus and form **context-target pairs** by extracting the target word and its surrounding context words based on the specified window size. This step finalizes the **data preparation phase**, and we are ready to introduce the architecture of the model.

The model's architecture, as illustrated in the Figure below, is very simple and consists of just two layers:

1. A **hidden layer** for generating embeddings.
2. An **output layer** to predict the target word as a probability distribution.

where the **input** to the network is a one-hot vector of the context word.



You can notice that, as part of the model, we maintain two embedding matrices W and W':

- The matrix W has dimensions $V \times N$, where V is the size of the vocabulary and N is the embedding dimension (a hyperparameter). Notably, this W matrix is **shared** among all the context words.
- The matrix W' has dimensions $N \times V$.

At the beginning of training, both embedding matrices W and W' are initialized with random values. During the training process, the model learns the optimal values for these matrices, ensuring they capture meaningful relationships between words.

Note: The embedding dimension N is a hyperparameter that plays a critical role in the model's performance. A larger embedding dimension allows the model to capture more meaningful relationships between words, but it also increases the computational complexity and risk of overfitting. Determining the optimal size for N is typically achieved through trial and error, balancing the need for expressiveness with computational efficiency.

Now that the data preparation and architecture have been outlined, let's delve into the details of how the model works.

We begin by selecting a context-target pair and converting both the context words and the target word into one-hot vectors as described earlier. The context word one-hot vectors x_i and the target word one-hot vector y are each of size $V \times 1$.

The context word one-hot vectors are then passed in input to the network. Each of these vectors is then transformed into its corresponding dense word embedding by multiplying it with the embedding matrix W:

$$v_i = W^T \times x_i$$

where $\boldsymbol{v_i}$ is the resulting word embedding for the i-th context word. The size of these dense word-embedding v_i is $N \times 1$. Here, basically, since x_i is a one-hot vector with a 1 at the index of the given context word and 0s elsewhere, multiplying it with the embedding matrix W effectively selects the corresponding row from W, which contains the embedding representation for that context word (if this is not clear now it will be clearer at the end).

Notes: It's important to note that, unlike the context word one-hot vectors x_i which are sparse vectors (vectors with mostly zero values), here these word-embedding v_i are encoded by continuous dense vectors (e.g. $[0.8, 0, 0.2, 0.4, -1.2]$) that capture meaningful semantic characteristics of the words.

At this point, these word embeddings are aggregated to form a single meaningful embedding N-dimensional (always of size $N \times 1$) that represents the combined context of the surrounding words. There are tow ways we can aggregate these embeddings:

- **Average**: The aggregated embedding $\hat{v}$ is computed as the **average** of the individual word embeddings:

$$\hat{v} = \frac{1}{2n} \sum_{i=1}^{2n} v_i$$

- **Sum**: The aggregated embedding $\hat{v}$ is computed as the **sum** of the individual word embeddings:

$$\hat{v} = \sum_{i=1}^{2n} v_i$$

The aggregated context embedding $\hat{v}$ in output from the hidden layer is then passed through a second matrix W', resulting in a V-dimensional (of size $V \times 1$) vector z representing the raw scores (**logits**) for each word in the vocabulary:

$$z = W'^T \times \hat{v}$$

The logits are then fed into a **softmax function** to obtain a probability distribution over all words in the vocabulary:

$$\hat{y} = softmax(z)$$

where each probability $\hat{y}_i$ is given by:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{V} e^{z_j}}$$

The softmax function converts the raw scores into probabilities, where the highest probability corresponds to the predicted target word.

The **cross-entropy loss** is then used to measure the error between this predicted probability distribution $\hat{y}$ and the actual target word (ground truth) one-hot vector y:

$$L = -\sum_{i=1}^{m} y_i \log(\hat{y}_i)$$

This loss is minimized using optimization techniques like Stochastic Gradient Descent (SGD) or its variants. Through backpropagation, the error is propagated through the model, adjusting the weights in the two embedding matrices W and W' to improve the word embeddings over time. This process is repeated for all context-target pairs in the corpus until the model converges.

At the end of training, the word embeddings learned are stored in two embedding matrices, W and W'. Specifically, each row of matrix W corresponds to the word embedding for a given word, alternatively each column of matrix W' corresponds to the word embedding for a given word.

You can use either matrix W or W' as the words representation (embeddings), or even take the average of both, to obtain more robust embeddings.

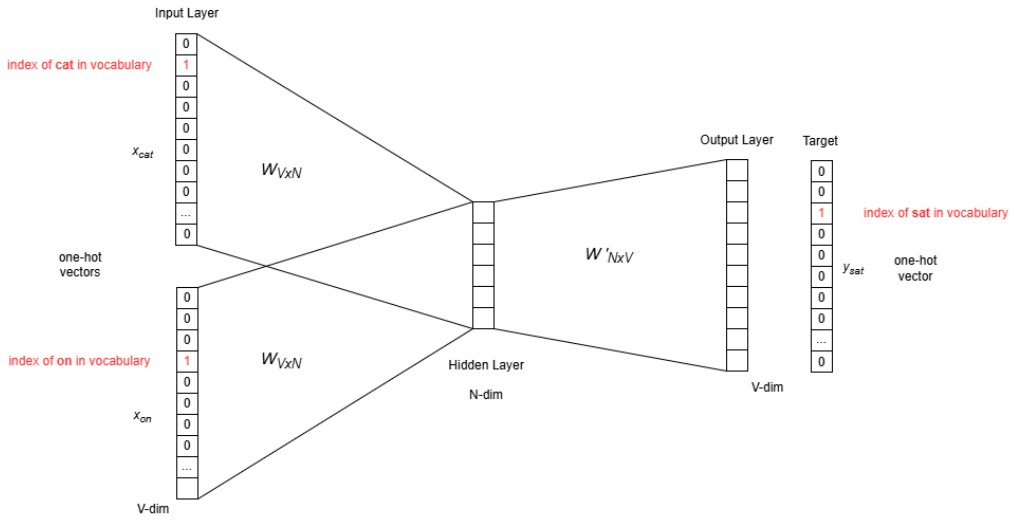
Example

To illustrate this process, let's revisit the example we discussed earlier. Suppose we choose a context window size of $n = 1$, which means we will have two context words—one from each side of the target word.

Consider the following context-target pair:

- Target word: "sat"
- Context words: ["cat", "on"]

First, we retrieve the one-hot encoded vectors for the words in the context x_{cat} and x_{on} and for the target y_{sat} based on our vocabulary:



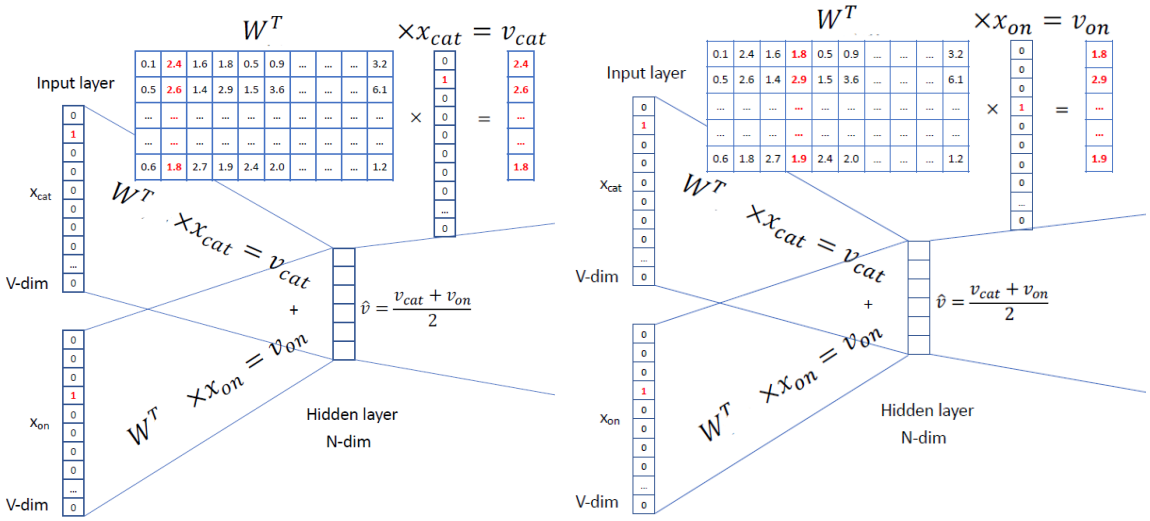
Now, these one-hot vectors x_{cat} and x_{on} are passed in input to the network. Each one is transformed into its corresponding dense word embedding using the embedding matrix W:

$$v_{cat} = W^T \times x_{cat}$$

$$v_{on} = W^T \times x_{on}$$

The hidden layer then takes these two word-embeddings and for instance average them to compute a single aggregated embedding:

$$\hat{v} = \frac{v_{cat} + v_{on}}{2}$$



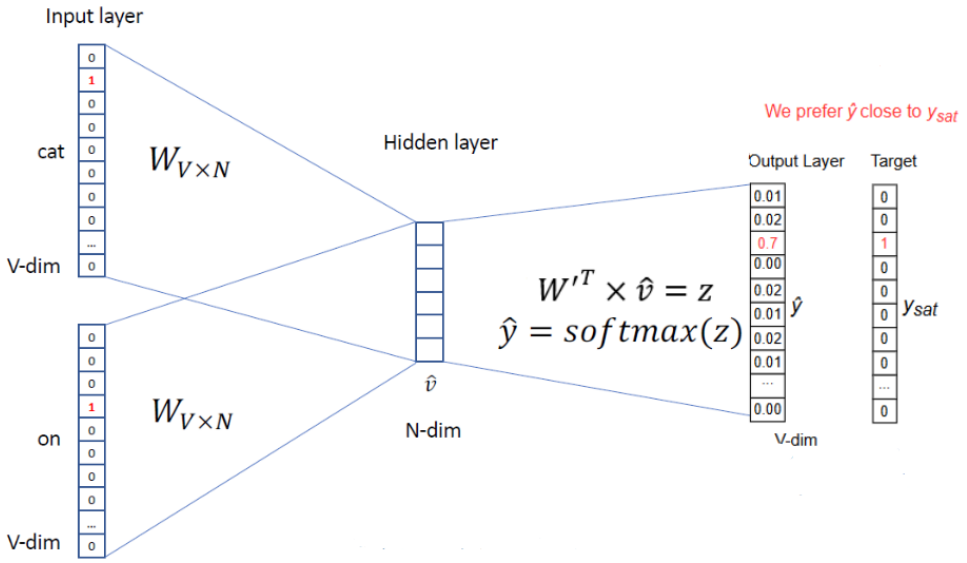
Now, the aggregated embedding $\hat{v}$ is passed through the second matrix W', producing a $V \times 1$ vector z (raw scores or logits):

$$z = W'^T \times \hat{v}$$

and these logits z are then passed through a softmax function to obtain a probability distribution $\hat{y}$ over the vocabulary:

$$\hat{y} = softmax(z)$$

At this point, we compare the predicted probability distribution $\hat{y}$ with the actual target word's one-hot vector y_{sat} using the cross-entropy loss function.



Additional Considerations on Word2Vec:

Word2vec is very fast, i.e. it takes less than a day to learn high quality word vectors from a 1.6 billion words dataset (large corpus) and can easily incorporate a new sentence/document or add a word to the vocabulary.

In real-world applications, Word2Vec is typically trained on massive text corpora, such as datasets containing around 13 million words. Commonly used parameters include a context window size $n = 5 - 10$ words and an embedding dimension N$\approx 300 - 500$. While training on such large datasets requires substantial computational resources, an alternative is to use pre-trained Word2Vec models.

To improve the quality of the embeddings, consider the following strategies:

- **Model architecture choice**:
    - For larger corpora, use the **Skip-gram** model which is slower but more accurate.
    - For smaller corpora, the **CBOW** model is faster and more efficient.
- **Increase the size of the training dataset** (which often leads to better results).
- **Increase the embedding dimension** to capture more semantic information.
- **Use additional optimization techniques** such as **Negative Sampling** or **Hierarchical Softmax** (introduced in this paper for Skip-Gram model).

12.7 Autoencoders

Autoencoders are a type of artificial neural network primarily used for **dimensionality reduction**. The main goal of autoencoders is to learn a lower-dimensional representation of input data which is able to maintain the most important information.

An autoencoder consists of two main components:

1. **Encoder**: Transforms the input data into a **lower-dimensional representation**, also known as **latent representation**.
2. **Decoder**: Tries to reconstruct the input data from the compressed representation.

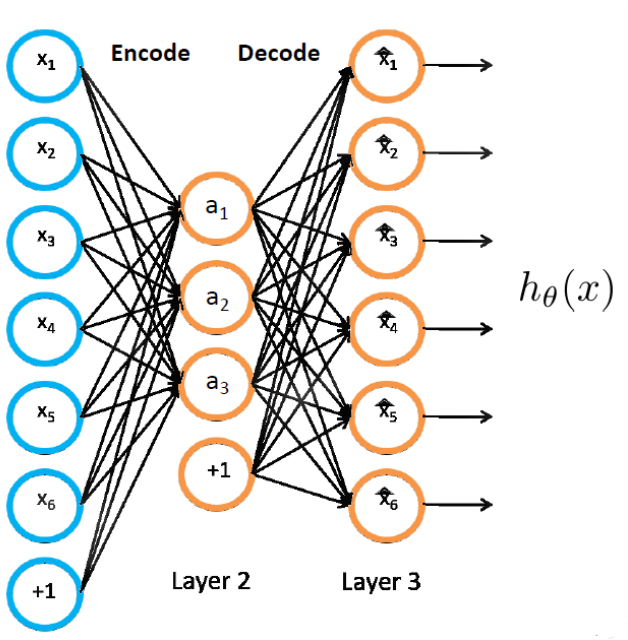
So, autoencoders learn to encode input data into a lower-dimensional representation and then from that reconstruct the original data as closely as possible.

The network is trained to output the input (learn identify function), aiming for:

$$h_\theta(x) \approx x$$

However, this leads to a trivial solution unless constraints are applied:

- **Limiting the number of units** in the last layer of the encoder (which outputs the latent representation) to enforce compression (**Autoencoder**).
- **Imposing sparsity** on the last layer of the encoder (**Sparse Autoencoder**).



An autoencoder takes an input $x \in [0,1]^d$ and first maps it (with an encoder) to a latent representation $z \in [0,1]^{d'}$ through a deterministic mapping

$$z = s(Wx + b)$$

where s is a non-linear activation function (such as sigmoid).

The latent representation z is then mapped back (with a decoder) into a reconstruction $\tilde{x}$ of the same shape as x,

$$\tilde{x} = s(W'z + b')$$

Here, $\tilde{x}$ is seen as a prediction of x.

Autoencoder Loss & Sparse Autoencoder

When training autoencoders, the goal is to minimize the **reconstruction error** between the original input x and its reconstruction $\tilde{x}$ (which we indicate with $h_\theta(x)$):

$$\min_\theta \|h_\theta(x) - x\|^2$$

This means that the encoder must find a latent representation that retains all the important information of the original input, while the decoder must be able to use this latent representation to reconstruct the original input as accurately as possible.

Moreover, during training, we can also apply an $l1$ regularization to our autoencoder to create a **sparse autoencoder**:

$$\min_{\theta} \|h_\theta(x) - x\|^2 + \lambda \sum_i |a_i|$$

Here, a_i represents the activation of the i-th neuron and λ is the regularization parameter.

The added $l1$ regularization term encourages many of the weights to reach zero, making the autoencoder sparser. Sparsity ensures that only a **subset of neurons** is activated, which helps identify the most **relevant features** of the data.

Why Sparsity?

- Sparse representations focus on dominant dimensions while ignoring irrelevant ones.
- Matrices with sparse weights (few non-zero values) activate fewer neurons, leading to a more efficient and interpretable model.

Deep Autoencoder

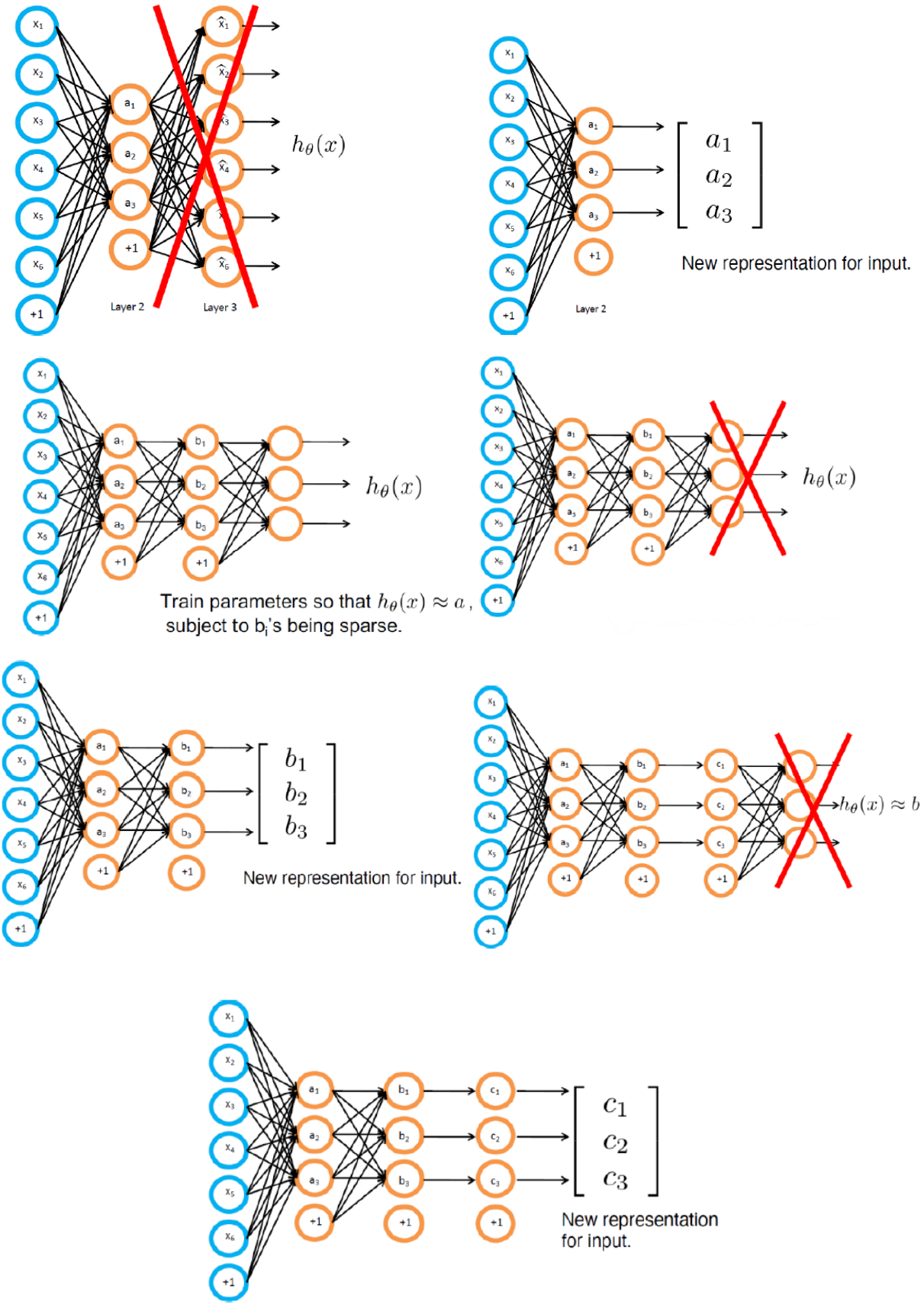
The topic we will address now extends beyond the scope of a standard machine learning course, however since it will re-appear in the deep learning course, analyzing it here will be beneficial for a better understanding later.

A **deep autoencoder** refers to an autoencoder with multiple hidden layers. Training such deep architectures can present challenges, one of which is the **vanishing gradient problem**. We already introduced this issue in the Neural Networks chapter, when we said that it arises when the gradients used to update weights during backpropagation diminish as they pass through many layers, resulting in values near to zero for earlier layers which essentially "shut down" the learning for them. We also saw that a key cause is the use of saturating activation functions such as sigmoid and tanh.

When autoencoders were first invented, many of the modern advancements in deep learning—such as non-saturating activation functions (e.g., ReLU) and advanced weight initialization techniques (e.g., Xavier and He initialization)—were not yet developed. Consequently, training deep neural networks, including deep autoencoders, was prone to significant difficulties like vanishing gradients and poor initialization. To address these issues, an intriguing solution called *layerwise training* was used.

Layerwise training works in the following manner:

1. Initially, a network with a single hidden layer is trained, where the desired output is set to match the input. This enables the first hidden layer to learn a latent representation of the input.
2. Next, the output layer is cut, and a new network with a single hidden layer is trained using the previous network's hidden layer as the input.
3. This procedure is repeated to train additional hidden layers, where each layer learns a more *abstract latent representation* of the input.
4. *Optional*: After all hidden layers are trained, for *supervised learning* a final output layer can be added.

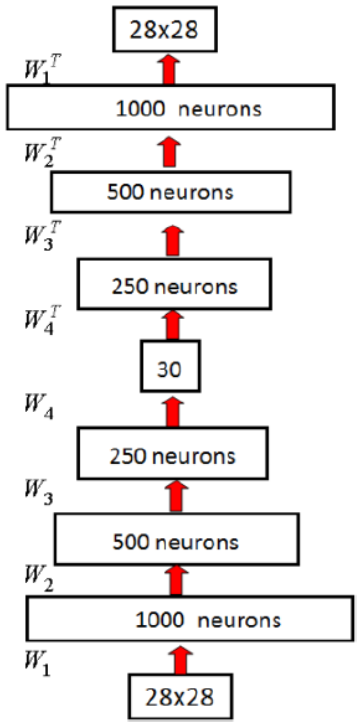


Optionally, at the end, the network can also undergo *fine-tuning* to adjust the weights globally.

This layerwise training approach offered several benefits. It ensured that weights were initialized meaningfully, based on the training of the previous layer, rather than being set randomly. Although the vanishing gradient problem could still occur in the lower layers (the first layers) during the final stages of pretraining (when training the last layers), its impact was minimal because these lower-layer weights had already been learned effectively in earlier stages, when the network had fewer hidden layers. As a result, the weights in these layers were already set to sensible values, reducing the difficulties associated with poor initialization and ineffective gradient propagation.

During the early days of deep learning, architectures such as **Deep Belief Networks (DBNs)** and **Deep Boltzmann Machines (DBMs)** were widely used for layerwise training. While these methods were foundational for enabling the training of deep architectures, advancements such as non-saturating activation functions, better initialization techniques, powerful optimization algorithms, and increased computational resources have largely rendered them unnecessary. Today, **end-to-end training** has become the standard approach, making it easier to train deep networks directly without the need for these intermediate steps.

Deep Autoencoder in Images



In the Figure below we can see an example of a **Deep Autoencoder** made of a stack of multiple encoder and decoder layers used for image dimensionality reduction.

In this particular example, the input data is a 28x28 image. This is passed through the encoder layers with gradually lower dimensions. The output of the last encoder layer is a latent representation of the input image, consisting of only 30 features. This latent representation is then passed through the corresponding decoder layers, which reconstruct the original image from the latent representation.

Using a stack of multiple encoder and decoder layers, the network can effectively learn a hierarchical representation of the input data, where each layer captures increasingly complex and abstract characteristics of the data.

Autoencoder vs PCA

Autoencoders go beyond creating a simple **linear combination** of the original data space, unlike PCA. Instead, they use **nonlinear activation functions** to map the input data to a **more complex latent space**, enabling the preservation of richer and more intricate information.

To better understand this, we can draw a parallel to **Kernel PCA**:

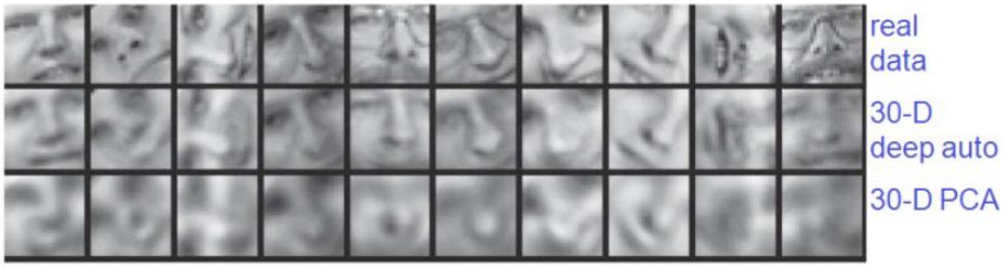
- Kernel PCA maps the data into a lower-dimensional space using a predefined kernel function to capture nonlinearity.

- The autoencoder essentially does the same job as Kernel PCA, but **without the need to explicitly define a kernel function**. Instead, they learn the nonlinear mapping **directly from the data** in a more efficient and adaptive way.

Autoencoders generally outperform PCA and Kernel PCA because they learn these nonlinearities more effectively and adaptively during training.

The Figure below shows the results of applying different dimensionality reduction techniques to the same data (in this case images); in particular, a 30-D deep autoencoder, a 30-D logistic PCA and a 30-D PCA are used.

As we can see, the difference between the effect of PCA and deep autoencoder is quite significant. This is because the deep autoencoder has learned a more complex latent space that takes into account more abstract characteristics of the data. Therefore, it is able to preserve more information during dimensionality reduction than simpler PCA techniques.



Some Other Important Principles in Machine Learning

In addition to the core concepts and algorithms discussed so far, there are several overarching principles that serve as guiding frameworks in machine learning.

Occam's (Ockham's) Razor

Occam's razor is a guiding principle in machine learning for selecting the best algorithm among various options. The key idea is to prioritize simplicity in explanations or models, based on the following considerations:

- **The simplest answer is usually the correct answer**
- Simplicity is the ultimate sophistication
- Of two equivalent theories or explanations, all other things being equal, the simpler one is to be preferred
- We are to admit no more causes of natural things than such as are both true and sufficient to explain their appearances
- The simplest explanation is usually the best

No free Lunch (NFL) Theorem

The **No Free Lunch (NFL) theorem** states:

- If an algorithm performs well on a certain class of problems, then it necessarily performs bad on the set of all remaining problems.

Theorem 1: For any pair of algorithms a_1 and a_2

$$\sum_f P(d_m^y|f,m,a_1) = \sum_f P(d_m^y|f,m,a_2).$$

- For any pair of algorithms a_1 and a_2, the average of the performance over all the optimization problem f is always the same.

Implications of NFL theorem:

- If an algorithms performs particularly well for a class of problems, then it must do worse on average over the remaining problems
- If an algorithms performs better than random search on some class of problems, then it must performs worse than random search on the remaining problems
- No algorithm can give good results on all the problems. You should not generalize good results obtained on some problems to other problems.